Ph.D. Thesis

Scalable Indexing and Exploration of Big Time Series Data

Author:
Georgios Chatzigeorgakidis

Supervisor:
Professor Spiros Skiadopoulos

21 Οκτωβρίου 2019

Κλιμακώσιμη Δεικτοδότηση και Εξερεύνηση Μεγάλων Δεδομένων Χρονοσειρών

Διδακτορική Διατριβή
του
Γεωργίου Κ. Χατζηγεωργακίδη

Διπλωματούχου Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών Πολυτεχνείου Κρήτης (2011) και MSc Πολυτεχνείου Δανίας (2014)

Συμβουλευτική Επιτροπή: Σπύρος Σκιαδόπουλος Επιβλέπων καθηγητής
Χρήστος Τρυφωνόπουλος
Θοδωρής Δαλαμάγκας

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την XX/XX/2019

Όνομα	Βαθμίδα	Ίδρυμα
Σπύρος Σκιαδόπουλος	Καθηγητής	Παν. Πελοποννήσου
Χρήστος Τρυφωνόπουλος	Αν. Καθηγητής	Παν. Πελοποννήσου
Θοδωρής Δαλαμάγκας	Ερευνητής Α	ΙΠΣΥ / Ε.Κ. ΑΘΗΝΑ
XX	XX	XX
XX	XX	XX
XX	XX	XX
XX	XX	XX

Η παρούσα έρευνα έχει συγχρηματοδοτηθεί από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο - ΕΚΤ) και από εθνικούς πόρους μέσω του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» του Εθνικού Στρατηγικού Πλαισίου Αναφοράς (ΕΣΠΑ) – Ερευνητικό Χρηματοδοτούμενο Έργο: Ηράκλειτος ΙΙ. Επένδυση στην κοινωνία της γνώσης μέσω του Ευρωπαϊκού Κοινωνικού Ταμείου.

Ευχαριστίες

[...]

Abstract

Time series are generated and stored at a vastly increasing rate in many industrial and research applications, including the Web and the Internet of Things, public utilities, finance, astronomy, biology, and many more. A significant portion concerns *geolocated* time series, i.e., those generated at, or otherwise associated with specific locations. Although several works have focused on efficient time series *similarity search*, there has been limited attention to the inherent challenge that geolocated time series introduce for hybrid queries on both spatial proximity and time series similarity. To efficiently process such queries, we propose a *hybrid index*, called BTSR-tree. Furthermore, we address the problem of *hybrid similarity joins* over such geolocated time series. We propose both centralized and MapReduce-based algorithms for performing such join operations using spatial-only, time series-only, and hybrid indices.

Although several works exist for time series *visualization* and visual analytics in general, there is a lack of efficient techniques for visual exploration and analysis of geolocated time series in particular. In this Thesis, we present two approaches that rely on hybrid indices to allow for efficient map-based visual exploration and summarization of geolocated time series data. In particular, we use the BTSR-tree index and we introduce a new variant of the standard iSAX index, called geo-iSAX. We describe the structure of the new index and show how they can be directly exploited to produce map-based visualizations of geolocated time series at different levels of granularity.

Apart from traditional similarity search, we also consider the problem of detecting *locally* similar pairs and groups, called *bundles*, over *co-evolving* time series. These are pairs or groups of subsequences whose values do not differ by more than a predefined threshold for a number of consecutive timestamps, thus indicating common local patterns and trends. We propose a filter-verification technique that only examines candidate matches at judiciously chosen checkpoints across time. In the same line of work, we consider hybrid queries for retrieving geolocated time series based on filters that combine spatial distance and time series local similarity. To efficiently support such queries, we introduce the $\mathcal{S}$BTSR-tree index, an extension of BTSR-tree that further optimizes local similarity search.

Finally, we focus on large-scale *forecasting* on big time series data. Specifically, we introduce FML-kNN, a novel distributed processing framework for Big Data that performs probabilistic classification and regression. The framework's core is consisted of a k-nearest neighbor joins algorithm which, contrary to similar approaches, is executed in a single distributed session and is able to operate on very large volumes of data of variable granularity and dimensionality.

Throughout this Thesis, we experimentally and empirically evaluate our work using synthetic and real-world datasets from diverse domains, against baseline and state-of-the-art existing methods, demonstrating the efficiency and superiority of our approaches.

Περίληψη

Στις μέρες μας, σε πολλές βιομηχανικές και ερευνητικές εφαρμογές (π.χ., διαδίκτυο των πραγμάτων, αστρονομία, οικονομικά, βιολογία) δημιουργείται και αποθηκεύεται μεγάλος όγκος δεδομένων *χρονοσειρών*. Ένα σημαντικό ποσοστό αυτών αποτελούν οι *γεωχωρικές* χρονοσειρές, δηλαδή εκείνες οι οποίες δημιουργούνται και σχετίζονται με συγκεκριμένες τοποθεσίες. Τα τελευταία χρόνια, πληθώρα επιστημονικών άρθρων μελετά μεθόδους *αναζήτησης ομοιότητας* σε δεδομένα χρονοσειρών αψηφώντας τη γεωχωρική τους υπόσταση, η οποία θα επέτρεπε παρόμοια ερωτήματα βασισμένα –εκτός από την ομοιότητα στο πεδίο του χρόνου– στη χωρική εγγύτητα των χρονοσειρών. Στη διατριβή αυτή, παρουσιάζουμε ένα *υβριδικό* ευρετήριο με το όνομα BTSR-tree, το οποίο μπορεί αποδοτικά να απαντήσει τέτοιου είδους υβριδικά ερωτήματα αναζήτησης ομοιότητας. Επιπροσθέτως, επικεντρωνόμαστε στο πρόβλημα των *υβριδικών ενώσεων ομοιότητας* σε δεδομένα γεωχωρικών χρονοσειρών. Για την επίλυσή του, προτείνουμε κεντρικούς και κατανεμημένους αλγορίθμους βασισμένους στη μέθοδο MapReduce, με τη χρήση υβριδικών και μη ευρετηρίων.

Πληθώρα επιστημονικών άρθρων επικεντρώνεται στην *οπτικοποίηση* και οπτική ανάλυση δεδομένων χρονοσειρών. Η αποδοτική οπτική εξερεύνηση γεωχωρικών χρονοσειρών, όμως, δεν έχει μελετηθεί επαρκώς. Στην παρούσα διατριβή, παρουσιάζουμε δυο προσεγγίσεις βασισμένες σε υβριδικά ευρετήρια, οι οποίες επιτρέπουν την αποδοτική εξερεύνηση δεδομένων γεωχωρικών χρονοσειρών μεγάλου όγκου με τη χρήση οπτικοποιήσεων σε χάρτη. Για την πρώτη, χρησιμοποιούμε το προαναφερθέν υβριδικό ευρετήριο BTSR-tree. Η δεύτερη προσέγγιση βασίζεται σε μια επέκταση του υπάρχοντος ευρετηρίου χρονοσειρών iSAX. Συγκεκριμένα, παρουσιάζουμε τη δομή του νέου ευρετηρίου και μεθόδους αποδοτικής οπτικοποίησης τέτοιων δεδομένων.

Εκτός των ανωτέρω, στην παρούσα διδακτορική διατριβή, επικεντρωνόμαστε στο πρόβλημα εντοπισμού ζευγαριών η ομάδων *τοπικά* όμοιων συν-εξελισσόμενων χρονοσειρών. Συγκεκριμένα, τα ζευγάρια (ή ομάδες) αυτά αποτελούνται από χρονοσειρές των οποίων οι τιμές σε οποιαδήποτε υποακολουθία τους δεν διαφέρουν περισσότερο από ένα δωθέν κατώφλι. Ο εντοπισμός τέτοιων ζευγαριών (ή ομάδων) μπορεί να φανερώσει χρήσιμα κοινά τοπικά μοτίβα και τάσεις σε δεδομένα χρονοσειρών. Για την εύρεσή τους, προτείνουμε μια μέθοδο φιλτραρίσματος-επαλήθευσης, η οποία επικεντρώνεται σε συγκεκριμένα σημεία ελέγχου στο πεδίο του χρόνου, επιταγχύνοντας τη διαδικασία. Παράλληλα, προτείνουμε μεθόδους απάντησης υβριδικών ερωτημάτων τοπικής ομοιότητας σε δεδομένα γεωχωρικών χρονοσειρών μεγάλου όγκου. Για την υποστήριξη τέτοιων ερωτημάτων, εισάγουμε μια επέκταση του ευρετηρίου BTSR-tree, με το όνομα $\mathcal{S}$BTSR-tree, το οποίο βελτιστοποιεί την βασισμένη σε τοπική ομοιότητα αναζήτηση.

Στο πλαίσιο της *πρόβλεψης* δεδομένων χρονοσειρών μεγάλης κλίματας, παρουσιάζουμε ένα παράλληλο και κατανεμημένο πλαίσιο επεξεργασίας, το οποίο εκτελεί αποδοτικά κατηγοριοποίηση και παλινδρόμηση. Ο κεντρικός αλγόριθμός του πλαισίου είναι βασισμένος στη

μέθοδο ενώσεων k-πλησιέστερων γειτόνων και μπορεί να εκτελεστεί σε μια παράλληλη συνεδρία –σε αντίθεση με παρόμοιες μεθόδους–, επιτρέποντας, έτσι, την εκτέλεση σε δεδομένα μεγάλου όγκου, διαφόρων βαθμών λεπτομέρειας και διαστατικότητας.

Τέλος, όλοι οι αλγόριθμοι και μέθοδοι που παρουσιαζονται στην διατριβή αυτή αξιολογούνται πειραματικά και εμπειρικά, με τη χρήση συνθετικών ή δεδομένων παργματικού κόσμου. Συγκεκριμένα, συγκρίνονται με βασικές, ή υπάρχουσες μεθόδους αιχμής (state-of-the-art), αποδεικνύοντας την υπεροχή τους και επιβεβαιώνοντας την αποδοτικότητά τους.

Contents

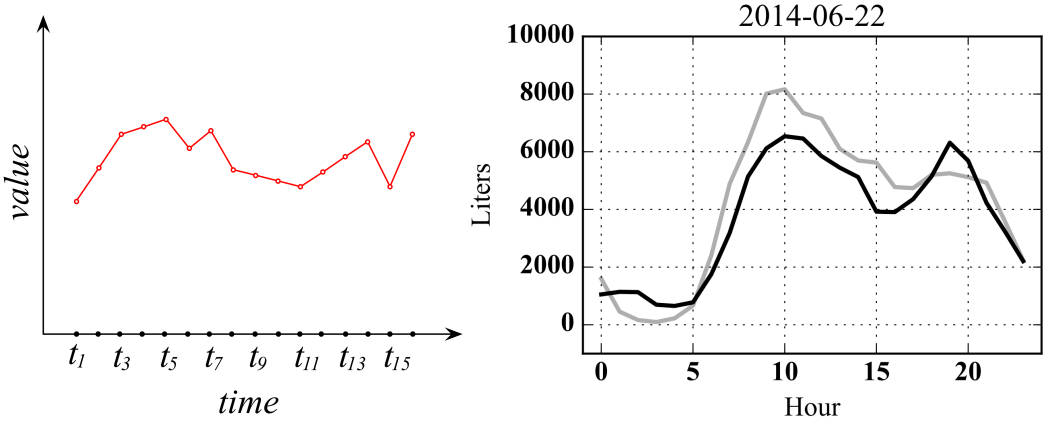
List of Figures

Chapter 1

Introduction

1.1 Concept and Motivation

Time series data is a treasure trove for a variety of mining and monitoring applications both in industry and in academia, while a rapidly increasing bulk of such data is also generated on the Web and the Internet of Things. They can represent various types of measurements, such as user check-ins at various Points of Interest, energy consumption in smart buildings, PM2.5 particle concentration measured by air pollution sensors, etc. A time series is a time-ordered sequence of values. Figure 1.1a depicts such an example. A real-world example of two time series representing the total per-hour water consumption during a specific day of two separate regions within a city, is illustrated in Figure 1.1b. Tasks such as exploring and mining time series data are highly important for discovering trends or patterns and extracting useful insights from such phenomena, having attracted extensive research interest over the last years [EZPB18, LZPK18, YZU+18, CSP+14, DTS+08, SK08].



(a) A simple time series example.

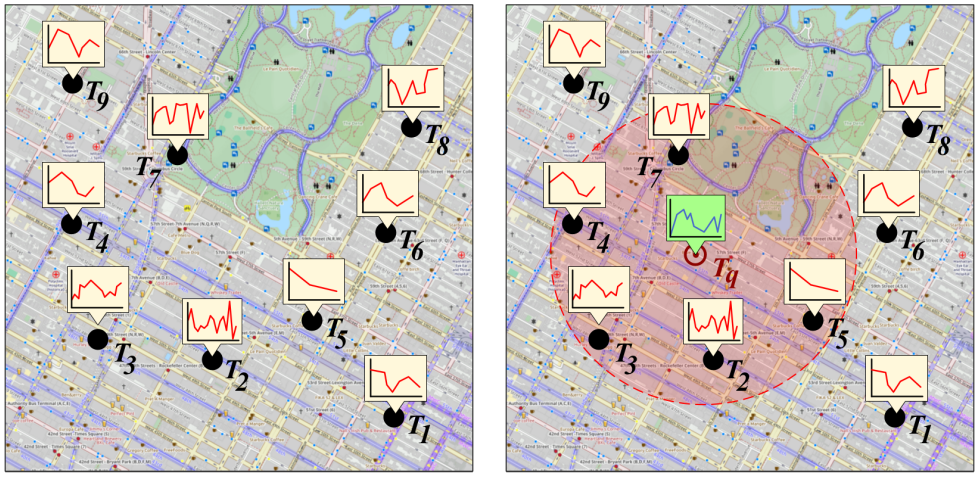
(b) Water consumption-related time series.

Figure 1.1: Examples of time series.

As an example of time series exploration, consider a large set of daily water consumption time series of a region within a city measured per hour, as the ones depicted in Figure 1.1b. Given a new time series, one could require to obtain the most similar one from the dataset,

to detect days of similar consumption. However, looping over all time series in the dataset and comparing similarities could be rather slow. To speed things up, we could build an *index* using all the time series in the dataset and then perform a similarity query on it. Several approaches have been proposed for efficiently indexing large amounts of time series data. One well-studied family of approaches includes wavelet-based methods [CF99], which rely on *Discrete Wavelet Transform* [Gra95] to reduce the dimensionality of time series and generate an index using the coefficients of the transformed sequences. Another line of work employs a *Symbolic Aggregate Approximation* (SAX) representation of time series [LKWL07], introducing a series of indices, such as *i*SAX [SK08], *i*SAX 2.0 [CPSK10], *i*SAX2+ [CSP+14], and ADS+ [ZIP14].

However, to the best of our knowledge, none of the existing works so far has considered the specific case of *geolocated time series* (i.e., produced at, or otherwise associated with, specific locations). Figure 1.2a depicts a set of geolocated time series on a map. Geolocated time series can be found in various domains and applications. For instance, time series can be used to represent, analyze and forecast water consumption measured by smart meters installed in urban households [CGA16]. Analyzing such time series can provide valuable insights regarding trends and patterns of consumer behavior in daily life. These results can then be used for customer segmentation, targeted marketing, planning future network upgrades, forecasting and balancing water demand, as well as planning and prioritizing interventions that can guide consumers towards more sensible water use. Similarly, check-ins in geosocial networks can also be modeled as geolocated time series. Analysis results can indicate nearby venues with similar frequency patterns, which may be used for social recommendations according to time, place, activity, etc. Other use cases can be found in other domains, such as in geomarketing or mobile advertisement, where geolocated time series may represent the number of visitors or the revenue generated at a certain location across time.



(a) Geolocated time series. **(b)** A query on geolocated time series.

Figure 1.2: Examples of geolocated time series.

As an example, given the set shown in Figure 1.2a and a query geolocated time series, one could request all similar ones in the time domain that are also spatially close, as illustrated in Figure 1.2b. All aforementioned indices aim at efficiently supporting similarity search; in case that the analyzed time series are associated with a spatial attribute and issued queries

involve spatial filters, these need to be treated independently. Thus, for queries employing both types of predicates, this implies evaluating each predicate separately. This can be done by first using a time series index to retrieve similar time series and then applying the spatial predicate on the results, or vice versa, by employing a spatial index to evaluate the spatial predicate and then filter the results according to their similarity with the query time series. Formally, geolocated time series can be defined as follows.

Definition 1 (Geolocated Time Series)**.** *A* time series *is a time-ordered sequence of values* $T = \{v_1, \ldots, v_w\}$*, where* v_i *is the value at the* i*-th time point and* w *is the length of the series. A time series is geolocated if it is also characterized by a location, denoted by* $T.loc$*. Assuming a 2-dimensional space,* $T.loc_x$*,* $T.loc_y$ *refer to the* (x, y) *coordinates of* T*'s location.*

Due to the rather complex nature of time series data, in conjunction with the extra spatial characteristics, such analysis tasks can be a rather slow and cumbersome procedure. To make such applications more efficient and scalable, geolocated time series need to be appropriately *indexed* to allow executing hybrid queries based on both *spatial proximity* and *time series similarity*, enhancing the pruning potential. In this thesis, we present novel approaches for efficient indexing, exploration and mining of large volumes of geolocated, as well as plain time series data. Then, we take it a step further by focusing on distributed joint forecasting on massive sets of time series.

1.2 Challenges and Contributions

Hybrid Indexing and Querying on Geolocated Time Series. In this Thesis, we introduce a *hybrid index* for efficiently supporting similarity search on geolocated time series combining both spatial proximity and time series similarity. The proposed index, called *TSR-tree*, extends the R-tree by introducing appropriate bounds for the time series indexed at each node. This reduces node accesses during query evaluation by simultaneously pruning the search space in the spatial domain and the time series domain while traversing the index. We also present an optimized version, the BTSR-tree, which uses tighter bounds by bundling together similar time series in each node. We describe how these indices can be used to efficiently evaluate different variants of hybrid queries combining spatial and time series filtering or ranking.

In the same line of work, we introduce a new type of hybrid query that applies hybrid *similarity join* on very large sets of geolocated time series. A hybrid similarity join query aims to identify *all pairs* between the two datasets qualifying to the criteria of *spatial proximity* and *time series similarity*. Clearly, performing a pairwise comparison among all pairs of objects in the two datasets is not an option when their size is large. Hence, indexing them is indispensable for efficient processing of such queries. In this work, we take advantage of the BTSR-tree's hybrid pruning potential, to deliver a more efficient and faster hybrid similarity join evaluation compared to other, adapted, state-of-the-art indices. However, this *centralized* approach has certain limitations, as it cannot sustain examination of large datasets. Hence, we further suggest a space-driven data partitioning scheme that enables a *parallel and distributed* approach for hybrid similarity joins.

In summary, in the fields of hybrid indexing and querying on geolocated tome series, our work makes the following contributions:

- We propose the TSR-tree, a hybrid index for geolocated time series, extending the spatial R-tree and augmenting each node with appropriate time series bounds.
- We further optimize the TSR-tree to derive a more efficient variant, the BTSR-tree, that clusters the time series of each subtree to derive and maintain tighter bounds for pruning.
- We address the problem of similarity search for geolocated time series, via hybrid boolean or top-k queries combining both spatial proximity and time series similarity.
- We adapt state-of-the-art indices (including the BTSR-tree index) for hybrid similarity join over geolocated time series data in centralized settings and propose traversal methods that can prune the search space and return answers without false misses.
- We suggest a space-driven partitioning method to distribute large datasets in cluster infrastructures, thus enabling faster, in-parallel evaluation of smaller similarity join tasks.
- We experimentally validate our proposed approach using real-world datasets from different application use cases, showing that our hybrid indices can effectively allow simultaneous pruning of the search space in both spatial and time series domains, significantly reducing the required number of node accesses and execution time.

The above results are published at the International Conference on Advances in Geographic Information Systems (SIGSPATIAL) 2017 [CSP+17] and 2018 [CPS+18].

Visual Exploration of Geolocated Time Series. Extracting insights, trends and patterns from large geolocated time series datasets can be significantly facilitated by *map-based visualizations* of *summarized* time series data. For instance, considering the scenario of water consumption measured by smart meters installed in urban households, such visualizations could reveal which type of consumption patterns are most frequently observed among consumers in a certain area, or what the spatial distribution of sales for a certain product looks like. However, the inherently complex nature of time series, combined with the extremely large volumes of such datasets, incommodes their management, analysis and exploration. In particular, visual exploration of geolocated time series needs to process the required information efficiently, while the user interacts with the application. For example, whenever the user zooms in or scrolls the map, visual analytics and aggregates should be computed on-the-fly, e.g., identifying the predominant patterns in the time series and their spatial distribution within the actual map area.

In this Thesis, we propose two geolocated time series summarization approaches for visual exploration, named *bundle* and *tile map summary*. These are supported and driven by two *hybrid* indices that speed up the result computation, providing efficient exploration of geolocated time series data. They consist of a spatial and a time series summary that jointly facilitate knowledge extraction and insight gaining. To the best of our knowledge, this is the first work that considers visual exploration and summarization of geolocated time series. In brief, our main contributions on this field are the following:

- We suggest an adapted variant of the BTSR-tree index, as well as a novel algorithm for its traversal in order to quickly retrieve summaries (a.k.a. bundles) of geolocated time series within a given spatial area.

- We propose a hybrid variant of the iSAX index, called geo-iSAX, which combines time series with spatial information within its nodes. Based on that, we describe a novel traversal algorithm for geo-iSAX that enables fast timebox search by performing efficient pruning, while avoiding false negatives.

- We exemplify the proposed visualization methods with two use cases based on real-world datasets. In addition, we empirically evaluate the performance of our summarization methods, confirming their low execution time against a large synthetic dataset of geolocated time series.

A preliminary version of the above results appear at the BigVis workshop, which was held in conjunction with EDBT/ICDT 2018 joint conference [CSP+18]. The complete work is published at the Elsevier Big Data Research journal [CPS+19] in 2019;

Local Similarity Search. Most time series similarity measures, such as *Euclidean distance* and *Dynamic Time Warping* (DTW) are globally calculated, i.e., across the entire length of time series. In this Thesis, we introduce the measure of *local similarity*, that can be applied on co-evolving (i.e., time aligned) time series. We consider the problem of detecting locally similar pairs and groups, called bundles. These are pairs or groups of subsequences whose values do not differ by more than ϵ for at least δ consecutive timestamps, thus indicating common local patterns and trends. We first present a baseline algorithm that performs a sweep line scan across all timestamps to identify matches. Then, we propose a filter-verification technique that only examines candidate matches at judiciously chosen checkpoints across time. Specifically, we introduce two block scanning algorithms for discovering local pairs and bundles respectively, which leverage the potential of checkpoints to aggressively prune the search space.

In the same line of work, we tackle the problem of similarity search on geolocated time series data, using local similarity. Our approach for hybrid search over geolocated time series using the BTSR-tree supports only *global* geolocated time series similarity. It turns out that such hybrid queries involving local similarity can still be evaluated using the BTSR-tree index. We first present a baseline method employing a sweep-line algorithm to check for local similarity, and then describe how this can be optimized by using appropriately placed checkpoints, based on the local similarity score threshold specified by the query, in order to skip unnecessary comparisons. Despite the fact that this saves some computations, the resulting time savings are relatively small, since the number of index nodes that need to be probed is not essentially reduced. To overcome this problem, we introduce an improvement to the BTSR-tree index, which is based on temporally segmenting the time series bounds within each node and deriving tighter bounds per segment. Once the time series bounds in each node become more fine-grained, pruning the search space for local similarity queries proves much more effective.

Our main contributions on pair/bundle discovery and local similarity search can be summarized as follows:

- We introduce the problems of local pair and bundle discovery over co-evolving time series.
- We suggest an aggressive checkpoint-based pruning method that drastically reduces the candidate pairs and bundles that need to be verified, significantly improving performance.
- We conduct an extensive experimental evaluation using both real-world and synthetic time series, showing that our algorithms outperform the respective sweep line baselines.
- We extend our previous work on hybrid queries for geolocated time series to support local time series similarity. We consider both range and top-k queries, including combined criteria for spatial distance and local time series distance.
- We present how such queries can be answered efficiently exploiting the previously introduced BTSR-tree index.
- To achieve greater savings in execution time by further reducing node accesses, we propose an enhanced variant of BTSR-tree, called $\mathcal{S}$BTSR-tree, which additionally employs temporal segmentation in each node to derive tighter, more fine-grained time series bounds.
- We experimentally evaluate our methods using real-world datasets from different application domains, showing that BTSR-tree can efficiently handle hybrid queries under local similarity search, while $\mathcal{S}$BTSR-tree achieves even higher performance due to the additional temporal segmentation.

The above results are published at the 16th International Symposium on Spatial and Temporal Databases (SSTD) 2019 [CSP+19a] and at the International Conference on Advances in Geographic Information Systems (SIGSPATIAL) 2019 [CSP+19b].

Scalable Time Series Forecasting. Forecasting time series is crucially useful in various applications, such as resource demand management (e.g., water, electricity, natural gas), stock market and supply demand forecasting (e.g., in super markets). Depending on the dataset, it can be a rather complex and computationally intensive task due to the high dimensionality and usual uncertainty in such data. In this Thesis, we introduce a framework for scalable data analysis on Big Data collections, named *Flink Machine Learning k-Nearest Neighbors* (FML-kNN for short). The framework implements a probabilistic classifier and a regressor. Its core algorithm is an extension of F-zkNN [CKAS15], which is built upon an optimized version of the H-zkNNJ [ZLJ12] distributed approximate kNN join algorithm. We showcase the framework's capabilities through a large scale time series forecasting use case on a large real-world dataset of hourly water consumption per household within a city.

The overall contributions of our work on FML-kNN are the following:

- We propose a novel, easily extensible distributed processing framework that performs probabilistic classification and regression using kNN join.

- The framework operates in a single distributed session, saving I/O and communication resources. Similar approaches require three distributed sessions.
- We present a detailed experimental and comparative evaluation with similar approaches and exhibit our framework's efficiency in terms of scalability and wall-clock completion time.
- We conduct experiments on two real-world cases using water consumption related datasets and extract useful knowledge and insights towards the induction of more efficient water use.

A preliminary version of these results appear at the 2015 IEEE International Conference on Big Data (BigData) [CKAS15]. The complete work is published at the Springer Journal of Big Data [CKAS18] in 2018;

1.3 Organization

The rest of this Thesis is structured as follows:

Chapter 2 surveys the related work in the field of time series indexing, querying, exploration and similarity search, as well as scalable kNN methods and machine learning.

Chapter 3 presents our hybrid BTSR-tree index, that indexes geolocated time series both in the spatial and in the time series domain.

Chapter 4 extends the work of Chapter 3, introducing a variety of hybrid queries that can be processed using the BTSR-tree index.

Chapter 5 considers an efficient exploration on large geolocated time series data, using hybrid indexing.

Chapter 6 introduces the pair and bundle discovery on large co-evolving time series datasets based on local similarity. It also presents a new collection of hybrid local similarity search queries, using hybrid indexing.

Chapter 7 considers scalable kNN joins and presents our FML-kNN framework for classification and regression on Big Data.

Chapter 8 concludes this Thesis and presents future work directions.

Chapter 2

State of the Art

Time series management, mining and exploration has many applications across several industrial and scientific domains. Due to the complex nature of time series data, these tasks have attracted a lot of academic and research interest. As a result, there is a plethora of methods in the literature, that (i) apply various techniques to improve the management of large time series datasets, (ii) facilitate querying and extracting insightful information and (iii) ultimately enable the efficient exploration of such data.

Furthermore, as the ubiquitous generation of information governs more and more aspects of human life, applying efficient analysis tasks on Big Data is a demanding task of increasing scientific and industrial importance. The simplicity along with the effectiveness of the k-Nearest Neighbors (kNN) algorithm, have motivated many research communities over the years with numerous applications and scientific approaches, which exploit or improve its potential on various data types, one of which being time series.

In the following sections, we outline related and state-of-the-art work on the above fields. More specifically, in Section 2.1, we discuss existing approaches for indexing, querying and visual exploration of time series and other types of data. In Section 2.2, we outline related work on similarity search applied on time series data. Finally, Section 2.3, presents existing scalable approaches on k-Nearest Neighbors and dimensionality reduction.

2.1 Time Series Indexing, Querying and Visual Exploration

Spatio-Textual Indices. There is an increasing amount of spatio-textual objects, e.g., Points of Interest (PoI) with textual descriptions, geotagged tweets or posts in social media, etc. This has motivated research on hybrid spatial-keyword queries combining location-based predicates with keyword search. Main query types include the *Boolean Range Query*, which retrieves all objects that contain a given set of keywords and are located within a specified spatial range; the *Boolean kNN Query*, which returns the k nearest objects to a specific location and contain the given keywords; and the *Top-k kNN Query*, which finds the top-k objects according to an objective function that assigns hybrid scores to objects based on both their keyword similarity and spatial proximity to the query object [CCJW13a].

To evaluate such queries efficiently, the main idea is to construct hybrid index structures that simultaneously partition the data in both dimensions, spatial and textual. Essentially, this implies combining a spatial index structure (e.g., R-tree, Quadtree, Space-Filling Curve)

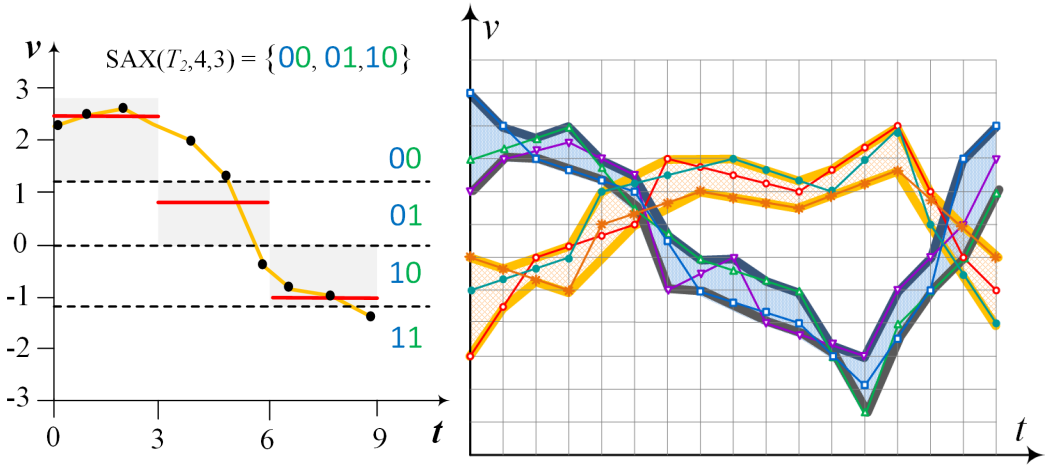
with a textual index (e.g., inverted file, signature file). Depending on their form, the resulting variants can be characterized either as *spatial-first* or *textual-first* indices [CHD+11]. One of the most fundamental and characteristic ones is the IR-tree [CJW09, LLZ+11], which extends the R-tree by augmenting the contents of each node with a pointer to an inverted file indexing terms and documents contained in its sub-tree. Several other hybrid spatio-textual indices extending the R-tree (or R*-tree) have been proposed, such as the IR^2-tree [FHR08], the KR*-Tree [HHLM07], SKI [CWR10] and S2I [RGJN11], while methods based on space filling curves include SF2I [CSM06] and SFC-QUAD [CHD+11].

In this thesis, we introduce a new hybrid indexing method, the BTSR-tree index, which is based on the R-tree [Gut84] for the spatial indexing part. Recall that an R-tree organizes a hierarchy of nested d-dimensional rectangles. Each node corresponds to a disk page and represents the MBR of its children or, for leaf nodes, the MBR of its contained geometries. The number of entries per node (excluding the root) is between a lower bound m and a maximum capacity M. Query execution in R-trees starts from the root. MBRs in any visited node are tested for intersection against a search region. Qualifying entries are recursively visited until the leaf level or until no further overlaps are found. Several paths may be probed, because multiple sibling entries could overlap with the search region.

All aforementioned approaches focus exclusively on combining spatial queries with keyword search. To the best of our knowledge, BTSR-tree is the first one to address geolocated time series, combining spatial queries with similarity search for time series.

Time Series Indexing. Earlier approaches towards indexing time series data were based on leveraging multi-resolution representations. For instance, the Discrete Wavelet Transform [Gra95] is used in [CF99] to gradually reduce the dimensionality of time series data via the *Haar wavelet* [Haa10] and generate an index using the coefficients of the transformed sequences. In [PM02], it is further observed that, other than orthonormal wavelets, bi-orthonormal ones can also be used for efficient similarity search over wavelet-indexed time series data, demonstrating several such wavelets that outperform the Haar wavelet in terms of precision and performance. In addition, an alternative approach to the k-nearest neighbor search over time series data is introduced in [KK11]. The proposed method accesses the coefficients of Haar-wavelet-transformed time series through a sequential scan over step-wise increasing resolutions.

State-of-the-art approaches for time series indexing comprise methods based on the *Symbolic Aggregate Approximation* (SAX) representation [LKWL07]. It is a multi-resolution representation of a time series introduced in [SK08]. It can be derived from its *Piecewise Aggregate Approximation* (PAA) [KCPM01, YF00] by quantizing the PAA segments on the v-axis. As exemplified in Figure 2.1a, a time series T_2 is transformed to a PAA representation of w=3 words with real-valued coefficients (the horizontal red bars). To get a SAX representation for a time series, these coefficients are discretized along the v-axis using *breakpoints* (shown with dashed lines) assuming a $\mathcal{N}(0,1)$ Gaussian distribution that enables generation of equi-probable symbols for a given cardinality ($b = 4$ symbols are used in this example). Interestingly, by using bitwise representations for these symbols, coarser SAX values can be obtained from more refined ones by simply ignoring trailing bits. Importantly, the Euclidean distance between SAX representations of two time series is guaranteed to be a *lower bound* with respect to the Euclidean distance over the original time series. Formally, for two time series T, T' of equal length n using their respective SAX words T_w, T'_w of size w, it holds



(a) SAX of a time series **(b)** MBTS for two sets of time series

Figure 2.1: SAX and MBTS representations over time series.

that:

$$dist_{SAX}(T_w, T'_w) = \sqrt{\frac{n}{w}}\sqrt{\sum_{j=1}^{w} d^2(t_j, t'_j)} \leq \sqrt{\sum_{i=1}^{n}(T.v_i - T'.v_i)^2} \tag{2.1}$$

where $d(t_j, t'_j)$ is the distance between symbols at the j-th position of each SAX word. Comparing iSAX words of different cardinality is possible by promoting the iSAX representation of lower cardinality to that of the larger, as the lower bound in Eq. 2.1 still holds.

The first attempt to leverage the potential of the SAX representation was presented in [SK08], introducing the indexable Symbolic Aggregate Approximation (iSAX), capable of a multi-resolution representation for time series. Considering a set of time series, an iSAX index [SK08] can be built as illustrated in Figure 2.2. The root node captures the complete iSAX space. It does not contain any SAX words, it only points to its children nodes (in the worst case, their number is $2w$). Each leaf has a pointer to a disk file containing the raw time series that it represents. The leaf itself also stores the iSAX word of highest cardinality among these time series. An internal node designates a split in SAX space and is created when the number of time series contained by a leaf node exceeds a fixed capacity M. This split is binary and is made at a given position $j = 1..w$ of the SAX word using a round-robin policy, so it always yields two children that differ on their j-th symbol while replicating the rest from their parent node. In essence, the SAX space represented by every node fully contains the union of the SAX spaces of its subtree.

iSAX can answer *similarity queries*, and thus can be also used in k-nearest neighbor search [SK08]. Searching for time series similar to a given query time series simply traverses the iSAX tree, looking for a leaf node having the same iSAX word as the query. The respective raw times series are fetched from disk and a sequential scan identifies those matching with q. The iSAX index was further extended to iSAX 2.0 in [CPSK10] by enabling bulk loading of time series data. Its next version is the iSAX2+ index [CSP+14], which handles better the expensive I/O operations caused by the aggressive node splitting while building

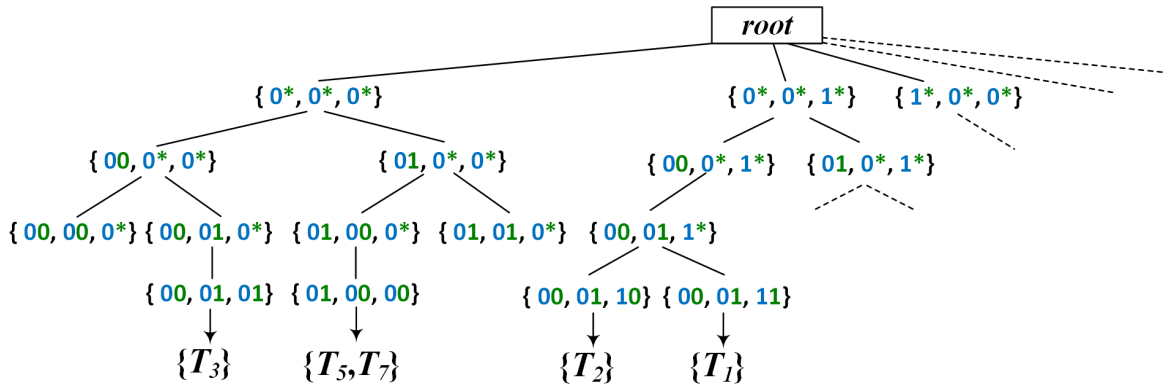


Figure 2.2: iSAX index over time series.

the index. Finally, the ADS+ index [ZIP14] is another extension of iSAX, which attempts to overcome the still significantly expensive index build time by adaptively building the index while processing the workload of queries issued by the user. A comprehensive overview and comparison of the time series indexing approaches based on the SAX representation is presented in [Pal16].

Contrary to BTSR-tree, none of the above approaches supports geolocated time series, and thus cannot efficiently process hybrid queries combining time series similarity with spatial proximity.

Spatial Join Queries With respect to *spatial join queries*, several methods have been proposed, often based on the R-tree family of indices [Gut84, BKSS90]. In particular, the spatial join algorithms over R*-trees introduced in [BKS93] can minimize the CPU and I/O cost in searching. *Multiway* spatial joins [PMT99] generalize search over more than two R-trees. Top-k spatial distance joins [QBM13] employ R-tree-based spatial joins in data blocks ordered by an objective score to retrieve k pairs of objects with highest score. However, all such algorithms are applied against spatial information only. Based on a similar observation for answering a variety of queries over geolocated time series, in [CSP+17] we proposed the BTSR-tree, a hybrid index based on the R-tree, but having nodes that also store bounds over the time series information in their underlying subtree. This index offers increased pruning capabilities for queries involving both time series similarity and spatial proximity. However, handling hybrid similarity joins is not addressed in [CSP+17]; we develop such a method next in this thesis.

Our work on geolocated time series similarity join queries using the BTSR-tree index is reminiscent of related approaches in *spatio-textual search*. Spatio-textual joins identify objects that are both spatially and textually close. In particular, the algorithm proposed in [BGM12] uses a spatial partitioning in conjunction with spatial joins over R-trees in order to batch process such queries. MapReduce-based methods in [ZMM14] resolve spatio-textual joins on spatially partitioned data. However, it should be stressed that time series information is quite distinct from documents or keywords used in those works and certainly requires a totally different processing paradigm. To the best of our knowledge, ours is the first approach for processing similarity joins on geolocated time series data.

Visual Exploration of Time Series. Numerous approaches attempt to leverage the potential of summarizing or aggregating the information of large time series data to facilitate visual exploration and knowledge extraction. An early approach is [MFSW97], where the authors use tile maps and box plots to discover ten-year trends in air pollution data. In [KAK95], the authors introduce a pixel-oriented visualization to detect recursive patterns, where each data value is represented by one pixel. The authors demonstrate the potential of their method using a stock market dataset. An extension of this work is presented in [LAB+09], where several time granularities are combined in a single visualization to enhance the knowledge extraction potential of recursive patterns.

Of particular interest are visualization approaches that attempt to leverage the potential of declarative SQL-like languages and DBMSs to enable exploratory queries. Such an approach is suggested in M4 [JJHM14], where the authors introduce an aggregation-based dimensionality reduction scheme for visualizing horizontally large time series using line charts. Their approach operates on top of an RDBMS and supports various SQL queries that select and visualize particular parts of time series. ForeCache [BCS16] leverages two prefetching mechanisms to facilitate exploration of large geospatial, multidimensional and time series data stored in a DBMS. By predicting the user's behavior, it fetches the necessary data as the user interacts with the application. Another declarative language-based visualization is suggested in [WBM14], where relational algebra queries are used to represent the visualization, leveraging the potential of traditional and visualization-specific optimizations. In contrast, a recent tutorial [MLVP17] advocates the use of example-based methods in exploration of large relational, textual, and graph datasets. Such a *query-by-example* approach has been applied in [EF13] so as to explore relevance feedback for retrieval from time series databases. Instead of returning the top matching time series, this technique incorporates diversity into the results, which are presented to the user for feedback and refined in several rounds.

RINSE [ZIP15] is a Recursive Interactive Series Explorer specifically designed for exploration of data series. Built on top of ADS+ [ZIP14], a special adaptive index structure for data series, it can progressively build parts of the index on demand at query time, concerning only those chunks of the data involved in users' queries. In terms of visualization, users can get those series qualifying to range or nearest-neighbor queries interactively drawn on screen, as well as monitor various statistics regarding the index footprint (e.g., RAM and disk usage) as it gets updated. In contrast, ATLAS [CXGH08] is a visual analytics tool specifically geared towards interactivity when ad hoc filters, arbitrary aggregations, and trend exploration are applied against massive time series data. This client-server architecture employs a column store as its backend equipped with indexing, and preemptively caches data that may be required in queries so as to reduce latency when *panning*, *scrolling*, and *zooming* over time series. Recently, the ONEX paradigm [NARS16] concerns online exploration of time series. It first constructs compact similarity groups over time series for specific lengths based on Euclidean distance, and then can efficiently support exploration of these groups with the Dynamic Time-Warping (DTW) method over their representatives of different lengths and alignments. *Smoothing* can be applied to streaming time series to remove noise in visualizations while preserving large-scale deviations [RB17]. To highlight important phenomena without harming representation quality from oversmoothing, this approach introduces quantitative metrics involving variance of first differences and kurtosis to automatically calibrate smoothing parameters.

The ability to zoom in to specific parts of interest of a large time series can significantly

enhance the exploratory potential of a visualization. Stack zooming [JE10] provides such a functionality, by building hierarchies of line chart visualizations for user-defined intervals on large time series data. Each selected interval is zoomed and stacked beneath the initial time series. A similar approach is KronoMiner [ZCB11], which employs a radial-based visualization to enable zooming functionality for specific time intervals. The interface is visually refined through an iterative design procedure involving expert user feedback. ChronoLenses [ZCPB11] introduces a domain-independent visualization that offers the ability to perform on-the-fly transformations (e.g., Fourier transform, auto-correlation) of the selected interval using *lenses*.

Zooming in regions of interest in time series can be performed via *timeboxes*, which essentially consist of rectangular regions on the time series domain thus specifying intervals in both the time and value axis. The procedure retrieves the time series whose values in the given region are fully contained in the rectangle. Hochheister et al. introduced timeboxes [HS04] along with *TimeSearcher*, an application for visual exploration of time series datasets that implements timebox queries. The user is able to draw rectangles on the time series domain and the results are separately displayed on-screen. Keogh et al. [KHS02] extended the timeboxes, introducing the *Variable Time Timeboxes*, which allowed a degree of uncertainty in the time axis. Later versions of TimeSearcher (such as [ASP+05]) provided enhanced functionality, allowing the visual exploration of longer time series ($>$10,000 time points) and offering forecasting functionality.

None of the aforementioned methods and systems provides map-based visual exploration of *geolocated* time series. In this thesis, we introduce a summary construction method for geolocated time series, that utilizes our spatial-first BTSR-tree index, to enable spatial-domain map-based exploratory visualizations. Further, we introduce a *time series-first* hybrid index to facilitate timebox search on both horizontally and vertically large time series datasets. We enable efficient exploration on geolocated time series datasets, by timely executing user-defined timebox search, enabling the exploration also in the time series domain.

2.2 Time Series Similarity Search

Correlated Time Series. Identifying similar subsequences between time series also indicates some *correlation* between them. Several approaches compute pairwise statistics (e.g., Pearson correlation, beta values) especially in streaming time series [ZS02, CSZ05, PSP06]. There are also works concerning *co-evolving* time series data, either towards detecting and correcting missing values [CTFJ15] or mining typical patterns and points of variation to achieve a meaningful segmentation of large time series [MSF14]. However, none of these approaches is applicable to our setting, where we require similarity in the time series values.

Time series clustering. Our work also relates to *clustering of time series*, where methods perform either partitioning or density-based clustering. In the former class, algorithms typically partition the time series into k clusters. Similarly to iterative refinement employed in k-means, the k-Shape partitioning algorithm [PG15, PG17] aims to preserve the shapes of time series assigned to each cluster by considering the shape-based distance, a normalized version of the cross-correlation measure between time series. In contrast, density-based clustering methods are able to identify clusters of time series with arbitrary shapes. YAD-

ING [DWD+15] is a highly efficient and accurate such algorithm, which consists of three steps: it first samples the input time series also employing PAA (Piecewise Aggregate Approximation) to reduce the dimensionality, then applies multi-density clustering over the samples, and finally assigns the rest of the input to the identified clusters. However, clustering methods consider time series in their entirety and not matching subsequences as we consider in this thesis.

Subsequence matching. The problem of *subsequence matching* over time series is to identify matches of a (relatively short) query subsequence across one or more (relatively long) time series. The UCR suite [RCM+12] offers a framework comprising various optimizations regarding subsequence similarity search. Matrix Profile [YZU+16] includes methods for detecting, for each subsequence of a time series, its *nearest neighbor* subsequence, by keeping track of Euclidean distances among candidate pairs. Applying such approaches in our setting is not straightforward. First, they involve Euclidean or DTW distances, which are different from our definition of local similarity score (see Section 6.2.1, hence the pruning heuristics do not hold in our case. Second, they do not consider geolocated time series, thus spatial filtering has to be carried out independently, which reduces pruning opportunities.

Similarity Search. Similarity search over time series has attracted a lot of research interest [EZPB18]. One well-studied family of approaches includes wavelet-based methods [CF99], which rely on *Discrete Wavelet Transform* [Gra95] to reduce the dimensionality of time series and generate an index using the coefficients of the transformed sequences. The *Symbolic Aggregate Approximation* (SAX) representation [LKWL07] has led to the design of a series of indices, including iSAX [SK08], iSAX 2.0 [CPSK10], iSAX2+ [CSP+14], ADS+ [ZIP14], Coconut [KDZP18], DPiSAX [YAMP18], and ParIS [PFP18]. However, these indices support similarity search over complete time series, i.e. whole-matching. Recently, the *ULISSE* index was proposed [LP18], which is the first index that can answer similarity search queries of variable length.

Moreover, many approaches have been proposed for subsequence matching. In this problem, a query subsequence is provided and the goal is to identify matches of this subsequence across one or more time series, typically of large length. The *UCR suite* [RCM+12] offers a framework comprising four different optimizations regarding subsequence similarity search. In computing *full-similarity-joins* over large collections of time series, i.e., to detect for each possible subsequence its *nearest neighbor*, the *matrix profile* [YZU+16] keeps track of Euclidean distances among each pair within a *similarity join set* (i.e., a set containing pairs of each subsequence with its nearest neighbor).

The problem of pair and bundle discovery that we address in this thesis differs from the above settings. Instead of identifying matches of a query subsequence against one, or more time series, we are interested in discovering locally similar pairs and bundles of time-aligned subsequences within a given collection of time series.

Discovery of Movement Patterns in Trajectories. Finally, our work on pair and bundle discovery also relates to approaches for discovering clusters of moving objects, in particular a type of movement patterns that is referred to as *flocks* [GvK06]. A flock is a group of at least m objects moving together within a circular disk of diameter ϵ for at least δ consecutive timestamps. Finding an exact flock is NP-hard, hence this work suggests an *approximate* solution

to find the *maximal* flock from a set of trajectories using computational geometry concepts. In [BGHW08], another *approximate* solution for detecting all flocks is based on a skip-quadtree that indexes sub-trajectories. Flock discovery over *streaming* positions from moving objects was addressed in [VBT09]. This *exact* solution discovers flock disks that cover a set of points at each timestamp. Their flock discovery algorithm finds candidate flocks per timestamp and joins them with the candidate ones from the previous timestamps, reporting a flock as a result when it exceeds the time constraint δ. An improvement over this technique was presented in [TVK15], using a *plane sweeping* technique to accelerate detection of object candidates per flock at each timestamp, while an inverted index speeds up comparisons between candidate disks across time. In our setting, detection of bundles is similar to flocks, thus for our baseline method we adapt the algorithm from [VBT09].

2.3 Scalable k-Nearest Neighbors and Machine Learning

k-Nearest Neighbors. Several works in the literature have reported the superiority of k-nearest neighbors on machine learning tasks over similar approaches, both in terms of processing time, as well as in terms of accuracy [ACC+14, Yan99, C+09]. There are numerous approaches that exploit its potential. Zhang et al. [ZZ05] introduced a multi-label classification method based on kNN which uses a *Maximum a Posteriori* (MAP) principle to predict the class of an new element. Oswald et al. [OSS01] used an approximate kNN-based regression approach in order to forecast traffic flow. Wei and Keogh [WK06] presented that an 1NN classifier outperforms other similar methods in terms of error rate, when applied on time-series data. Xu [Xu11] introduced a multi-label weighted kNN classifier, where the weights for each class are computed via mathematical optimization, using Least Squared Errors (LSE). Gou et al. [GXK11] presented a weighted voting scheme for such classifiers, where the distance between an element and its nearest neighbors determines the weight of each neighbor's vote. The query element is classified to the most weighted class. Our FML-kNN framework extends this functionality by also calculating the probability of each element belonging to each class. Also, by employing a k-nearest neighbors joins approach, we allow for simultaneous classification or regression on datasets of really high volume, addressing the challenges that arise when processing Big Data in a distributed environment.

Dimensionality Reduction and Space Filling Curves. Similarly to many data analysis and management tasks, kNN joins suffer from the *curse of dimensionality* [BKD98]. Liao et al. [LLL01] stated that as the number of dimensions increases, such techniques need an exponentially larger amount of CPU time. Consequently, executing a kNN joins method on Big Data requires prohibitively long-lasting operations. To overcome this issue, *dimensionality reduction* can be applied on the datasets by indexing their elements via a *Space Filling Curve* (SFC) [Sag12]. This approach reduces data dimensionality to one dimension, which allows for a significantly faster execution of an approximate nearest neighbor search, based on the indexed elements. The most widely used SFCs are the *z-order*, *Gray-code* and *Hilbert*, among which Mokbel et al. [MAK02] concluded that Hilbert is the most "fair", due to the fact that two consecutive points in the curve are always nearest neighbors. Yao et al. [YLK10] used the z-order curve to significantly boost the query performance over huge amounts of data. Lawder et al. [LK01] efficiently executed range queries in data indexed by the Hilbert curve, while

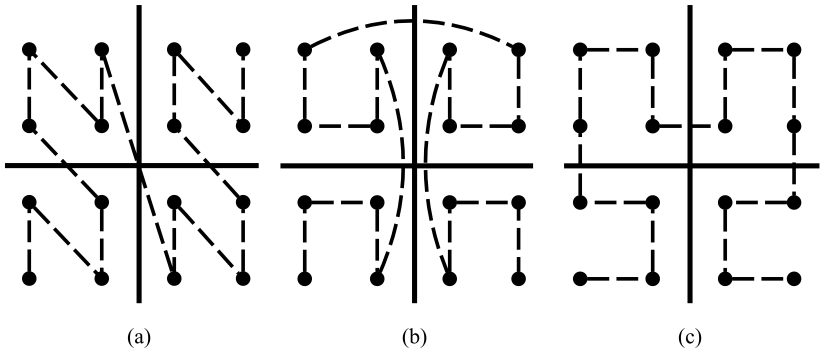


Figure 2.3: The three Space Filling Curves: (a) z-order curve, (b) Gray-code curve (c) Hilbert curve.

Faloutsos [Fal86] presented a mathematical model for indexing the multi-attribute records of a data collection, using Gray-codes instead of binary values. Similarly, Chatzigeorgakidis et al. [CKAS15] and Zhang et al. [ZLJ12] exploit the z-order curve in order to perform kNN joins on a distributed environment. To support selection variety, our framework can operate by tuning and using the most preferable among these SFC methods, as space traversal quality and computation performance may differ according to the data context. Figure 2.3 shows an example of the recursive way the three SFCs scan the elements in a two-dimensional space.

k-Nearest Neighbors on Distributed Environments The increasing scientific interest in the Big Data area has introduced modern distributed implementations of famous algorithms. More particularly, several approaches of MapReduce-based kNN joins algorithms have been proposed. Song et al. [SRHM15] present a review of the most efficient among them, denoting that all share the same three processing stages, i.e., (i) *data pre-processing*, (ii) *data partitioning and organization*, and (iii) *kNN computation* stage. They conclude that the SFC-based H-zkNNJ [ZLJ12] algorithm, outperforms in terms of completion time other similar methods like *RankReduce* [SMS10], which uses *Locality Sensitive Hashing* (LSH) [IM98]. In [CKAS15], we extended the functionality of H-zkNNJ [ZLJ12], by delivering the F-zkNN probabilistic classifier, which performs classification on the results of a kNN joins query. The F-zkNN probabilistic classifier is based on the MapReduce programming model and executed in a single distributed session. However, the datasets need to be propagated between the first two execution stages using local caches, which results in increased communication costs. Our approach optimizes F-zkNN by avoiding this costly propagation using Flink's broadcast sets and augments its functionality by incorporating an additional regression analysis operation. Both the probabilistic classifier and the regressor are included in our framework and are experimentally applied on water consumption related Big Data analysis tasks.

k-Nearest Neighbors in Resources Consumption Domain Similar approaches which apply machine learning methods on resource consumption data (e.g., energy, water) but still not from a Big Data perspective, include the work of Chen et al. [CDW+11] who used a kNN classification method and labeled water data to identify water usage. Naphade et al. [NLKS11] and Silipo and Winters [SW13] focused on identifying water and energy consumption patterns, providing analytics and predicting future consumptions but in high gran-

ularity levels and in small scale. Schwarz et al. [SLS+12] used kNN for classification and short-term prediction in energy consumption by using smart meter data, however, they focused on in-memory and non-distributed approaches, thus, limiting the applicability on larger datasets. Our approach in the consumption domain partially relates to the work of Kermany et al. [KMB+13], where the kNN algorithm was applied for classification on water consumption data, in order to detect irregular consumer behavior. We take a step forward by applying our algorithm on two real-world case studies, delivering predictive analytics for water consumption in a Big Data scale.

Chapter 3

Hybrid Indexing of Geolocated Time Series

In this chapter, we propose a *hybrid index* for efficiently supporting similarity search on geolocated time series combining both spatial proximity and time series similarity. First, we introduce the *TSR-tree*, an extension of the R-tree spatial index. In the TSR-tree, each node is augmented with additional information corresponding to the bounds of the time series contained in its subtree, in addition to the standard *Minimum Bounding Rectangle* (MBR) denoting the spatial bound of its contents. Maintaining both kinds of bounds in each node allows to prune the search space simultaneously in the spatial dimension and in the time series dimension while traversing the index. Thus, the number of required node accesses is significantly reduced, since we only retrieve the contents of nodes that may actually contain objects satisfying both types of predicates.

Our proposed index, called *BTSR-tree* is an optimized variant of *TSR-tree*, with its nodes having entries with more refined bounds by bundling together similar time series. This allows to compute and maintain tighter bounds for each individual bundle, hence increasing pruning effectiveness. To allow for a larger number of bundles in nodes at higher levels in the tree hierarchy, we exploit *Piecewise Aggregate Approximation* [KCPM01, YF00] to trade off between the number of bundles and the resolution of the bounding time series for each bundle.

Our approach follows a similar rationale to that applied in *hybrid spatio-textual indices* [CCJW13b]. In that line of research, several variants of hybrid indices (e.g., the IR-tree [CJW09]) have been proposed to tackle the problem of combining spatial predicates with keyword search. Constructing a hybrid index that combines spatial and textual pruning has been shown to speedup processing of hybrid spatial-keyword queries. Motivated by this, our goal is to provide similar improvements for queries on geolocated time series data. Nevertheless, spatio-textual indices are designed specifically for keyword search and typically rely on inverted indices for the textual part, hence they are not applicable to queries that involve similarity of time series where the sequence of values is important.

The rest of this chapter is organized as follows. Section 3.1 provides background on distance functions that we use for spatial and time series domains. Sections 3.2 and 3.3 present our proposed indices. Finally, Section 3.4 concludes the chapter.

3.1 Preliminaries

Next, we introduce our notation and define the distance functions for geolocated time series.

In the *spatial domain*, the distance between two geolocated time series T and T' is calculated using the Euclidean distance of their respective locations. Furthermore, we normalize this distance with $maxDist_{sp}$, i.e., the maximum possible spatial distance in the dataset, to obtain a measure in the interval $[0, 1]$. Thus:

$$dist_{sp}(T, T') = \frac{\sqrt{(T.loc_x - T'.loc_x)^2 + (T.loc_y - T'.loc_y)^2}}{maxDist_{sp}} \tag{3.1}$$

In the *time series domain*, similar to many prior works (e.g., [SK08]), we also apply the Euclidean distance to measure the similarity. Specifically, we calculate the distance between two geolocated time series T and T' as follows:

$$dist_{ts}(T, T') = \frac{\sqrt{\sum_{i=1}^{w} (T.v_i - T'.v_i)^2}}{maxDist_{ts}} \tag{3.2}$$

where $maxDist_{ts}$ denotes the maximum possible distance in the time series domain and is used for normalization, as above.

3.2 The TSR-tree Index

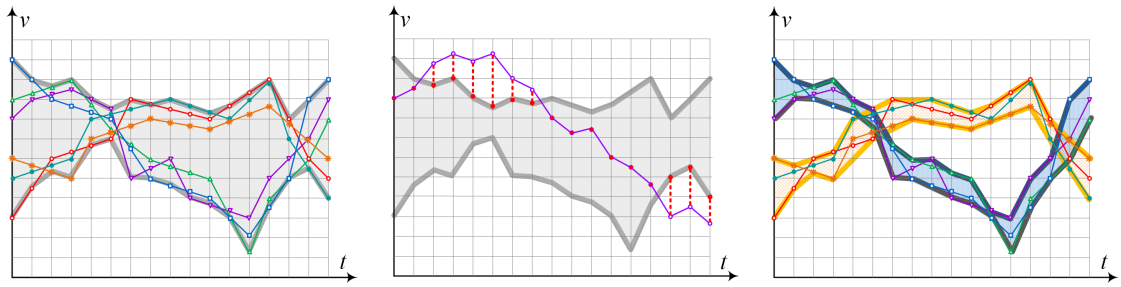
This section presents the architecture of the TSR-tree index and its hybrid pruning capability both in the spatial and in the time series domain.

3.2.1 Index Structure

The TSR-tree is an enhanced R-tree. As in the standard R-tree [Gut84], each node has at least m and at most M entries and stores the MBRs of its children. Additionally, for each child, a node stores a pair of *Minimum Bounding Time Series* (MBTS) that enclose all the time series indexed in its subtree. More specifically, this pair consists of an *upper bounding time series* $B^{\sqcap}$ and a *lower bounding time series* $B^{\sqcup}$, respectively constructed by selecting the maximum (for $T^{\sqcap}$) and minimum (for $T^{\sqcup}$) of values at each time point among all geolocated time series indexed therein. More formally:

Definition 2 (Minimum Bounding Time Series (MBTS))**.** *Given a set of time series $\mathcal{T}$, its MBTS consists of an* upper bounding time series $B^{\sqcap}$ *and a* lower bounding time series $B^{\sqcup}$*, constructed by respectively selecting the maximum and minimum of values at each time point among all time series in set $\mathcal{T}$ as follows:*

$$\begin{aligned} B^{\sqcap} &= \{\max_{T \in \mathcal{T}} T.v_1, \ldots, \max_{T \in \mathcal{T}} T.v_n\} \\ B^{\sqcup} &= \{\min_{T \in \mathcal{T}} T.v_1, \ldots, \min_{T \in \mathcal{T}} T.v_n\}. \end{aligned} \tag{3.3}$$



(a) MBTS in TSR-tree **(b)** Distance bounds with MBTS **(c)** MBTS in BTSR-tree

Figure 3.1: Examples illustrating the time series bounds and pruning in TSR-tree and BTSR-tree.

□

Example 1. *Figure 3.1a illustrates an example of the MBTS of a set of given time series. The latter are represented in the figure by colored solid lines with different markers. The upper and lower bounding time series are represented by thick grey lines, enclosing the whole (shaded) area where the individual time series lie.* □

Construction and maintenance of the TSR-tree follow the standard procedures of the R-tree for data insertion, deletion and node splitting. Raw geolocated time series are always inserted into leaves. The difference is that, after the R-tree has been built, it is traversed in a reverse breadth-first manner and the MBTS of each node are calculated. Moreover, the heuristic for determining where each geolocated time series will be inserted can be adapted to also account for the similarity in the time series domain, in addition to their spatial distance. Recall that in the standard R-tree, selecting the node where a new object will be inserted is based on finding the entry where such an insertion incurs the least possible enlargement of its MBR. Now, this is extended to also consider the enlargement incurred in that entry's MBTS. Specifically, selecting the appropriate node N should minimize the following hybrid cost function:

$$cost(T, N) = \lambda \cdot cost_{sp}(T, N) + (1 - \lambda) \cdot cost_{ts}(T, N) \tag{3.4}$$

where T is the new geolocated time series for insertion, $cost_{sp}$ and $cost_{ts}$ are functions quantifying the cost of enlarging N's MBR and MBTS, respectively, and λ is a weight parameter determining the relative importance of the two factors. Notice that for $\lambda = 1$, this heuristic behaves exactly as in the standard R-tree. When checking a given geolocated time series T for insertion in node N, we calculate distance δ_i between T and current MBTS at each time point $i \in \{1, \ldots, n\}$:

$$\delta_i = \begin{cases} T.v_i - B_N^{\sqcap}.v_i, & \text{if } T.v_i > B_N^{\sqcap}.v_i \\ B_N^{\sqcup}.v_i - T.v_i, & \text{if } T.v_i < B_N^{\sqcup}.v_i \\ 0, & \text{if } TB_N^{\sqcup}.v_i \leq T.v_i \leq B_N^{\sqcap}.v_i. \end{cases} \tag{3.5}$$

Such distances δ_i are shown with dashed red lines in Figure 3.1b. Thus, we can quantify enlargement $cost_{ts} = \sum \delta_i$ of MBTS for N.

3.2.2 Hybrid Node Pruning

For efficient hybrid query execution, the goal is to reduce the number of node accesses by pruning subtrees of the index that cannot contain any results. To do so, we need to establish lower bounds for the different types of distances between the query T_q and any geolocated time series contained in the subtree rooted at a node N in the index. In the spatial domain, this bounding $mindist_{sp}(T_q, N)$ is computed as in the case of the standard R-tree, i.e., based on $N's$ MBR. Similarly, in the time series domain, the corresponding bounding distance is derived by $N's$ MBTS. Following Equations 3.2 and 3.3, it can be easily seen that this can be calculated as follows:

$$mindist_{ts}(T_q, N) = \frac{\sqrt{\sum_{i=1}^{w} \delta_i^2}}{maxDist_{ts}}, \tag{3.6}$$

where δ_i represents distances between T_q and MBTS at each time point i, computed analogously to Equation 3.5.

Example 2. *Figure 3.1b depicts a query time series (the magenta solid line) and the bounds (thick gray lines) in the MBTS of a node. The vertical dashed red lines outside MBTS indicate distances between the query and those bounds contributing to* $mindist_{ts}(T_q, N_i)$.

Moreover, given that the geolocated time series under a node N are a subset of those of its parent N', it follows from the definition of the time series bounds (Equation 3.3) that the MBTS of N is tighter than (or equal to) the MBTS of N'. From Equation 3.6, this guarantees that:

$$mindist_{ts}(T_q, N') \leq mindist_{ts}(T_q, N). \tag{3.7}$$

So, if $mindist_{ts}(T_q, N')$ is higher than threshold θ_{ts} specified by the query, the entire subtree rooted at N' can be pruned altogether.

We define a *hybrid distance* measure $dist_h(T, T')$ between two geolocated time series T and T', combining both their distances $dist_{sp}(T, T')$ in the spatial domain and $dist_{ts}(T, T')$ in the time series domain. For that purpose, we apply an *exponential decay* function to decrease the similarity of two time series based on their spatial distance:

$$sim_h(T, T') = sim_{ts}(T, T')\ e^{-\gamma\ dist_{sp}(T,T')} \tag{3.8}$$

where $sim_{ts}(T, T') = 1 - dist_{ts}(T, T')$ and γ is the exponential decay constant. Then:

$$dist_h(T, T') = 1 - sim_h(T, T') \tag{3.9}$$

Accordingly, the hybrid similarity (distance) between two geolocated time series T and T' is equal to their standard time series similarity (distance) if they are located at the same position, and it gradually decreases (respectively, increases) if one is placed farther apart from the other. From Equations 3.8 and 3.9 we can observe that hybrid distance $dist_h(T_q, T)$ is monotone with respect both to $dist_{sp}(T_q, T)$ and $dist_{ts}(T_q, T)$. This implies that a minimum hybrid distance bound $mindist_h(T_q, N)$ for the contents of a given node N can be similarly

established by combining the individual bounds $mindist_{sp}$ (T_q, N) and $mindist_{ts}(T_q, N)$, i.e.:

$$mindist_h(T_q, N) = 1 - (1 - mindist_{ts}(T_q, N))\ e^{-\gamma\ mindist_{sp}(T_q, N)} \tag{3.10}$$

3.3 The BTSR-tree Index

As explained in the previous section, each node in the TSR-tree holds an MBTS (i.e., a pair of upper and lower bounding time series) that encloses all the time series contained in its subtree. This allows to prune nodes while traversing the index by computing lower bounds for similarity in the time series domain. Clearly, the pruning effectiveness depends on the tightness of these bounds.

The TSR-tree does not provide any explicit guarantee that time series contained in the same node are highly similar to each other, hence the produced bounds are typically expected not to be sufficiently tight. Further, constructing the aggregate bounds from a set of dissimilar time series, yields bounding time series that are not very similar to any of the enclosed ones. These observations are obvious in the example of Figure 3.1a, where the constructed MBTS encloses a much larger (grey shaded) area than the one actually occupied by the contained time series, and the resulting bounding time series do not closely resemble any of the original ones.

To overcome this issue, we present an optimized version of the TSR-tree, which we call BTSR-tree. In a nutshell, the idea is to *bundle* similar time series together in each node, and construct individual MBTS per bundle. This allows to derive bounding time series that are tighter and resemble more closely the enclosed ones.

Example 3. *Figure 3.1c illustrates this idea using the same example as Figure 3.1a. The original time series are now grouped in two bundles, and the MBTS of each bundle is constructed separately. The resulting bounds are now much tighter, eliminating much of the dead space within the MBTS and allowing more precise similarity comparisons.*

The BTSR-tree index is built similarly to the TSR-tree index. To construct the time series bundles within each node, we rely on k*-means clustering*. To avoid confusion with the top-k predicate in queries, next we symbolize with β is the number of bundles to be created. The process is performed bottom-up, starting from the leaf nodes of the index. In each leaf node, the contained time series are clustered into β bundles. Then, the MBTS of each bundle is computed and stored in the node. Each internal node receives all the MBTS of its children so as to compute its own β bundles and respective set of MBTS. Thus, the process propagates upwards, until reaching the root of the tree.

Figure 3.2 depicts an example of a BTSR-tree index for $\beta = 2$. As described above, it is essentially a standard R-tree structure, with each node containing the necessary MBTS of its underlying geolocated time series. Notice that, as we move bottom-up towards the root of the tree, the MBTS of each node tend to be less and less tight and, consequently, overlap. Thus, a higher number β of bundles is needed, since nodes become increasingly more heterogeneous in the time series they contain in their subtree. The higher the number β of bundles, the tighter the resulting bounds. But this also implies that a larger number of MBTS needs to be maintained within each node. However, the number of bundles that can be created is limited by node capacity.

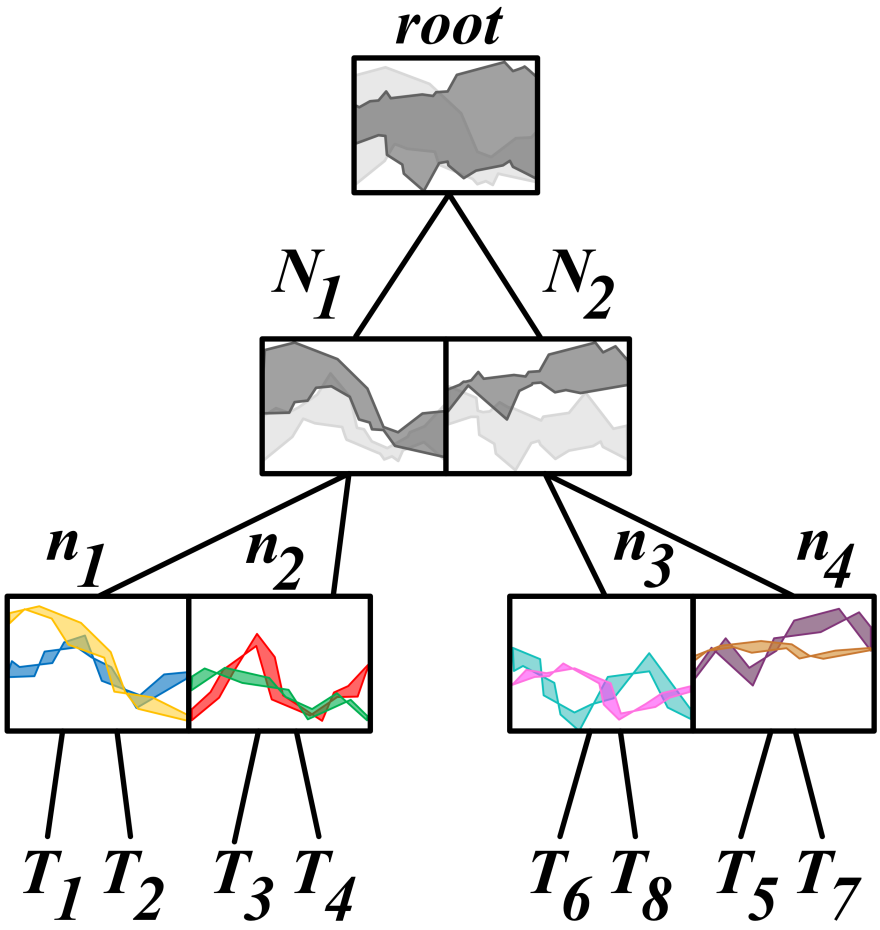


Figure 3.2: An example of a BTSR-tree index.

To address such issues, we increase the number of bundles bottom-up at every tree level by a factor c. Hence, a node at level i has $\beta_i = c \cdot \beta_{i-1}$ bundles; at leaf level, such number β_0 is fixed. At the same time, in order to compensate this increase in terms of node capacity M, we decrease the resolution of the MBTS by the same factor c. The latter is achieved using PAA [KCPM01, YF00]. PAA is a common technique that can approximate a time series $T = \{v_1, \ldots, v_w\}$ of length w into a time series $\bar{T} = \{\bar{v}_1, \ldots, \bar{v}_{w'}\}$ of any arbitrary length $w' \leq w$. In general, each $\bar{v}_i$ is calculated as follows:

$$\bar{v}_i = \frac{w'}{w} \sum_{j=(w/w')(i-1)+1}^{(w/w')i} v_j \tag{3.11}$$

In our case, $w' = w/c$. To preserve bounds when applying PAA on an upper (lower) bounding time series $B^{\sqcap}$ ($B^{\sqcup}$), instead of taking the average as in Equation 3.11, we compute their max (min) values.

The BTSR-tree can execute hybrid queries in a similar manner to the TSR-tree, with a straightforward adaptation. Whenever a node is accessed, instead of evaluating the pruning condition on a single pair of MBTS, each individual MBTS in each bundle is checked, and

the node is pruned if all checks fail.

3.4 Summary

In this chapter, we have addressed the problem of indexing geolocated time series data. Our hybrid TSR-tree index enhances the R-tree by augmenting the spatial bounds in its nodes with respective time series bounds. We have also presented an optimized variant, the BTSR-tree, which maintains tighter time series bounds in each node by bundling similar time series together.

In the following chapter, we will showcase the potential of our hybrid indexing methods for processing various types of hybrid queries on geolocated time series, that combine spatial proximity with time series similarity. These include several similarity search queries, as well as a similarity join query, implemented both for centralized and distributed environments.

Chapter 4

Hybrid Queries on Geolocated Time Series

In this chapter, we focus on efficient evaluation of hybrid similarity search and similarity joins queries on large datasets of geolocated time series.

Hybrid Similarity Search. Consider a geolocated time series dataset that contains the user check-ins of a geosocial network. Each location is associated with the number of visitors across time. Given a query q, one possible query would be to retrieve all the geolocated time series within a given spatial range from that query, that are also similar in the time series domain by a given threshold. Similarly, in a geomarketing or mobile advertisement, one could identify nearby places with similar visiting pattern. Other examples involve time series generated by sensors installed at fixed locations, such as noise, weather or pollution sensors. In the *DAIAD* project[1], smart water meters are installed at households within a city to measure, analyze and mine water consumption patterns towards promoting more intelligent and efficient water use. Finding households in a region or close to a given location that exhibit consumption patterns similar to a given one can offer precious insights.

All such applications need to index geolocated time series to allow efficient similarity search based on both *spatial proximity* and *time series similarity*. Several approaches have been proposed for time series similarity, efficiently indexing large amounts of time series data. One well-studied family of approaches includes wavelet-based methods [CF99], which rely on *Discrete Wavelet Transform* [Gra95] to reduce the dimensionality of time series and generate an index using the coefficients of the transformed sequences. Another line of work employs a *Symbolic Aggregate Approximation* (SAX) representation of time series [LKWL07], introducing a series of indices, such as iSAX [SK08], iSAX 2.0 [CPSK10], iSAX2+ [CSP+14], and ADS+ [ZIP14].

However, to the best of our knowledge, none of the existing works so far has considered the specific case of geolocated time series. All aforementioned indices aim at efficiently supporting similarity search for time series; in case that the analyzed time series are associated with a spatial attribute and issued queries involve spatial filters, these need to be treated independently. Thus, for queries employing both types of predicates, this implies evaluating each

[1] http://daiad.eu/

predicate separately. This can be done by first using a time series index to retrieve similar time series and then applying the spatial predicate on the results, or vice versa, by employing a spatial index to evaluate the spatial predicate and then filter the results according to their similarity with the query time series. In our case, we make use of our BTSR-tree index, leveraging its hybrid indexing potential, allowing for more aggressive pruning in the spatial and time series domains simultaneously.

Hybrid Similarity Join. Consider two such datasets containing time series of CO_2 emissions collected from two sensor networks R and S spread in different locations over a given spatial region. A hybrid similarity join query retrieves pairs of sensors (the first from R, the second from S) such that both the distance between the locations of the two sensors and the distance between the time series of their measurements do not exceed certain given thresholds. Then, an environmentalist may use the matching pairs to identify common patterns in nearby areas and get a better insight about the sources of pollution, its spread, etc. Similarly, check-ins in geosocial networks can also be modeled as geolocated time series and analyzed with hybrid similarity join queries. Results can indicate nearby venues with similar frequency patterns, which may be used for social recommendations according to time, place, activity, etc. Moreover, geolocated time series can indicate water or gas consumption in households. A utility company may identify nearby customers who have similar consumption profiles. Results may be used for customer segmentation, targeted marketing, planning future network upgrades, etc.

Note that similarity join differs substantially from the above hybrid similarity queries over such datasets. A hybrid similarity join query aims to identify *all pairs* between the two datasets qualifying to the criteria of *spatial proximity* and *time series similarity*. Clearly, performing a pairwise comparison among all pairs of objects in the two datasets is not an option when their size is large. Hence, *indexing* them is indispensable for efficient processing of such queries. Certainly, similarity search over indexed time series is a well-studied topic and several schemes have been proposed, as discussed in the previous paragraph. Likewise, efficient methods for distance joins in spatial databases also exist, usually over R-trees [BKS93, PMT99].

Our starting point is to employ such indices either for *time series-only* (with iSAX) or *spatial-only* (using R-trees) filtering of candidate pairs during query evaluation. We also take advantage of the BTSR-tree index, which enables *combined search* over both the time series and the spatial information of candidates and thus excels in pruning power. These algorithms concurrently traverse those indices and identify subtrees that may contain candidate matches. However, this *centralized* approach has certain limitations, as it cannot sustain examination of large datasets. Hence, we further suggest a space-driven data partitioning scheme that enables a *parallel and distributed* approach for hybrid similarity joins. Following the MapReduce paradigm, our method leverages any of the aforementioned indices to efficiently handle similarity join queries locally within each partition. This is then combined with an optimization that minimizes the amount of data transferred between worker nodes at query time without false misses. To the best of our knowledge, this is the first work to address hybrid similarity join queries over large datasets of geolocated time series.

The rest of this chapter is organized as follows. Section 4.1 describes and formulates the proposed hybrid queries. Section 4.2 presents our approach for centralized processing of hybrid similarity search and join queries. Section 4.3, introduces a parallel and distributed

approach for similarity join over large datasets of geolocated time series. Section 4.4 reports our experimental results and, finally, Section 4.5 concludes this chapter.

4.1 Hybrid Query Variants

Next, we outline different hybrid similarity search and join query variants, combining spatial distance with time series similarity.

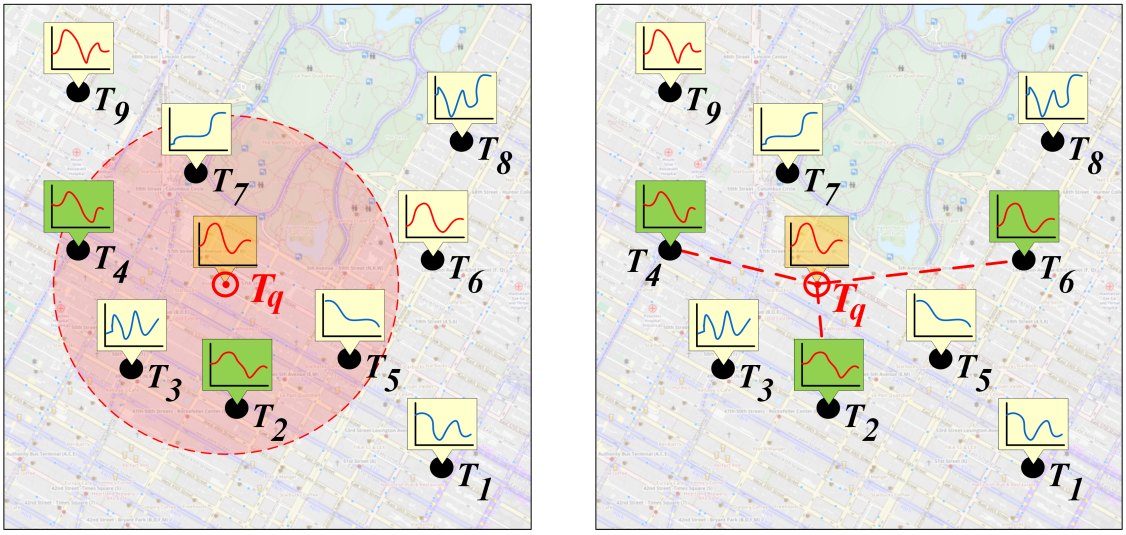
4.1.1 Similarity Search

We are interested in hybrid similarity search queries on geolocated time series, i.e., queries that retrieve search results based on both spatial distance and time series distance. In these queries, a geolocated time series T_q is given as a reference, and the goal is to identify similar time series to T_q, based on both spatial distance and time series distance. Different query variants can be derived. First, the query condition can be applied on each distance independently or on the hybrid distance defined in Section 3.2.2. Second, the condition can be a boolean filter (i.e., retrieve all geolocated time series with distance lower than θ) or a top-k filter (i.e., retrieve the k geolocated time series having the smallest distance). These lead to the query variants listed in Table 4.1 and outlined below:

- $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$. This query applies two individual boolean filters. It retrieves each time series T having $dist_{sp}(T_q, T) \leq \theta_{sp}$ and $dist_{ts}(T_q, T) \leq \theta_{ts}$. In other words, it retrieves all time series that are located within a radius $\theta_{sp} \cdot maxDist_{sp}$ from T_q's location and their time series dissimilarity to T_q is at most $\theta_{ts} \cdot maxDist_{ts}$.
- $Q_{kb}(T_q, k, \theta_{ts})$. This query retrieves the k time series closest to T_q's location also having $dist_{ts}(T_q, T) \leq \theta_{ts}$.
- $Q_{bk}(T_q, \theta_{sp}, k)$. This query retrieves the k most similar time series to T_q which are also located within distance $\theta_{sp} \cdot maxDist_{sp}$ from T_q's location.
- $Q_{hb}(T_q, \theta_h, \gamma)$. This query retrieves all time series having hybrid distance to T_q at most θ_h, i.e., $dist_h(T_q, T) \leq \theta_h$.
- $Q_{hk}(T_q, k, \gamma)$. This query retrieves the k time series with the smallest hybrid distance $dist_h(T_q, T)$ to T_q.

Table 4.1: Similarity search query variants.

Query Variant	Spatial Filter	Time Series Filter
Q_{bb}	boolean (θ_{sp})	boolean (θ_{ts})
Q_{kb}	top-k	boolean (θ_{ts})
Q_{bk}	boolean (θ_{sp})	top-k
Q_{hb}	boolean (θ_h)	
Q_{hk}	top-k	



(a) $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$ **(b)** $Q_{kb}(T_q, k, \theta_{ts})$

Figure 4.1: Hybrid queries on geolocated time series.

Example 4. *Figure 4.1 illustrates an example including two hybrid queries,* $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$ *and* $Q_{kb}(T_q, k, \theta_{ts})$*, on a set of geolocated time series* T_1*, ...,* T_9*. In both cases, the reference time series* T_q *specified by the query is shown in red. Moreover, those time series having dissimilarity to* T_q *at most* θ_{ts} *(i.e.,* T_2*,* T_4*,* T_6*,* T_9*) are also shown with red lines. In the first query, only time series within a radius equal to the spatial distance threshold* θ_{sp} *are retrieved. Thus, the result contains only time series* T_2 *and* T_4*, whereas* T_6 *and* T_9 *are excluded. The second query for* $k = 3$ *retrieves the three closest time series to* T_q *having dissimilarity at most* θ_{ts}*, i.e.,* T_2*,* T_4 *and* T_6*.* □

4.1.2 Similarity Join

Contrary to similarity search, the hybrid similarity join query does not require an initial geolocated time series as reference. Given two datasets R and S (or just one dataset S in the case of self-join), we seek all the existing pairs of geolocated time series that are close to each other both in terms of spatial and time series distance. Of course, both distance thresholds need to be provided before the query is executed. More formally:

Definition 3 (Hybrid Similarity Join over Geolocated Time Series)**.** *Given two sets of geolocated time series* $\mathcal{T}_R$ *and* $\mathcal{T}_S$*, and two thresholds* ϵ_{sp} *and* ϵ_{ts}*, the* hybrid similarity join *query returns all pairs qualifying w.r.t. to both criteria on spatial proximity and time series similarity, i.e.,*

$$\{(T_R, T_S) : T_R \in \mathcal{T}_R, T_S \in \mathcal{T}_S, dist_{sp}(T_R, T_S) \leq \epsilon_{sp} \wedge dist_{ts}(T_R, T_S) \leq \epsilon_{ts}\}.$$

That is, this query searches for pairs of geolocated time series that are within spatial distance at most ϵ_{sp}, while also their respective time series do not deviate by more than ϵ_{ts}. Spatial proximity is measured in distance units (e.g., meters). As mentioned before, since the (transformed) time series are z-normalized, values for parameter ϵ_{ts} are *unitless* and are

typically expressed in standard deviations.

Example 5. *Figure 4.2 depicts two sets of geolocated time series, $\{R_1, \ldots, R_5\}$ (in red bullets) and $\{S_1, \ldots, S_6\}$ (in green squares) that represent CO_2 emissions collected by two sensor networks R and S in an urban area during a day. Suppose that a similarity join query over those two datasets specifies a distance radius $\epsilon_{sp} = 500$ meters to identify nearby sensors and a maximum deviation of $\epsilon_{ts} = 0.4$ to find similar CO_2 patterns. Qualifying pairs $\{(R_1, S_5), (R_3, S_5), (R_5, S_2)\}$ are shown connected with dashed lines. Note that other pairs, e.g., (R_4, S_3), may be even closer in space, but their time series deviate more than the given ϵ_{ts}, so they are filtered out. Besides, time series like those in rejected pair (R_3, S_3) may have almost the same pattern, but their locations are too far from each other to qualify for this query.* □

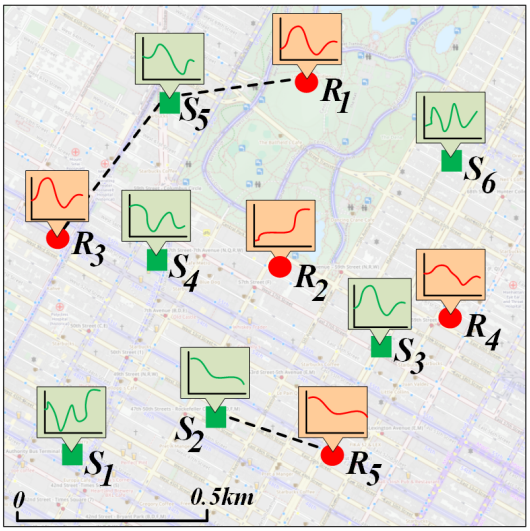


Figure 4.2: Hybrid similarity join over geolocated time series.

4.2 Hybrid Query Processing

This section provides with a detailed description of both similarity search and join query evaluation.

4.2.1 Similarity Search

As already mentioned, for the efficient evaluation of hybrid similarity search queries, we employ our BTSR-tree index. Query processing in BTSR-tree follows a similar procedure to the respective processes for boolean range [Gut84] and kNN queries [HS99] in R-trees. However, this is now extended by utilizing all available bounds $mindist_{sp}$, $mindist_{ts}$ and $mindist_h$, thus pruning nodes simultaneously in the spatial dimension and the time series dimension while traversing the index. This reduces node accesses during evaluation of hybrid queries. Next, we outline the algorithms for executing each of the query variants presented in Section 4.1.

Algorithm 1: $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$

```
1  begin
2      R ← ∅
3      L ← root entries
4      while L ≠ ∅ do
5          N ← L.getNext()
6          if N is not leaf then
7              foreach N' ∈ N.getChildren() do
8                  if mindist_sp(T_q, N') ≤ θ_sp ∧ mindist_ts(T_q, N') ≤ θ_ts then
9                      L ← L ∪ {N'.getChildren()}
10         else
11             foreach T ∈ N.getGeoTS() do
12                 if dist_sp(T_q, T) ≤ θ_sp ∧ dist_ts(T_q, T) ≤ θ_ts then
13                     R ← R ∪ {T}
14             return(R)
```

$Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$**.** To evaluate this query, we recursively traverse the BTSR-tree starting from the root. For each node N, the following two conditions are checked: (a) $mindist_{sp}(T_q, N) \leq \theta_{sp}$ and (b) $mindist_{ts}(T_q, N) \leq \theta_{ts}$. If either of these conditions returns false, N is pruned. Once a leaf node is reached, the geolocated time series contained therein are retrieved, and each one is checked if it qualifies the query criteria. The steps for processing this query are shown in detail in Algorithm 1.

$Q_{kb}(T_q, k, \theta_{ts})$ **and** $Q_{bk}(T_q, \theta_{sp}, k)$**.** These queries combine a boolean filter and a top-k filter. To retrieve top-k results, we employ a *best-first search* approach using a priority queue, as in a typical best-first traversal algorithm [HS99].

Assume that we are evaluating a $Q_{bk}(T_q, \theta_{sp}, k)$ query. Initially, the root entries are retrieved, and their lower spatial distance bound $mindist_{sp}$ is calculated. For a given such entry, if its $mindist_{sp} \geq \theta_{sp}$, then the underlying subtree can be safely pruned. Otherwise, this entry's $mindist_{ts}$ is calculated, and the entry is pushed to the priority queue according to its $mindist_{ts}$ value in ascending order. The process continues recursively, pulling the next item from the queue. If this is a leaf, we retrieve all its raw geolocated time series, calculate their exact spatial distances $dist_{sp}$, and the ones that are not filtered out are pushed to the queue, based on their $mindist_{ts}$. If a pulled item is a geolocated time series, we add it to the result set. The process terminates once k results have been retrieved. Algorithm 2 describes the above procedure in more detail.

Evaluating a $Q_{kb}(T_q, k, \theta_{ts})$ is straightforward; the treatment of the spatial and the time series dimensions is simply reversed.

$Q_{hb}(T_q, \theta_h, \gamma)$ **and** $Q_{hk}(T_q, k, \gamma)$**.** These queries apply a boolean and a top-k filter, respectively. In both cases, this is applied on the hybrid distance $dist_h(T_q, T)$ of the query T_q to each of the candidate geolocated time series T.

Algorithm 2: $Q_{bk}(T_q, \theta_{sp}, k)$

```
1  begin
2      R ← ∅
3      Queue ← root
4      while Queue is not empty do
5          X ← Q.pull()
6          if X is a time series then
7              R ← R ∪ {X}
8              if |R| = k then
9                  break
10         else if X is leaf node then
11             foreach T ∈ X.getGeoTS() do
12                 if dist_sp(T_q, T) ≤ θ_sp then
13                     T.dist ← dist_ts(T_q, T)
14                     Queue.push(T, T.dist)
15         else
16             foreach N ∈ X.getChildren() do
17                 if mindist_sp(T_q, N) ≤ θ_sp then
18                     N.dist ← mindist_ts(T_q, N)
19                     Queue.push(N, N.dist)
20     return(R)
```

For $Q_{hb}(T_q, \theta_h, \gamma)$, query evaluation follows the same process as outlined in Algorithm 1. The difference is that instead of checking separately the given distance thresholds on the spatial and the time series distances, a single check is made comparing the hybrid distances $dist_h$ and $mindist_h$ against the single threshold θ_h.

Evaluation of the $Q_{hk}(T_q, k, \gamma)$ query is similar to the one outlined in Algorithm 2. In this case the difference is that there is no boolean filtering involved, so raw geolocated time series and nodes are inserted into the priority queue according to their hybrid distance, $dist_h$ and $mindist_h$, respectively. Again, the algorithm terminates once k geolocated time series are pulled from the queue; these are the top-k results.

4.2.2 Similarity Join.

A naïve approach to answer a hybrid similarity join query over two geolocated time series $\mathcal{T}_R, \mathcal{T}_S$ would involve examination of all possible pairs, i.e., calculating their Cartesian product $\mathcal{T}_R \times \mathcal{T}_S$ and filtering each candidate pair with the two criteria. Clearly, such a technique has limitations due to its quadratic processing cost and cannot be realistically applied against datasets with more than a few thousand geolocated time series each.

Hence, we propose three *index-based* techniques for answering hybrid similarity join queries:

- We describe a *spatial-only* filtering method that employs R-trees over the locations of geolocated time series so as to identify candidate pairs close enough in space. Afterwards, each such candidate pair should also be checked on their similarity in the time series domain, to finally yield the exact answer (Section 4.2.2.1).
- We build iSAX indices *over the time series* information only per dataset and we introduce a traversal method to facilitate similarity search. Refinement over returned candidate pairs by their spatial distance issues the final results (Section 4.2.2.2).
- We employ BTSR-trees that can *jointly* index the positional and time series information of each geolocated time series. We introduce a *hybrid similarity join* algorithm that descends these two BTSR-trees in tandem and can safely prune subtrees that cannot possibly contribute any valid results (Section 4.2.2.3).

In each of these methods, one global index is created per dataset. Hence, a *centralized* processor is responsible to maintain these indices and access them when evaluating similarity join queries.

4.2.2.1 Spatial-Only Filtering using R-Trees

One possible approach to similarity join search over two datasets $\mathcal{T}_R, \mathcal{T}_S$ of geolocated time series is to build an R-tree [Gut84] per dataset by organizing its spatial locations into a hierarchy of nested d-dimensional rectangles. Each node corresponds to a disk page and represents the MBR of its children. A leaf holds the MBR of its contained geometries. The number of entries per node (excluding the root) is between a lower bound m and a maximum capacity M.

With respect to *hybrid similarity joins*, we search over R-trees using the spatial condition, exactly as in [BKS93]. So, both R-trees are concurrently traversed starting from their roots and recursively examining their respective descendants only if the minimum distance $MINDIST$ of their MBRs [RKV95] does not exceed parameter ϵ_{sp}. Obviously, a pair of nodes breaking this spatial constraint cannot possibly contain any qualifying results, so their respective subtrees can be safely pruned. Once the leaf levels are reached, the candidate pairs of raw time series are accessed and refined according to both criteria in Definition 3 in order to issue the final results.

4.2.2.2 Time Series-Only Filtering with iSAX

This method makes use of available iSAX indices over two datasets $\mathcal{T}_R, \mathcal{T}_S$, each concerning their respective time series as discussed in Section 2.1. Each iSAX index solely indexes the time series part of a dataset; their leaf entries point to the raw time series including their location. Searching for similar geolocated time series starts from the root of each tree and progressively descends by visiting nodes that may contain candidate answers based on their similarity, strictly on the time series domain, as listed in Algorithm 3. Without loss of generality, we assume that either the trees have the same height, or the iSAX for $\mathcal{T}_R$ is less deep than the iSAX for $\mathcal{T}_S$.

Suppose that at a given iteration, node R in the first iSAX needs to be checked against node S in the second iSAX. If neither of them is leaf, the algorithm is recursively called

against all combinations of their children entries, provided that these are within distance ϵ_{ts} as computed by Eq. 3.2 (Lines 1-5). Once the leaf level is reached in the first iSAX but not yet in the second iSAX (if they differ in height), recursive calls examine that specific leaf of the former against each of the children entries of the latter (Lines 6-8).

Eventually, when the leaf level is reached in both trees, we compare each combination of their respective contents (Lines 9-13). Each geolocated time series from leaf R of the first iSAX is checked with its counterparts in leaf S of the second iSAX. Since raw time series data is fully accessible at the leaf level (including locations), refinement of candidates is based not only on their distance $dist_{ts}$ in the time series domain, but also on their spatial distance $dist_{sp}$. If both distances are below the respective constraints, then this specific pair qualifies for the final result Q to the query. The algorithm terminates once there are no remaining pairs of leaves to check.

Algorithm 3: $SimJoinSAX(R, S, \epsilon_{sp}, \epsilon_{ts})$

Input: Nodes R, S, spatial constraint ϵ_{sp}, time series constraint ϵ_{ts}
Output: Set Q of pairs of geolocated time series satisfying constraints

```
1  if R is not leaf ∧ S is not leaf then                ▷ internal nodes in both trees
2      foreach N_R ∈ R.getChildren() do
3          foreach N_S ∈ S.getChildren() do
4              if dist_SAX(N_R, N_S) ≤ ε_ts then              ▷ compare SAX words
5                  SimJoinSAX(N_R, N_S, ε_sp, ε_ts)
6  else if R is leaf ∧ S is not leaf then             ▷ trees of different height
7      foreach N_S ∈ S.getChildren() do
8          SimJoinSAX(R, N_S, ε_sp, ε_ts)
9  else if R is leaf ∧ S is leaf then                 ▷ leaf level in both trees
10     foreach T_R ∈ R.getChildren() do
11         foreach T_S ∈ S.getChildren() do
12             if dist_sp(T_R, T_S) ≤ ε_sp ∧ dist_ts(T_R, T_S) ≤ ε_ts then
13                 Q ← Q ∪ {(T_R, T_S)}                  ▷ add pair to result set
```

4.2.2.3 Hybrid Filtering using BTSR-Trees

This method makes use of a hybrid BTSR-tree index per dataset $\mathcal{T}_R, \mathcal{T}_S$ of geolocated time series. Initially, let us assume that both BTSR-trees have the same height. Exactly like the iSAX-based method, search starts from the root of each tree and descends them in tandem by checking their nodes pairwise, as listed in Algorithm 4.

In case that the currently examined entries R and S are internal (directory) nodes, a nested-loop check finds which of their descendants may contain candidate results (Lines 1-6). In particular, for each child entry N_R of node R, we calculate a buffer rectangle by expanding its respective MBR by distance ϵ_{sp}. In case that this buffer intersects with the MBR of entry N_S from node S, whereas also their MBTS do not deviate by more than ϵ_{ts}, then search

should be recursively applied against those two entries N_R, N_S. Clearly, if neither of these criteria is met, those candidate entries cannot possibly contain any matching time series.

Note that this step involves comparison between two MBTSs. Consider two MBTSs $B_1 = (B_1^{\sqcup}, B_1^{\sqcap})$ and $B_2 = (B_2^{\sqcup}, B_2^{\sqcap})$, each constructed according to Eq. 3.3 over two disjoint subsets of time series data. We compute their *deviation* $dist_{MBTS}$ by first comparing the lower bounding time series of the former with the upper bounding time series of the latter per time point $i \in \{1, \ldots, n\}$, depending on which of these two values is larger, i.e.:

$$\delta_i = \begin{cases} B_1^{\sqcup}.v_i - B_2^{\sqcap}.v_i, & \text{if } B_1^{\sqcup}.v_i \geq B_2^{\sqcap}.v_i \\ B_2^{\sqcup}.v_i - B_1^{\sqcap}.v_i, & \text{if } B_2^{\sqcup}.v_i \geq B_1^{\sqcap}.v_i \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

and we take the average Euclidean norm over these n differences:

$$dist_{MBTS}(B_1, B_2) = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(\delta_i)^2} \tag{4.2}$$

Quite importantly, this measure is a *lower bound* of the Euclidean distance $dist_{ts}(T_1, T_2)$ between two time series T_1, T_2 that are enclosed in MBTSs B_1, B_2, respectively. Consider the situation at a given time point $i \in \{1, \ldots, n\}$. In case that $B_1^{\sqcup}.v_i \geq B_2^{\sqcap}.v_i$, it is straighforward that $T_1.v_i \geq B_1^{\sqcup}.v_i \geq B_2^{\sqcap}.v_i \geq T_2.v_i$ by definition of the MBTS, hence $\delta_i' = T_1.v_i - T_2.v_i \geq \delta_i$. This also holds for the other branches of Eq. 4.1. But, taking the Euclidean norm of these δ_i' values over all time points expresses the time series similarity according to Eq. 3.2. Overall, this confirms that $dist_{ts}(T_1, T_2) \geq dist_{MBTS}(B_1, B_2)$, so checking with MBTSs does not cause any false misses.

If both R and S are leaves, the algorithm retrieves all time series from either leaf and checks every combination against both criteria (Lines 10-14). If a time series from $\mathcal{T}_R$ and a time series from $\mathcal{T}_S$ are close enough in space (i.e., less than ϵ_{sp}) and also similar in the time series domain by ϵ_{ts}, this pair is issued as result.

Handling the case of BTSR-tree indices with different height is handled as in R-trees [BKS93]. Without loss of generality, let the first BTSR-tree (over $\mathcal{T}_R$) be shorter than the second BTSR-tree (over $\mathcal{T}_S$). Then, once a leaf entry R is reached in the former, comparisons are made against any subtrees under nodes N_S in the latter (Lines 7-9). In case that both criteria are met, we descend this latter BTSR-tree and recursively check for similarity joins between its children entries with the same leaf entry R fixed in the first BTSR-tree. Eventually, the leaf level in the second BTSR-tree will be reached and refinement against the raw time series on both the spatial and time series criteria can yield the final results.

4.3 Distributed Similarity Join

As any join query over large datasets, computing hybrid similarity joins over millions of geolocated time series is a very demanding task. Building a global index per dataset and applying any of the methods in Section 4.2.2 still incurs excessive cost, as demonstrated in our empirical tests (Section 5.3.3). To tackle scalability, we present a *parallel and distributed*

Algorithm 4: $SimJoinBTSR(R, S, \epsilon_{sp}, \epsilon_{ts})$

Input: Nodes R, S, spatial constraint ϵ_{sp}, time series constraint ϵ_{ts}
Output: Set Q of pairs of geolocated time series satisfying constraints

```
1  if R is not leaf ∧ S is not leaf then                    ▷ internal nodes in both trees
2      foreach N_R ∈ R.getChildren() do
3          N_R.buf ← buffer(N_R.mbr, ε_sp)                  ▷ expand MBR by ε_sp
4          foreach N_S ∈ S.getChildren do
5              if N_R.buf ∩ N_S.mbr ≠ ∅ ∧ dist_MBTS(N_R, N_S) ≤ ε_ts then
6                  SimJoinBTSR(N_R, N_S, ε_sp, ε_ts)
7  else if R is leaf ∧ S is not leaf then                   ▷ trees of different height
8      foreach N_S ∈ S.getChildren() do
9          SimJoinBTSR(R, N_S, ε_sp, ε_ts)
10 else if R is leaf ∧ S is leaf then                       ▷ leaf level in both trees
11     foreach T_R ∈ R.getChildren() do
12         foreach T_S ∈ S.getChildren() do
13             if dist_sp(T_R, T_S) ≤ ε_sp ∧ dist_ts(T_R, T_S) ≤ ε_ts then
14                 Q ← Q ∪ {(T_R, T_S)}                     ▷ add pair to result set
```

approach based on space-driven partitioning (Section 4.3.1). We also describe an *optimized*, index-guided variant to reduce the amount of data shuffled between workers (Section 4.3.2).

4.3.1 MapReduce Method with Spatial Partitioning

Typically, in MapReduce-based processing, both input datasets $\mathcal{T}_R$, $\mathcal{T}_S$ should be divided into smaller chunks that may be efficiently processed in a distributed fashion by a number of worker nodes. In our case, distributing geolocated time series data by their spatial location is straightforward and can be effectuated much faster as opposed to a times series-based subdivision that may need to examine long sequences. Our method employs a subdivision $\mathcal{P}$ into *disjoint partitions* over the spatial area covering all locations in either dataset $\mathcal{T}_R$, $\mathcal{T}_S$. Partitioning $\mathcal{P}$ is *identical* over both datasets. Without loss of generality, we consider $\mathcal{P}$ as a uniform grid tessellation into $g \times g$ square equi-sized cells, but our method can be easily adjusted to other space-driven subdivisions into disjoint regions (e.g., quadtrees). Choosing a suitable grid granularity g over each axis mostly depends on dataset size, but also on the number and processing power of available nodes in cluster infrastructures.

The pseudocode listed in Algorithm 5 outlines the entire process. It proceeds in two successive phases: (1) a *local search* per partition and (2) *cross-partition search* by shuffling subsets of data between neighboring partitions. In particular, we make use of distinct *tiers* of *blocks* with increasingly finer spatial resolution (Figure 4.3):

1) *Local search per partition* (Lines 1-6): The first tier concerns individual *partitions*, and the algorithm needs to check for similarity join between those geolocated time series from $\mathcal{T}_R$ and those from $\mathcal{T}_S$ contained in the same partition $p \in \mathcal{P}$. This is depicted for

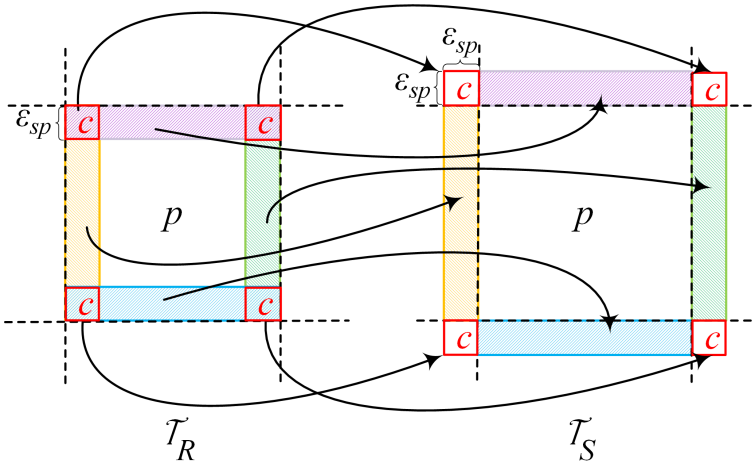


Figure 4.3: Blocks in cross-partition search for a partition p.

a given partition (cell) p enclosed with dashed line segments in Figure 4.3 over each of the two datasets.

2a) *Cross-partition search in pairs of adjacent bands* (Lines 7-12): A collection $\mathcal{B}$ of spatial *bands* of width ϵ_{sp} is created inwards along each side of every partition in $\mathcal{P}$. Geolocated time series of each dataset coming from adjacent bands across every pair of neighboring partitions need to be checked against the query criteria. For a given partition p, each of the four bands created over $\mathcal{T}_R$ must be compared with respective bands created not in the same partition p for $\mathcal{T}_S$, but in each of the four partitions sharing one common side with p. In Figure 4.3, the pairs of respective bands are shown hatched with the same colored pattern and are connected with curly arrows. A set $L_{\mathcal{B}}$ consisting of pairs of such adjacent bands indicates those that must be probed across all partitions.

2b) *Cross-partition search in pairs of boxes with one common corner* (Lines 13-18): The finest tier concerns a set $\mathcal{C}$ of square boxes of side ϵ_{sp} created at the corner of each partition in $\mathcal{P}$. Geolocated time series from either dataset contained in boxes having one common corner need further probing (i.e., *corner-wise* in $\mathcal{P}$), and all pairs of such boxes are collected in set $L_{\mathcal{C}}$. As shown in Figure 4.3, each box c created at the four corners of partition p over $\mathcal{T}_R$ should be checked against one equi-sized box over $\mathcal{T}_S$; this latter box belongs to a neighboring partition $p' \neq p$, which has only one common corner with p.

Since all blocks are purely space-driven, the rationale is that spatial filtering comes first, whilst the time series criterion is checked afterwards for any remaining candidate pairs. At each block level, our method creates disjoint data chunks for subsets of geolocated time series located in that block; this is applied against both datasets similarly for *partitions* (Lines 2-3), *bands* (Lines 8-9), and *boxes* (Lines 14-15).

Furthermore, at each block tier, a *local index* is built for every derived chunk. Interestingly, we may plug in any of the similarity join methods suggested in Section 4.2.2. The same indexing scheme must be used at each tier, i.e., either R-tree, iSAX, or BTSR-tree (hereafter referred to as $\mathcal{X}$-index). For partitions, such indices can be suitably built *in advance* with a

predefined subdivision $\mathcal{P}$ and thus can be readily available for any similarity join query that may specify varying values on parameters ϵ_{sp} and ϵ_{ts}. In contrast, indices over data contained in each of the bands listed in $L_{\mathcal{B}}$ or each corner box in $L_{\mathcal{C}}$ have to be created *at query time*, since they clearly rely on distance threshold ϵ_{sp}, which may vary among queries.

Algorithm 5: $SimJoinMR(\mathcal{T}_R, \mathcal{T}_S, \epsilon_{sp}, \epsilon_{ts}, \mathcal{X})$

Input: dataset $\mathcal{T}_R$, dataset $\mathcal{T}_S$, spatial constraint ϵ_{sp}, time series constraint ϵ_{ts}, index method $\mathcal{X}$ (e.g., BTSR-tree or R-tree)
Output: Set Q of pairs of geolocated time series satisfying constraints
/* **PHASE #1**: *local search per partition* */
1 $\mathcal{P} \leftarrow$ space partitioning common for both datasets $\mathcal{T}_R, \mathcal{T}_S$
2 $R_{\mathcal{P}} \leftarrow$ distribute $\mathcal{T}_R$ and build a local $\mathcal{X}$-index per partition $p \in \mathcal{P}$
3 $S_{\mathcal{P}} \leftarrow$ distribute $\mathcal{T}_S$ and build a local $\mathcal{X}$-index per partition $p \in \mathcal{P}$
4 $L_{\mathcal{P}} \leftarrow \{(p, p) : \forall p \in \mathcal{P}\}$ ▷ `pairs of identical partitions`
5 $Q_{\mathcal{P}} \leftarrow blockwiseSimJoin(L_{\mathcal{P}}, R_{\mathcal{P}}, S_{\mathcal{P}}, \epsilon_{sp}, \epsilon_{ts})$
6 $storeHDFS(Q_{\mathcal{P}})$ ▷ `partial results over partitions`
/* **PHASE #2a**: *cross-partition search in pairs of adjacent bands* */
7 $\mathcal{B} \leftarrow$ create bands of width ϵ_{sp} inwards each side of every $p \in \mathcal{P}$
8 $R_{\mathcal{B}} \leftarrow$ filter $R_{\mathcal{P}}$ by $\mathcal{B}$ and build a local $\mathcal{X}$-index per band $b \in \mathcal{B}$
9 $S_{\mathcal{B}} \leftarrow$ filter $S_{\mathcal{P}}$ by $\mathcal{B}$ and build a local $\mathcal{X}$-index per band $b \in \mathcal{B}$
10 $L_{\mathcal{B}} \leftarrow \{(r \in R_{\mathcal{B}}, s \in S_{\mathcal{B}})$: bands $r.b, s.b$ share a side in partitioning $\mathcal{P}\}$
11 $Q_{\mathcal{B}} \leftarrow blockwiseSimJoin(L_{\mathcal{B}}, R_{\mathcal{B}}, S_{\mathcal{B}}, \epsilon_{sp}, \epsilon_{ts})$
12 $storeHDFS(Q_{\mathcal{B}})$ ▷ `partial results over bands`
/* **PHASE #2b**: *cross-partition search in corner-wise pairs of boxes* */
13 $\mathcal{C} \leftarrow$ create boxes of side ϵ_{sp} at the corners of each partition $p \in \mathcal{P}$
14 $R_{\mathcal{C}} \leftarrow$ filter $R_{\mathcal{P}}$ by $\mathcal{C}$ and build a local $\mathcal{X}$-index per box $c \in \mathcal{C}$
15 $S_{\mathcal{C}} \leftarrow$ filter $S_{\mathcal{P}}$ by $\mathcal{C}$ and build a local $\mathcal{X}$-index per box $c \in \mathcal{C}$
16 $L_{\mathcal{C}} \leftarrow \{(r \in R_{\mathcal{C}}, s \in S_{\mathcal{C}})$: boxes $r.c, s.c$ share a single corner in $\mathcal{P}\}$
17 $Q_{\mathcal{C}} \leftarrow blockwiseSimJoin(L_{\mathcal{C}}, R_{\mathcal{C}}, S_{\mathcal{C}}, \epsilon_{sp}, \epsilon_{ts})$
18 $storeHDFS(Q_{\mathcal{C}})$ ▷ `partial results over boxes`
19 **Function** $blockwiseSimJoin(L, R, S, \epsilon_{sp}, \epsilon_{ts})$
20 $Q \leftarrow \emptyset$
21 **foreach** *block pair* $(a, b) \in L$ **do** ▷ `local search`
22 $\mathcal{I}_a^R \leftarrow$ local $\mathcal{X}$-index available for dataset R in block a
23 $\mathcal{I}_b^S \leftarrow$ local $\mathcal{X}$-index available for dataset S in block b
24 $Q \leftarrow Q \cup SimJoin\mathcal{X}(\mathcal{I}_a^R.root, \mathcal{I}_b^S.root, \epsilon_{sp}, \epsilon_{ts})$
25 **return** Q ▷ `results collected from all pairs of blocks`

Once pairs of blocks need be checked at each tier (either $L_{\mathcal{P}}$ for partitions, or $L_{\mathcal{B}}$ for bands, or $L_{\mathcal{C}}$ for boxes), *blockwiseSimJoin* (Lines 19-25) takes advantage of the created indices and applies the respective method from Section 4.2.2 to return their results. Each pair of blocks at any tier can be processed independently. Hence, for a given partitioning $\mathcal{P}$, once the query is submitted, the required subsets and their indices can be prepared in a *distributed* fashion and the respective block-wise checking can be evaluated *in parallel*. For a given partition p (first tier), subsets from both datasets are assigned to the same worker node. This policy is also applied in the case of blocks that need to be checked: the worker responsible for a

given partition p receives the data and index concerning geolocated time series from the other dataset within an adjacent block. Such processing fits well under the *MapReduce* paradigm; mappers assign data subsets to workers according to partitioning scheme $\mathcal{P}$. Each worker employs reduce operations to generate the respective indices and store them on HDFS. At query time, a map procedure assigns the indices that reside within each partition to a reducer, which calculates the local results. Simultaneously, mappers shuffle data per block to workers responsible for their neighboring partitions. Finally, a reduce operation is carried out on each pair of such blocks to compute their similarity joins and to store results on HDFS.

Overall, this method manages to provide correct and complete results for any similarity join query over two datasets of geolocated time series. This is stated in the following:

Lemma 1. *Algorithm 5 issues all qualifying results of similarity join between two datasets $\mathcal{T}_R$, $\mathcal{T}_S$ of geolocated time series, without probing candidate pairs more than once and without any false misses.*

Proof. Regarding *correctness*, consider a given partition $p \in \mathcal{P}$, as depicted in Figure 4.3 and let R_p be the geolocated time series of $\mathcal{T}_R$ having their location contained therein. Obviously, any of their possibly qualifying pairs from dataset $\mathcal{T}_S$ must be within distance ϵ_{sp}. So, it suffices to examine similarity between geolocated time series in R_p with those of $\mathcal{T}_S$ topologically located within a *buffer* that expands partition p by distance ϵ_{sp}. Clearly, the area covered by the nine blocks (one partition, four bands, and four boxes) concerning $\mathcal{T}_S$ is exactly this buffer zone, so $Q_{cand}(p) = \{(T_R, T_S) : within(T_R.loc, p), within(T_S.loc, buffer(p, \epsilon_{sp}))\}$ provides all possible candidates in a given partition p. As partitions hold disjoint subsets of the raw data, iterating with the same logic over each partition $p \in \mathcal{P}$, confirms that all candidates are examined.

Regarding *completeness*, observe that the pairs of blocks involved at each stage include all possible candidates to probe from each dataset. At the first tier, searching in each partition p (common for either dataset) provides all qualifying pairs having their constituent geolocated time series both located in p. Cross-searching beyond the boundary of each partition p is meaningful only along the adjacent bands and boxes, each of them coming from a distinct neighboring partition to p. Clearly, in each of those nine blocks, a disjoint subset of candidate pairs from the two datasets is examined and their union is $Q_{cand}(p)$. Hence, each candidate pair is probed only once, and no qualifying results can ever be missing from the final answer. □

4.3.2 Minimizing Data Shuffling

Recall that a worker w already has locally available all data for subsets of $\mathcal{T}_R, \mathcal{T}_S$ located in its assigned partition p. This is sufficient for its own local search, but w still needs to send raw data concerning its four bands and four boxes for the cross-partition search.

This evaluation strategy can be further optimized by minimizing the amount of data that needs to be shuffled during cross-partition search. We introduce an intermediate filtering step that takes advantage of the pruning power of our BTSR-tree index. Consider a given block a over dataset $\mathcal{T}_R$ held in worker w' that must be shipped to worker w responsible for partition p. Instead, w' builds a BTSR-tree $\mathcal{I}_a^R$ over this subset in a, and sends this index only to worker w, which builds its own BTSR-tree $\mathcal{I}_b^S$ over its local subset of $\mathcal{T}_S$ within its corresponding block b. Checking for similarity joins against those two indices can be carried out with Algorithm 4.

This returns pairs $\{(mbr_i^a, mbr_j^b)\}$ of overlapping MBRs, where mbr_i^a is an MBR over block a and mbr_j^b is over block b, and each one contains candidate geolocated time series for refinement. The list $\{mbr_i^a\}$ of all identified MBRs concerning block a is returned to worker w' and the raw geolocated time series within each such MBR can be readily accessed thanks to the already available BTSR-tree $\mathcal{I}_a^R$. Those MBR-filtered time series are then shipped to worker w, which also retrieves its own raw data from BTSR-tree $\mathcal{I}_b^S$ concerning those MBRs $\{mbr_j^b\}$ identified for its own block b. Finally, those two MBR-filtered subsets of geolocated time series are each indexed with a new BTSR-tree and joined according to the similarity criteria to yield their matching results. As confirmed in our empirical tests, this index-guided shuffling can reduce the raw data transferred between workers by more than 50% without sacrificing performance.

4.4 Experimental Evaluation

This chapter experimentally evaluates all our hybrid queries processing approaches and discusses the obtained results. First, we evaluate the TSR-tree and BTSR-tree indices in terms of construction time and index size, in contrast with the standard R-tree. Then we evaluate the performance of TSR-tree and BTSR-tree against a standard R-tree implementation, where we first apply spatial filtering and then scan through all the obtained geolocated time series to keep only the ones that satisfy ϵ_{ts}. Next, we evaluate our hybrid similarity join approach using iSAX, R-tree and BTSR-tree. Finally, we compare our distributed similarity join algorithm against a basic version that utilizes standard R-trees for local indexing. We first describe our experimental setup, the datasets and the applied parameterizations.

4.4.1 Experimental Setup

Table 4.2: Datasets and default thresholds used in the similarity search experiments.

Dataset	Area (km^2)	Number of locations	Length of timeseries	Default query thresholds θ_{sp}	θ_{ts}	θ_h
Water	114	200,000	168	0.15	0.175	0.15
Taxi	2,500	417,960	168	0.2	0.001	0.0025
Flickr	Earth	414,967	96	0.25	0.0005	0.001
Crime	392,000	362,215	76	0.3	0.01	0.02

4.4.1.1 Datasets

We use four real-world datasets, which are selected from various application domains and have different characteristics, in order to evaluate our approach with diverse types of geolocated time series. Next, we describe the characteristics of each dataset. A summary is listed in Table 4.2.

DAIAD Water Consumption (Water). Courtesy of the DAIAD project (http://daiad.eu/), we acquired a geolocated time series dataset of hourly water consumption for 822 households in Alicante, Spain from 1/1/2015 to 20/1/2017. In order to get a more representative dataset for our tests, we first calculated the weekly (24×7) time series per household by averaging corresponding hourly values over the entire period. Then, these weekly time series were used as seeds in order to synthetically increase the size of the dataset to 200,000, by introducing some random variations in their location and pattern.

NYC taxi drop-offs (Taxi). This dataset contains time series extracted from yellow taxi rides in New York City during 2015. The original data[2] provide pick-up and drop-off locations, as well as corresponding timestamps for each ride. For each month, we generated time series by applying a uniform spatial grid over the entire city (cell side was 200 meters) and counting all drop-offs therein for each day of the week at the time granularity of one hour. Thus, we obtained the number of drop-offs for 24×7 time intervals in every cell, which essentially captures the weekly fluctuation of taxi destinations there. Without loss of generality, the centroid of each cell is used as the geolocation of the corresponding time series.

Flickr geotagged photos (Flickr). This dataset contains time series data extracted from geolocated Flickr images between 2006 and 2013 over the entire planet[3]. In order to get meaningful geolocated time series, we partitioned the space by a uniform grid of 7200×3600 cells (each one spanning 0.05 decimal degrees in each dimension) and we counted the number of photos contained in every cell each month, excluding cells with no data at all (e.g., in the oceans). Each time series conveys the visits pattern (in terms of number of photos taken per month) of that region over this period.

UK historical crime data (Crime). This dataset contains time series representing the temporal variation in the number of crime incidents reported across England and Wales over 76 months (December 2010– March 2017). From the original data[4], we generated time series over a grid with cell size 200 meters. For each month, we counted incidents having their location within each cell.

For assessing the performance of hybrid similarity join, we generated several *synthetic* datasets of various sizes, using the DAIAD water consumption dataset as a seed. On this data, we first calculated the weekly (24×7) time series per household by averaging corresponding hourly values over the entire period. Then, these weekly sequences were used as seeds to synthetically increase the size of the dataset up to 2 million geolocated time series, by introducing small random variations in their location and pattern.

4.4.1.2 Index Parameters

For hybrid similarity search queries, we set the minimum (m) and maximum (M) number of entries stored in each node to 60 and 200, respectively. To insert time series in the TSR-tree and the BTSR-tree in the same manner as in the R-tree, the weight parameter λ, used in Equation 3.4 to select the node with the least cost during an insertion, is set to 1. This implies that only the spatial cost is considered during insertion. This way, the performance benefits observed in the experiments are only due to the pruning conditions. Finally, for the BTSR-tree, we fix the number of bundles $\beta_0 = 5$ for its leaf nodes; factor c, specifying the increase

[2] http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
[3] https://code.flickr.net/category/geo/
[4] https://data.police.uk/data/

(decrease) rate of the number of bundles (respectively, time series resolution) at each higher level in the tree hierarchy, is set to $c = 2$.

Regarding hybrid similarity join, we fine-tuned parameters for the various indices used against this data. For BTSR-trees and R-trees, the number of entries per node ranges between m=10 and M=50. In iSAX, up to M=250 time series can be stored per leaf and the length of each SAX word is w=8.

4.4.1.3 Query Parameters

For hybrid similarity search, the query parameters involve the different types of distance thresholds for queries involving boolean filtering, namely spatial (θ_{sp}), time series (θ_{ts}) and hybrid (θ_h), as well as the number k of results to return for queries involving top-k filtering. Values to these parameters are set differently for each dataset, based on their characteristics; default values are shown in Table 4.2. Moreover, for queries involving hybrid distance, we fix the exponential decay constant γ to 0.025 for the Water and Taxi, 0.001 for the Flickr and 0.05 for the Crime dataset, to better reflect the spatial distribution and coverage of each particular dataset.

For hybrid similarity join, Table 4.3 lists the range of values for the rest of parameters used in our tests; default values are in bold.

Table 4.3: Parameters tested in the hybrid similarity join experiments.

Parameter	*Values*
Dataset size (*centralized*)	50K, 100K, 150K, **200K**, 500K, 1000K
Dataset size (*distributed*)	500K, **1000K**, 1500K, 2000K
Number of partitions $g \times g$ (*distributed*)	10^2, 20^2, $\mathbf{30^2}$,40^2,50^2,60^2,70^2
Distance radius ϵ_{sp} (meters) in queries	100, 125, **150**, 175, 200
Time series deviation ϵ_{ts} in queries	0.3, **0.35**, 0.4, 0.45, 0.5

4.4.1.4 Evaluation Setting

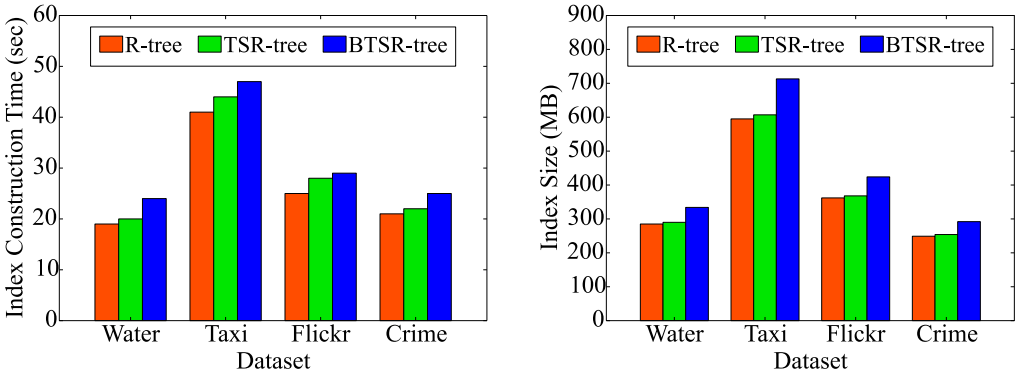
All algorithms were implemented in Java. For hybrid similarity search, we compare the performance of our hybrid indices to the standard R-tree, used as baseline. In the R-tree, query evaluation is done by traversing the index according to the spatial predicate of the query, retrieving a set of intermediate results that satisfy the spatial condition, and then filtering out candidates according to the time series predicate to produce the final result set. We measure the portion of tree nodes accessed by each method, since this is the dominant performance factor for each index. Moreover, for each index, we measure its size and the time required for its construction. The implementations of all indices are in-memory and developed in Java. The tests are run on a Debian Linux machine with 4 CPUs, each containing 8 cores clocked at 2.13GHz, and 256 GB RAM. Query workloads were created by randomly picking 500 geolocated time series separately for each dataset.

Regarding hybrid similarity join, distributed methods were developed on Apache Spark 2.3.0. The centralized experiments were executed on a machine running MacOS 10.13.5 with a 2GHz CPU and 8GB of RAM. The distributed tests were conducted on a cluster with 7 virtual machines running Ubuntu 16.04.3 LTS, with 4 cores each, clocked at 2100MHz. Each

node had a total of 5GB of RAM. Next, we report performance in terms of average response time per query. Each query runs against two instances of the same dataset (i.e., *self-join*), excluding identity matches from resulting pairs. In the distributed case, we also measure the amount of raw data transferred between workers during the cross-partition phase.

4.4.2 Index Construction Time and Size

Figure 4.4 shows the index construction time and the resulting index size (i.e., memory footprint) for each dataset. Construction time of the TSR-tree index is slightly higher than that of the R-tree, and the same holds for the index size. This is natural, because after building the spatial part of the index, it has to be traversed in order to calculate the MBTS of each node, which incurs additional cost both for computing and for storing it in each node. Compared to the TSR-tree, building the BTSR-tree index is even more costly, since it incurs an additional time and space overhead to compute the time series bundles in each node and maintain the MBTS of each bundle. Still, these costs are not considered significant. Even for the largest dataset (Taxi), which contains $417,960$ time series and has an original size of 153 MB, the index construction time is around 43 seconds for TSR-tree and 45 seconds for BTSR-tree, while the index size is around 610 MB and 710 MB, respectively. Note that the extra space for the BTSR-tree is needed for storing the bounds for each of the progressively multiple bundles in internal nodes.



(a) Index construction time

(b) Index size

Figure 4.4: Comparison of index construction time and size for each dataset.

4.4.3 Hybrid Similarity Search Query Performance

We compare the percentage of nodes accessed by each index in each query. All measurements are average cost per query in each particular workload.

Q_{bb}**.** Figure 4.5 illustrates the performance of the Q_{bb} query on each dataset for varying spatial distance threshold (θ_{sp}). Both the TSR-tree and the BTSR-tree clearly outperform the standard R-tree in all datasets, with gains in performance increasing even further as the threshold θ_{sp} increases. The standard R-tree needs to access a significant amount of nodes

as it can only prune in the spatial domain. As a result, it performs increasingly worse than the TSR-tree and the BTSR-tree. Due to its tighter bounds, the BTSR-tree manages to prune more nodes than the TSR-tree, especially for larger spatial threshold values.

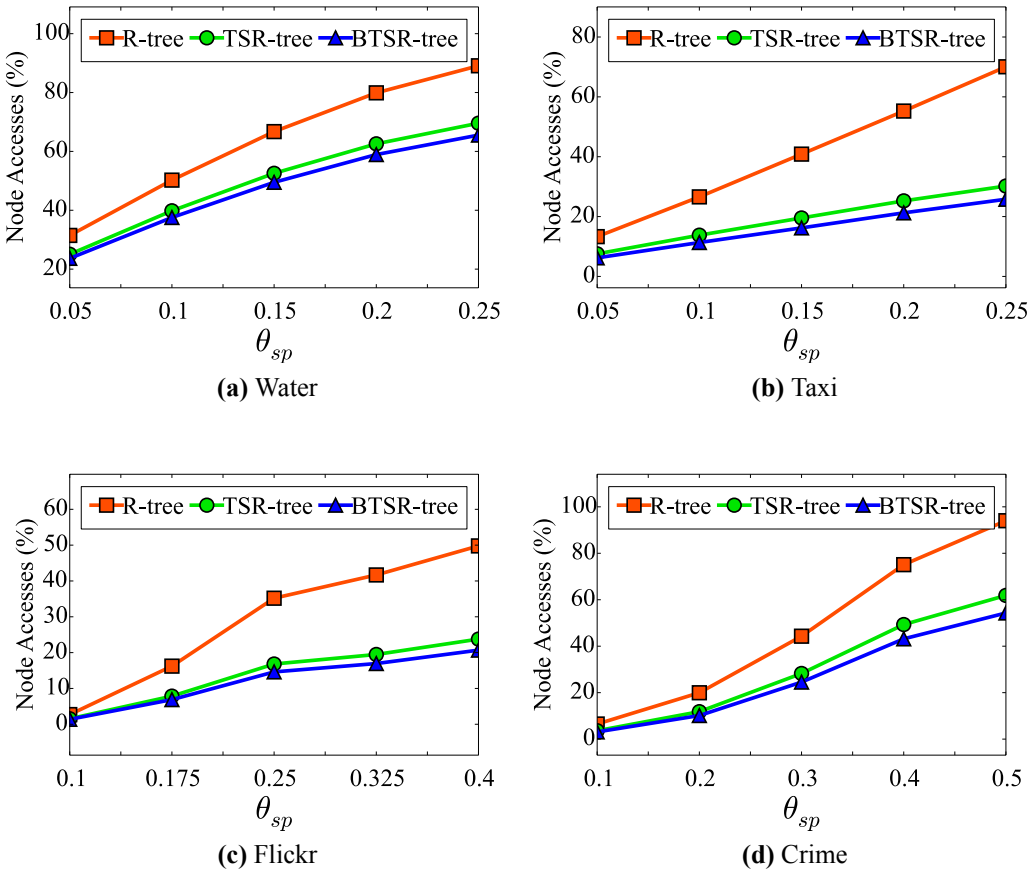


Figure 4.5: Query $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$ with varying spatial distance threshold θ_{sp}.

Obviously, increasing the time series threshold (Figure 4.6) has absolutely no impact on the performance of the standard R-tree. On the contrary, the node accesses required by the TSR-tree and the BTSR-tree are much lower. Nevertheless, these also increase with more relaxed θ_{ts}, asymptotically reaching those of the R-tree. Performance of the BTSR-tree is better for lower values of θ_{ts}, as it allows more pruning thanks to the tighter bounds of bundles. It is apparent that the sensitivity of the threshold heavily depends on the dataset. Even slightly increasing θ_{ts} in the Taxi and Flickr datasets (Figure 4.6b and 4.6c), significantly affects performance of both the TSR-tree and the BTSR-tree, as they are more sparse than the Water and Crime datasets (Figure 4.6a and 4.6d), with a large number of time series having many zero values. Consequently, even a small increase in this threshold causes a larger amount of nodes to be probed.

Q_{kb}. Performance of Q_{kb} for varying number k of results is depicted in Figure 4.7. In all cases, both the TSR-tree and the BTSR-tree cope better compared to the standard R-tree. Varying the number k of results does not significantly affect performance, as nearby elements

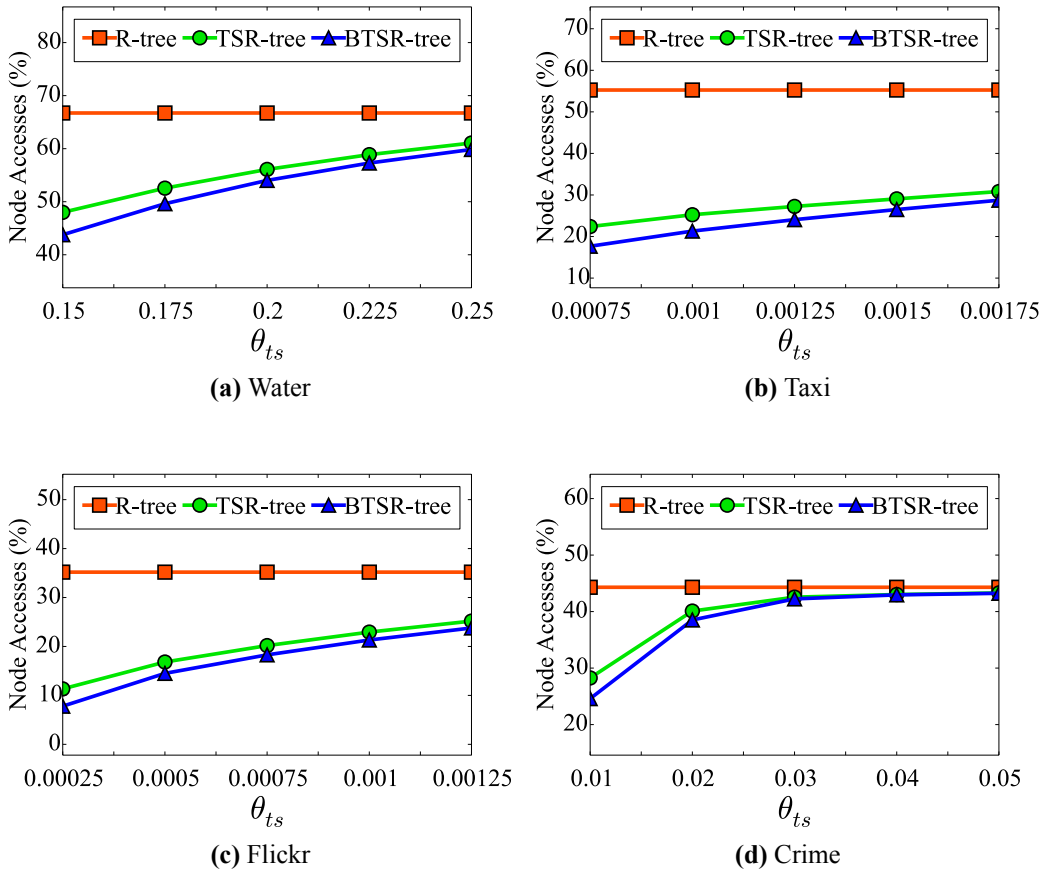


Figure 4.6: Query $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$ with varying time series distance threshold θ_{ts}.

tend to have similar time series values and all required results are found in close distance. This is an interesting insight, especially for the Water dataset, which suggests that neighboring households tend to have similar water consumption. The R-tree performs really poor in the Crime dataset (Figure 4.7d), requiring a full traversal (100% of its nodes), as the sought number of results cannot be found. This is not the case with the TSR-tree and the BTSR-tree, respectively accessing 65% and 58% of their nodes due to their effective pruning in the time series domain.

Increasing the time series threshold θ_{ts} (Figure 4.8) in Q_{kb}, improves performance of the R-tree and it is sometimes competitive to that of the TSR-tree and the BTSR-tree, as more nearby elements qualify as results and are thus quickly found. However, this is strongly influenced by dataset characteristics. For the Taxi dataset (Figure 4.8b), node accesses in the R-tree are similar for different values of θ_{ts}, but performance for the BTSR-tree slightly decreases with higher threshold values. Indeed, relaxing this threshold effectively allows more tolerance in the similarity of time series and thus, extra nodes need to be visited for checking.

Q_{bk}. Varying the number k of results in the Q_{bk} query (Figure 4.9), shows similar behavior to the Q_{kb} query. Similar time series are located close to each other, so k results are quickly obtained once the first qualifying time series is retrieved. The R-tree is always significantly

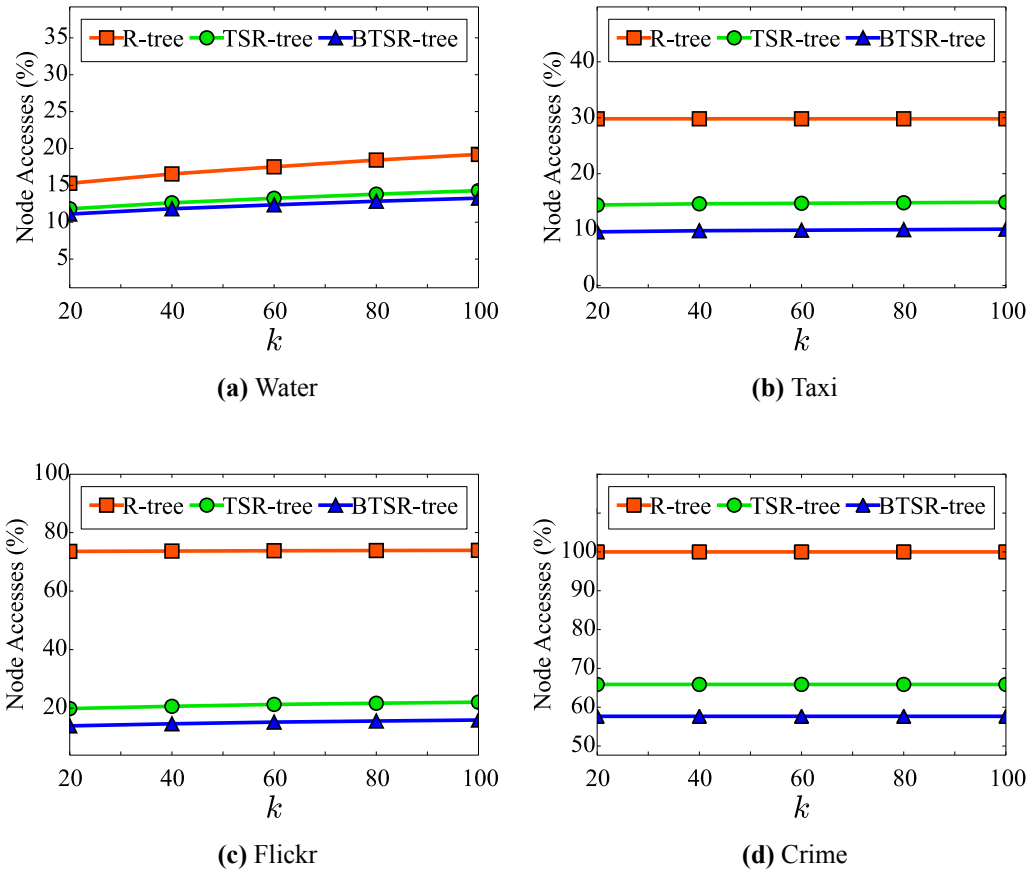


Figure 4.7: Query $Q_{kb}(T_q, k, \theta_{ts})$ with varying number k of results.

outperformed by the TSR-tree and BTSR-tree; the effect is less apparent in the Crime dataset (Figure 4.9d).

Figure 4.10 illustrates performance of the Q_{bk} query for varying spatial distance threshold (θ_{sp}). The significantly better performance of the TSR-tree and the BTSR-tree is also apparent here, with the difference getting more pronounced with larger thresholds (i.e., wider radii). This advantage is less manifest in the Crime dataset (Figure 4.10d) because the applied thresholds cover increasingly larger spatial areas (practically the entire dataset for $\theta_{sp} = 0.5$), thus more nodes need to be examined in order to identify the k results. In the Taxi dataset, the performance improvement of the BTSR-tree compared to the TSR-tree is smaller due to data sparsity, which diminishes the pruning effect of bundles in the nodes of the BTSR-tree.

Q_{hb} and Q_{hk}. First, note that such hybrid queries cannot be possibly applied on the R-tree at all. Regarding the Q_{hb} query, Figure 4.11 plots performance for varying hybrid distance threshold (θ_{hb}). The BTSR-tree fares better than the TSR-tree in all cases. The effect is more intense with smaller θ_{hb} values, since the tighter bundles in the BTSR-tree allow more effective pruning. For each dataset, divergence in performance between the two indices largely depends on variance among closely located time series. For instance, taxi dropoffs exhibit similar patterns locally, so the derived bounds also tend to be similar, diminishing the prun-

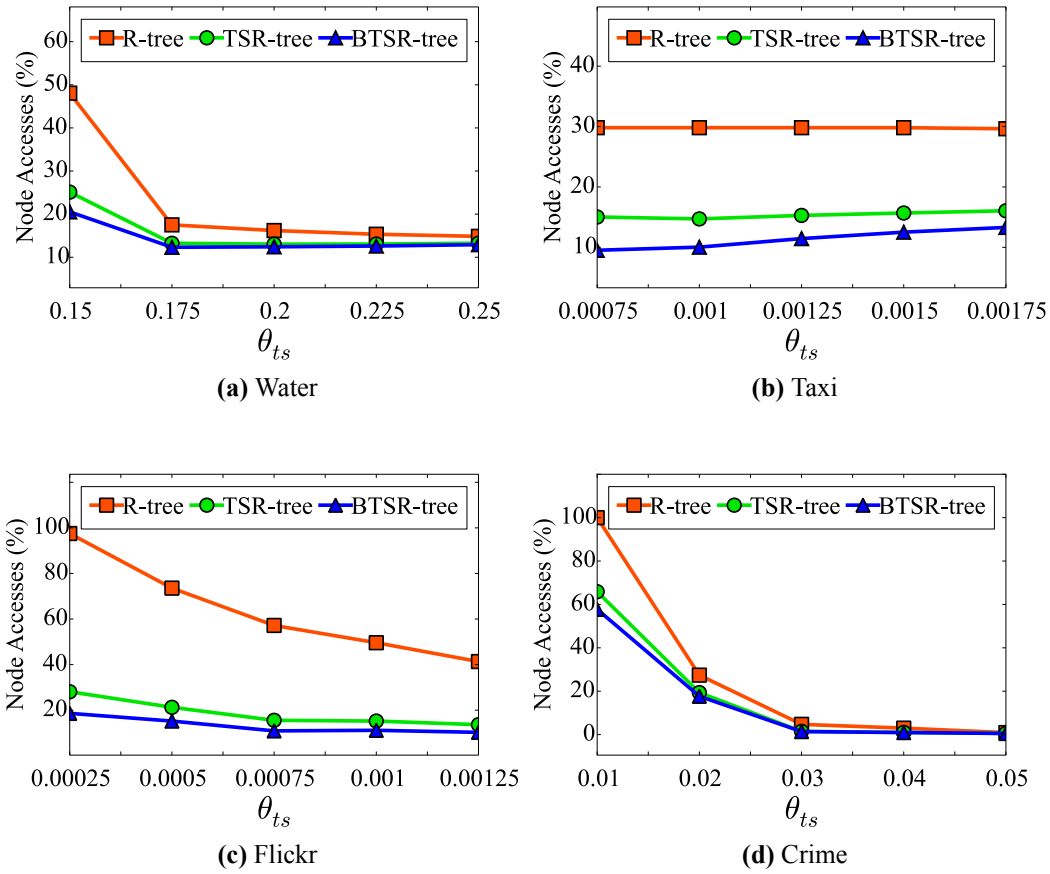


Figure 4.8: Query $Q_{kb}(T_q, k, \theta_{ts})$ with varying time series distance threshold θ_{ts}.

ing power of both indices as shown in Figure 4.11b. Instead, when nearby time series have many diverse patterns, the BTSR-tree becomes more effective by allocating them to distinct bundles and offering considerable performance gain as for the Water dataset (Figure 4.11a).

Finally, Figure 4.12 depicts the performance of Q_{hk} for varying number k of results. BTSR-tree always outperforms TSR-tree with a margin of more than 10% node accesses on average, again with the exception of the Taxi dataset (Figure 4.12b), as mentioned before.

4.4.3.1 Centralized HSJ Results

Figure 4.13 depicts performance for different parameter values and dataset sizes. The iSAX-based algorithm performs significantly worse than the rest, mostly because each node comparison involves reconversion of the iSAX symbols to the Euclidean space [SK08] and consequently, calculation of Euclidean distances over long sequences (up to 168 values in this data). BTSR-tree is superior in all cases, as it is able to prune in both time series and spatial domains. As shown in Figure 4.13a, BTSR-tree and R-tree-based methods perform similarly for smaller ϵ_{sp} values, as fewer candidates are found and need refinement in the time series domain. However, as the distance radius ϵ_{sp} is relaxed, R-tree search worsens significantly, while BTSR-tree still copes well due to its hybrid pruning ability. iSAX-based search is

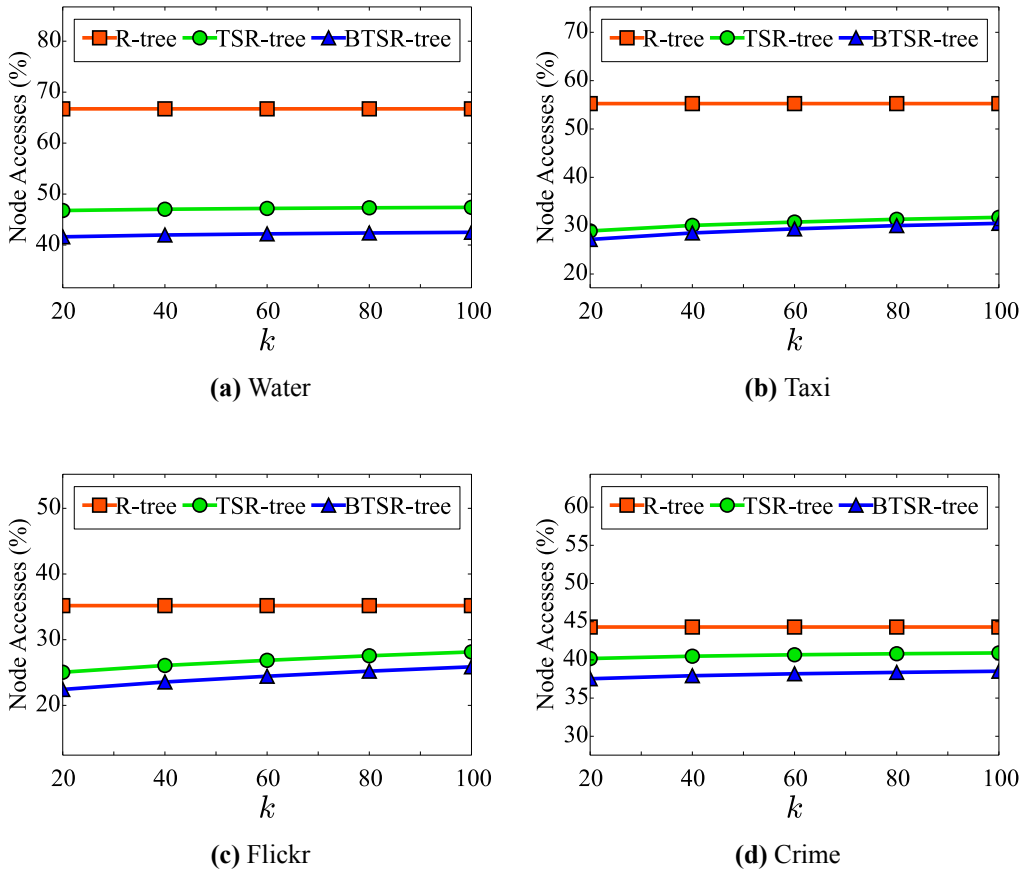


Figure 4.9: Query $Q_{bk}(T_q, \theta_{sp}, k)$ with varying number k of results.

immune to different values of ϵ_{sp}, as filtering with spatial distance is only involved at refinement. With varying ϵ_{ts} values (Figure 4.13b), BTSR-tree and R-tree approaches have no fluctuations in performance, as ϵ_{sp} is fixed and refinement of candidates involves a similar cost in the time series domain. However, using iSAX indexing is faster for lower ϵ_{ts} values and performance slowly degrades for larger ϵ_{ts}, as more candidates become eligible. In terms of scalability (Figure 4.13c), all algorithms are almost equally fast over small datasets. But, as dataset sizes grow, the BTSR-tree approach scales better thanks to its hybrid pruning, although the number of matching pairs escalates. Indicatively, for 100K input data, we get 2K qualifying pairs; in the 500K dataset, we get 40K results. For input data sizes larger than 500K, all centralized methods fail to finish execution, issuing an out-of-memory error. This manifests the necessity of distributed processing schemes for similarity joins over larger datasets.

4.4.3.2 Distributed HSJ Results

First, we compare *SimJoinMR* (using R-trees for local indexing) with its *SimJoinOPT* variant (employing BTSR-trees) for varying ϵ_{sp} values. It is apparent from Figure 4.14a that query response times for the *SimJoinMR* method are increasing, since the underlying R-trees fare

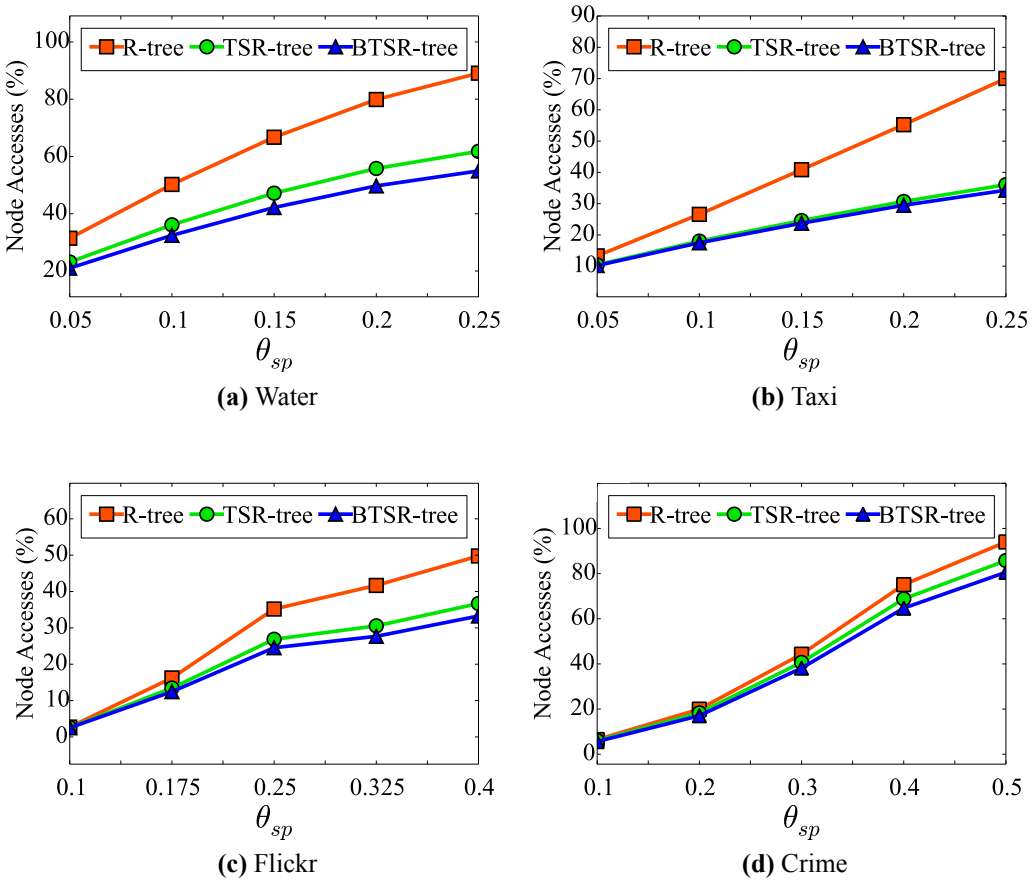


Figure 4.10: Query $Q_{bk}(T_q, \theta_{sp}, k)$ with varying spatial distance threshold θ_{sp}.

worse for larger distance radii. With larger ϵ_{sp} values, more raw data has to be shuffled between workers during the cross-partition checks, as the size of bands and boxes involved gets bigger and covers more candidate geolocated time series. Concerning exactly this shuffling overhead, Figure 4.14b reveals that this is indeed lower in the *SimJoinOPT* variant, which explains its processing cost advantage. Finally, Figure 4.14c illustrates the number of results produced from the two stages; first locally in each partition, and then after cross-partition checks in bands and boxes. As distance constraint ϵ_{sp} gets more relaxed, more pairs qualify as answers. For smaller ϵ_{sp}, the majority of results come locally from each partition. But as ϵ_{sp} is relaxed, many more pairs are found in neighboring partitions, as bands and boxes also become larger and increase their share in qualifying results much more.

With regard to increasing ϵ_{ts} values, observe in Figure 4.15a that method *SimJoinMR* is consistently worse than *SimJoinOPT*, basically due to the different pruning power of their respective indices. The former relies on R-trees, which have no effect with varying ϵ_{ts}; in contrast, BTSR-trees employed by *SimJoinOPT* can effectively filter candidates also in the time series domain. With a more relaxed ϵ_{ts}, more results qualify, hence the linear increase in processing cost. Regarding the data shuffling overhead, this is practically stable for each method irrespective of the ϵ_{ts} constraint (Figure 4.15b). In *SimJoinMR*, selection of geolocated time series that should be transmitted is solely based on their spatial containment in

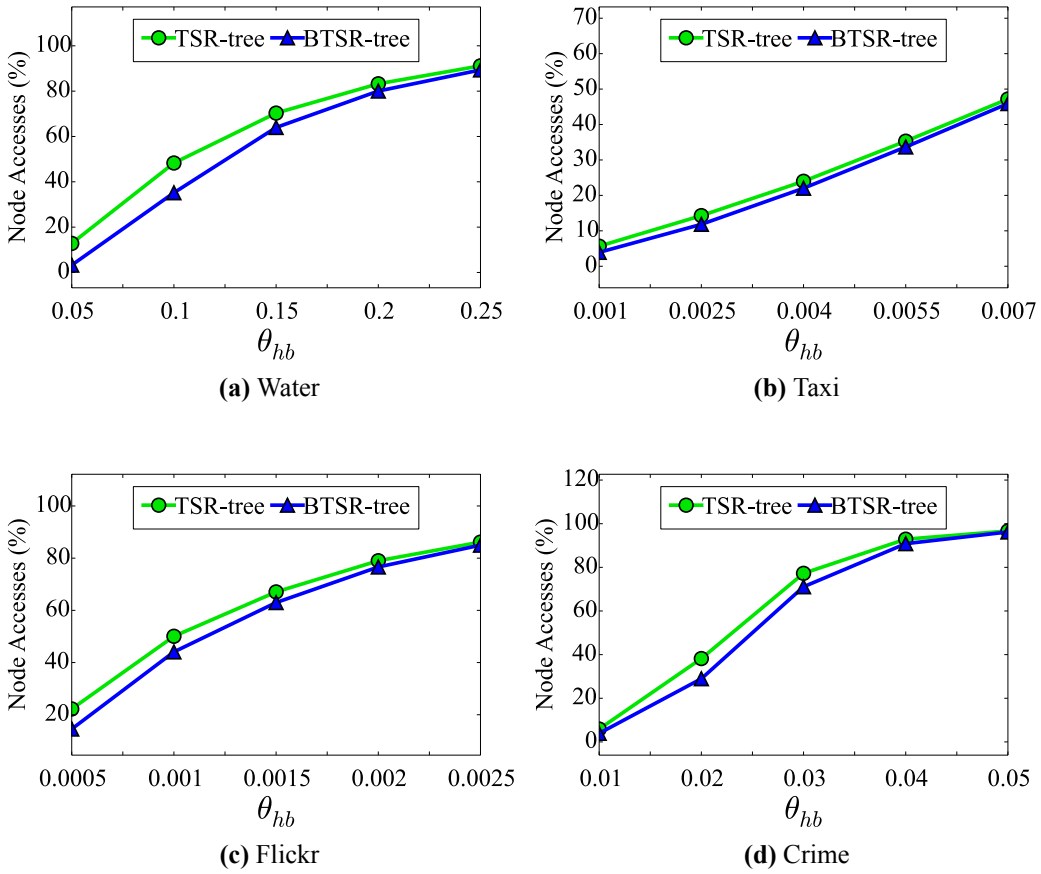
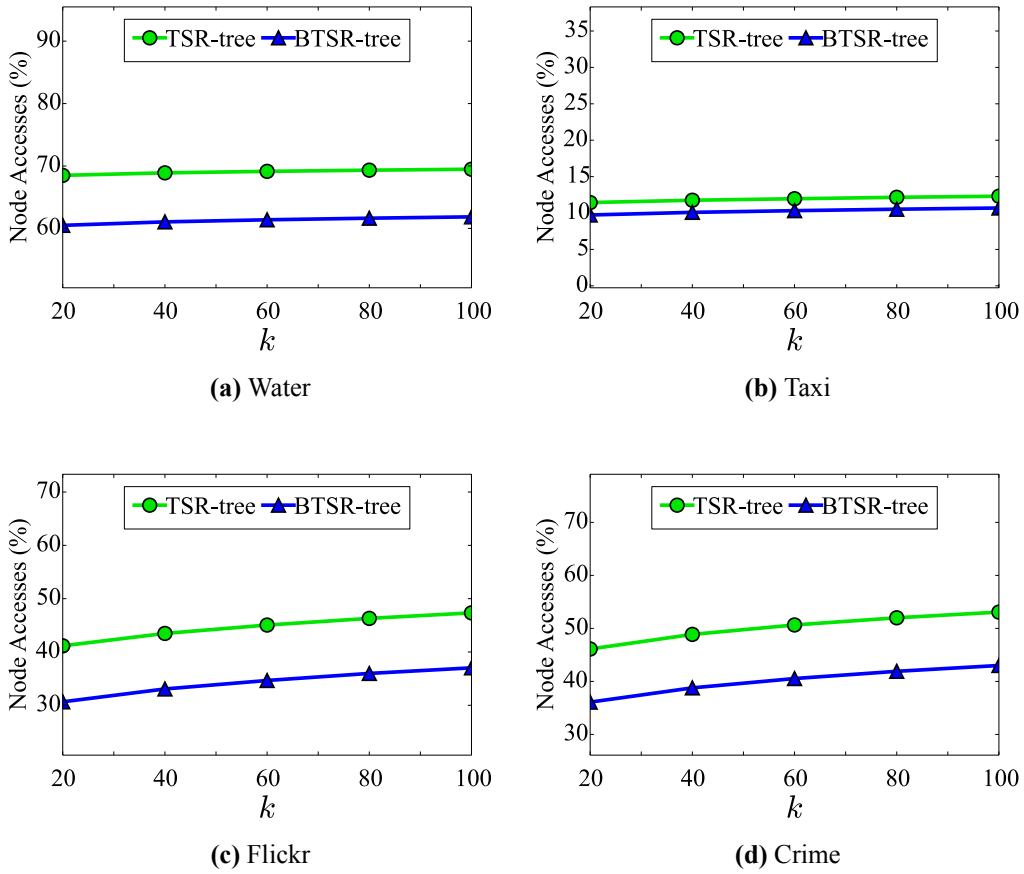


Figure 4.11: Query $Q_{hb}(T_q, \theta_h, \gamma)$ with varying hybrid distance threshold θ_h.

the respective bands and corner-wise boxes. But *SimJoinOPT* avoids many irrelevant transfers, as it also uses filtering with ϵ_{ts}; the amount of dispatched geolocated time series is only slightly increasing with ϵ_{ts}. Regarding the number of generated results, Figure 4.15c reveals a rather steep increase for small variations of ϵ_{ts}, which indicates that most time series are clustered within a small range of ϵ_{ts} deviations. As original data concern water consumption, this explains such highly correlated behavior, especially among neighboring households; of course, this pattern is replicated in the synthetic data as well. The percentage of results from cross-partition checks in each full answer is similar across various ϵ_{ts} values, as distance ϵ_{sp} is fixed and so are the respective bands and boxes involved in this phase.

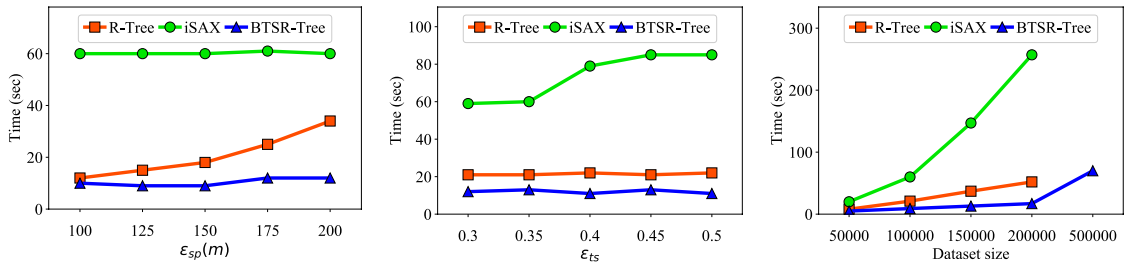
Figure 4.16 concerns scalability of the distributed methods with increasing dataset sizes. For smaller datasets, both methods are competitive, but response times for *SimJoinMR* escalate with larger sizes. With 2 million geolocated time series as input, this method did not finish, as it required traversal of too many paths in its underlying R-trees per partition, exceeding the capabilities of the workers. Regarding communication (Figure 4.16b), *SimJoinMR* requires shuffling of more raw data, especially for input size of 1.5 million. In contrast, *SimJoinOPT* maintains lower communication overhead, as it uses light-weight indices to guide data shuffling. Figure 4.16c indicates that the number of results is growing according to the input size, as the spatial density also increases with larger synthetic datasets that still



(a) Water **(b)** Taxi

(c) Flickr **(d)** Crime

Figure 4.12: Query $Q_{hk}(T_q, k, \gamma)$ with varying number k of results.

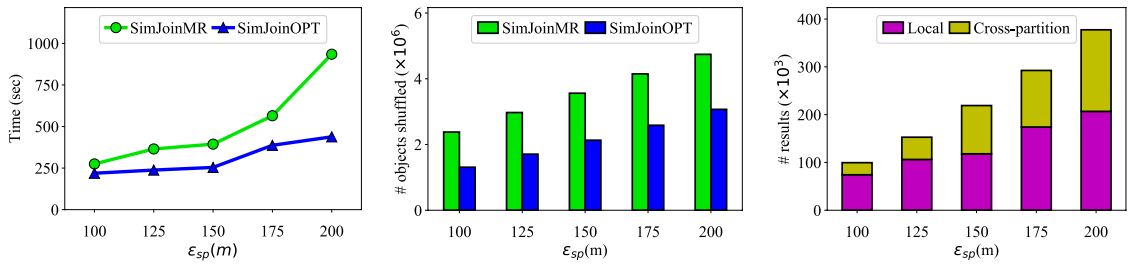


(a) Varying ϵ_{sp} **(b)** Varying ϵ_{ts} **(c)** Scalability

Figure 4.13: Processing cost for *centralized* execution of similarity join queries employing different indices.

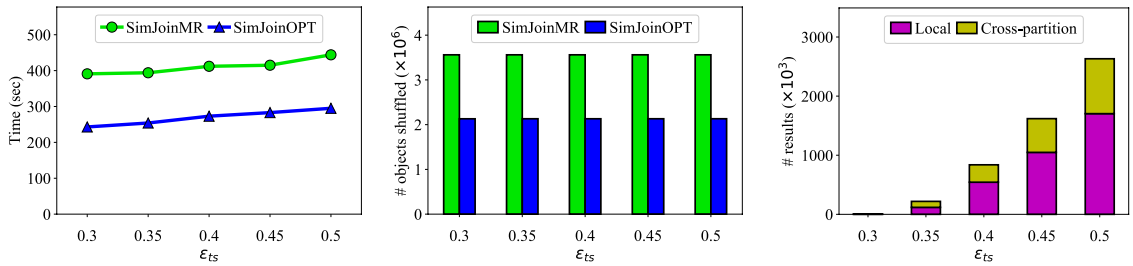
cover the same area (Alicante).

Last but not least, we conducted tests concerning partitioning, i.e., varying the grid granularity and distributing input data accordingly. As shown in Figure 4.17a, *SimJoinMR* was not able to conclude its evaluation over coarser spatial subdivisions, as each resulting partition can hardly cope with the larger subsets of data held locally. For 30×30 partitions, *SimJoinOPT* performs better thanks to the superiority of BTSR-tree in pruning. But *SimJoinMR*



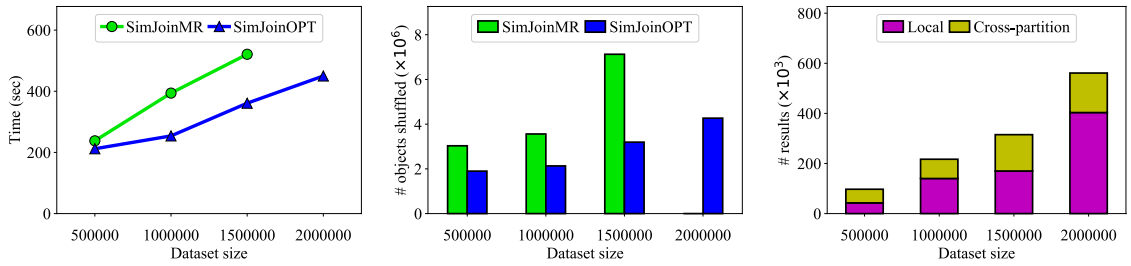
(a) Query response time **(b)** Geolocated time series shuffled **(c)** Query results per phase

Figure 4.14: Performance results for the *distributed* methods with varying ϵ_{sp}.



(a) Query response time **(b)** Geolocated time series shuffled **(c)** Query results per phase

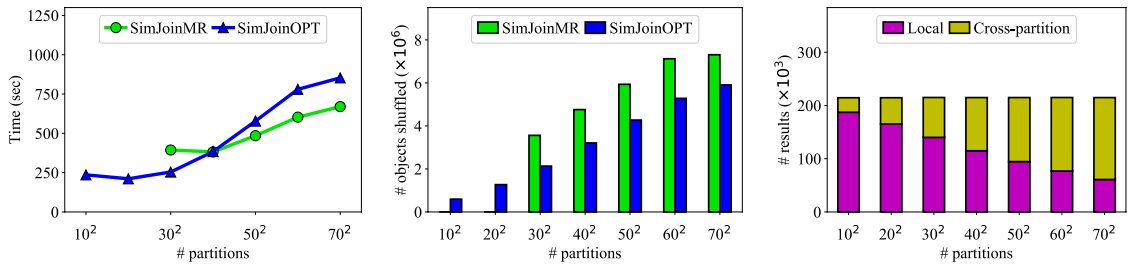
Figure 4.15: Performance results for the *distributed* methods with varying ϵ_{ts}.



(a) Query response time **(b)** Geolocated time series shuffled **(c)** Query results per phase

Figure 4.16: Scalability of the *distributed* methods.

overtakes *SimJoinOPT* when allowing finer partitioning (40×40 partitions or more), as the R-tree overhead diminishes. Each such index has to deal with smaller subsets, although it incurs higher communication overhead compared to *SimJoinOPT* (Figure 4.17b). Indeed, having more partitions forces *SimJoinOPT* to search for joins pairwise in many more bands and boxes, while also building the respective intermediate indices. So, such optimization really compensates with a coarser partitioning, achieving its best performance with a 20×20 grid as depicted in Figure 4.17a. Finally, Figure 4.17c indicates that the majority of resulting pairs are derived locally under a coarser partitioning. However, this is reversed with finer partitioning, as the size of blocks (bands and boxes) during the cross-partition checks cover much more area per cell, hence many more qualifying pairs are found while searching across neighboring partitions.



(a) Query response time **(b)** Geolocated time series shuffled **(c)** Query results per phase

Figure 4.17: Effect of partitioning on the performance of *distributed* methods.

4.5 Summary

In this chapter, we addressed the problem of hybrid similarity search and join over geolocated time series according to both their spatial proximity and time series similarity. Our approach takes advantage of different state-of-the-art indexing schemes to design an efficient algorithm for query evaluation. As far as similarity join is concerned, given that scalability is a bottleneck in such centralized settings, we show how a space-driven partitioning can be employed to deal with much larger datasets in cluster environments. Our parallel and distributed method can efficiently execute similarity joins per partition locally, while also minimizing the amount of data shuffled between processing nodes.

Our experiments against diverse real-world and synthetic datasets from different domains and varying sizes confirm the efficiency and effectiveness of our algorithms and show that the proposed indices significantly improve performance when evaluating hybrid similarity joins, as well as queries combining boolean and/or top-k filtering on both spatial proximity and time series similarity.

In the following chapter we introduce two visual exploration methods for geolocated time series datasets, that leverage the BTSR-tree index to speed-up the search and allow interactive exploratory visualizations.

Chapter 5

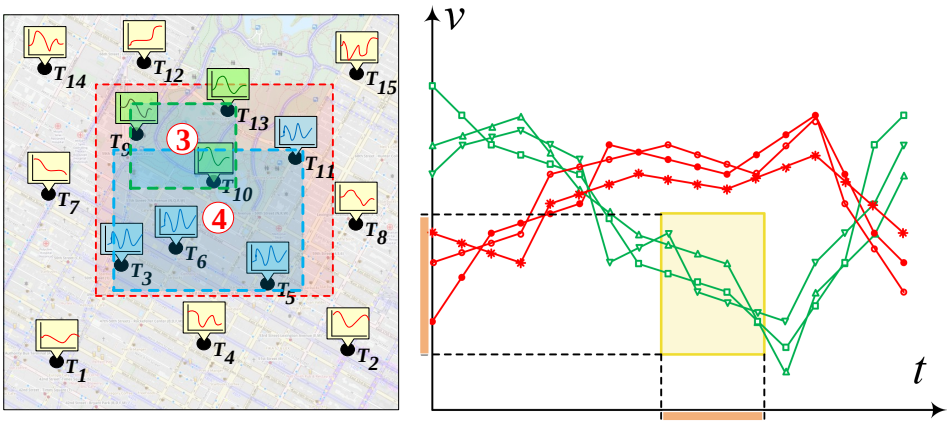
Visual Exploration of Geolocated Time Series

Time series is an inherently complex data type. Datasets containing time series can reach extremely large volumes, both *horizontally* (i.e., very long series of values across time) and *vertically* (i.e., time series generated by countless sources). Consequently, management, analysis and exploration of big time series data is a task requiring efficient and scalable algorithms. In particular, visual exploration of geolocated time series needs to process the required information efficiently, while the user interacts with the application. For example, whenever the user zooms in or scrolls the map, visual analytics and aggregates should be computed on-the-fly, e.g., identifying the predominant patterns in the time series and their spatial distribution within the actual map area.

Consider the example illustrated in Figure 5.1a. When the user zooms the map into the red rectangle, the visualization application should identify, summarize and present the two patterns (shown in blue and green color) appearing therein. For such requests that inherently combine spatial filters with time series analysis, it is inefficient to evaluate each predicate separately, e.g., apply a spatial filter on the time series of a large dataset and then calculate summaries of the candidates, or vice versa. The same stands for the case of exploration on the time series domain, as depicted in Figure 5.1b. Consider a user drawing a *timebox* (i.e., a rectangle in the time series domain) or zooming in the yellow part. The application should identify the time series that are fully contained within that filter area, i.e., their values along the specified time range fall within the value range (both ranges shown in orange in Figure 5.1b), and then provide an informative summary comprising aggregate spatial information to avoid cluttering the map.

Efficient filtering and retrieval over large datasets of geolocated time series can be enabled by *indexing*. Several approaches have been proposed that efficiently index large amounts of plain time series data. They either rely on *Discrete Wavelet Transform* to reduce the dimensionality of time series [CF99], or make use of a family of indices based on *Symbolic Aggregate Approximation* (SAX) [SK08, CPSK10, CSP+14, ZIP14]. However, all aforementioned techniques index the data solely on the time series domain, not taking the spatial dimension into account. If each analyzed time series is inherently associated with a spatial attribute (e.g., locations of smart meters), such indexing is not sufficient for queries and visualizations that additionally involve spatial filters.

In this chapter, we propose two geolocated time series summarization approaches for



(a) Exploring time series patterns in a *spatial region*. **(b)** Exploring time series in a *timebox* of interest.

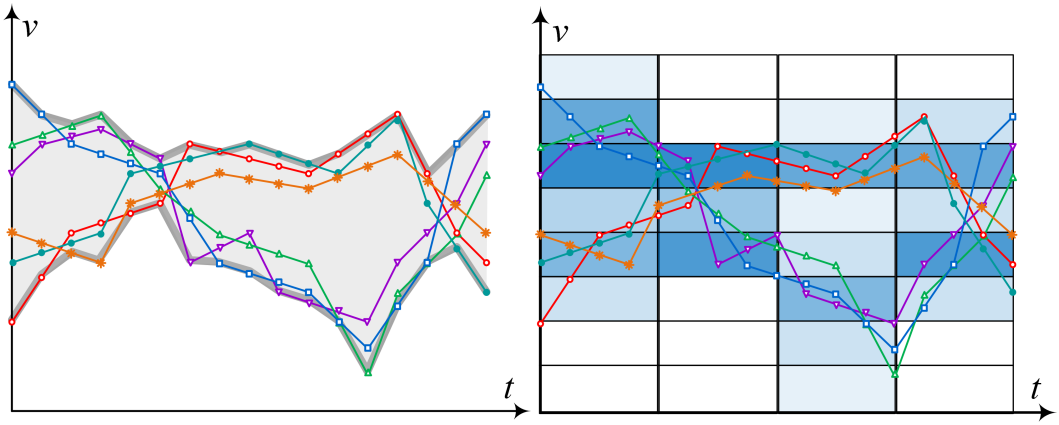
Figure 5.1: Visual exploration examples over geolocated time series.

visual exploration, named *bundle* and *tile map summary*. These are supported and driven by two appropriate *hybrid* indices that speed up the result computation, providing efficient exploration of geolocated time series data. They consist of a spatial and a time series summary that jointly facilitate knowledge extraction and insight gaining. The spatial summary is similar for both and consists of *Minimum Bounding Rectangles* (MBRs) of geolocated time series, according to a specific predicate (i.e., spatial proximity, or time series similarity). Each MBR is associated with a counter denoting the number of time series it contains. A visualization example of the spatial summary is depicted in Figure 5.1a, where the geolocated time series are organized in two groups (i.e., green and blue colored) according to their similarity. Each group is depicted along with a number that indicates the amount of time series that it contains (i.e., three geolocated time series for the first group and four for the second).

The main difference among the two methods lies in the time series part of the summary. The bundle summary consists of sets of MBTS, an example of which is depicted in Figure 5.2a. An MBTS is a band with upper and lower bounds that encloses all time series of a set, providing with a notion of a range of the time series values throughout the time axis. On the other hand, the tile map summary (Figure 5.2b) of a set of time series indicates (using a corresponding shading), the density of the time series points at each tile of a partitioning of the domain, obtained by discretizing the time and value axes. This way, it avoids overplotting that would be caused by outputting a large number of resulting time series and provides a notion of how the values of the time series are distributed across time.

For providing prompt visualizations of summaries over geolocated time series data and minimizing latency when drawing the relevant graphic elements, we need early access to both spatial and time series information while traversing the index. For this purpose, we adapt our BTSR-tree index so as to also include *aggregates* per node, i.e., the number of time series pertaining to each bundle. Subsequently, we introduce a new traversal algorithm for efficient retrieval of a given number of bundles that are the most representative in the map area.

The tile map summary is driven by geo-*i*SAX, a hybrid index we introduce in this chapter. This is a *time series-first* index, i.e., it is primarily built in the time series domain. More specifically, it constitutes a hybrid variant of the *i*SAX index [SK08, CPSK10, CSP+14],



(a) Bundle summary. **(b)** Tile map summary.

Figure 5.2: Examples of computed summaries on the time series domain.

augmented with spatial attributes of its nodes' children, to combine spatial and time series information. In each node, besides the SAX word that describes all its children time series, geo-iSAX keeps also the MBR that they form. To minimize the size and overlap of the MBRs, we propose a spatial splitting policy, that instead of choosing the splitting dimension in a round-robin fashion (as in iSAX), it does so by selecting the dimension that produces the smallest overlap and overall size of the two generated MBRs. We introduce a traversal algorithm for applying timebox search on large (both vertically and horizontally) geolocated time series datasets. The traversal algorithm is applied on our geo-iSAX index and returns a tile map-like summary of the qualifying geolocated time series, by taking advantage of the SAX representation's properties.

To the best of our knowledge, this is the first work that considers visual exploration and summarization of geolocated time series. Specifically, we propose two summarization methods enabling efficient map-based exploration driven by suitable hybrid indices.

The rest of this chapter is organized as follows. Section 5.1 outlines basic concepts and formulates the problem. Sections 5.2 and 5.3 introduce our methods for efficient visual exploration of geolocated time series by harnessing the potential of the BTSR-tree and geo-iSAX indices, respectively. Section 5.3.3 presents indicative use cases with map visualizations and also reports performance results from our empirical study. Finally, Section 5.4 concludes the chapter.

5.1 Summaries for Exploration

Our objective in this chapter is to compactly represent a large number of geolocated time series by some form of summaries so as to support and facilitate their visual exploration. Intuitively, given a set of geolocated time series, we want to provide summaries that express both their pattern across time as well as their corresponding spatial extent. These summaries may be constructed over the whole dataset, e.g., to provide an initial quick overview of the whole data, or over the results of a previous query, e.g., over those time series located inside a bounding box drawn by the user on the map. Specifically, we consider two types of sum-

maries, called *bundle summaries* and *tile map summaries*, respectively, which we describe next.

5.1.1 Bundle Summaries

This type of summary is composed of a set of k *"bundles"*, where each bundle comprises the following information:

- a cluster of similar time series in the temporal domain
- a set of Minimum Bounding Rectangles (MBRs) summarizing in the spatial domain the respective locations of those time series
- an integer indicating the number of objects located within each of the above MBRs.

To derive such bundles, we use the notion of *Minimum Bounding Time Series* (MBTS) introduced in Chapter 3. As a reminder, an MBTS bundles together a set of time series $\mathcal{T}$ using a pair of bounds that fully contain all of them. Hence, we can formulate the problem of summarizing a set of geolocated time series by means of a bundle summary as follows:

Problem 1 (Bundle Summary)**.** *Given a set $\mathcal{T}$ of geolocated time series, a spatial area q of interest, a number k of desired bundles, and a number of l MBRs per bundle, the problem is to efficiently compute a* bundle summary *that consists of a list of k tuples over the subset $\{T \in \mathcal{T} : within(T.loc, q)\}$ of time series located within area q. Each such tuple in the bundle summary has the following structure:*

$$R_b = \{\langle mbts, \{\langle mbr, cnt\rangle\}\rangle\} \tag{5.1}$$

where $mbts$ is a time series summary in the form of MBTS and is associated with a list of l MBRs; the count cnt of objects within each such mbr is also available. □

In Section 5.2, we show how the BTSR-tree index can be used to address this problem, i.e., to efficiently compute bundle summaries.

5.1.2 Tile Map Summaries

The bundles in the summary type introduced above are formed in a *data-driven* manner, as objects belonging to the same bundle should be similar (e.g., based on clustering). An alternative way to visually highlight spatio-temporal patterns in a large set of geolocated time series is through summaries that rely on a *fixed partitioning* of the time series domain. More specifically, the entire domain may be subdivided into adjacent, non-overlapping *tiles* (Figure 5.2b), so that each tile captures the portion of a time series falling within this tile. Simple aggregates (e.g., counts) of time series per tile can easily convey the distribution of values in the dataset. As shown in the example, the higher the concentration of data points within a tile, the darker its shade. More formally:

Definition 4 (Tile Map)**.** *Let the* time domain $[t_{min}, t_{max})$ *be divided into successive intervals of equal size τ, resulting into a subdivision $\{[t_{min}, t_{min}+\tau), [t_{min}+\tau, t_{min}+2\tau), \ldots, [t_{max}-$

$\tau, t_{max})\}$. *Subdivision of the* value domain $[v_{min}, v_{max})$ *is carried out using a finite number of* breakpoints $\{v_1, v_2, \ldots, v_h\}$ *where* $v_1 = v_{min}$*, and* $v_h = v_{max}$*, whereas the rest may be arbitrary values provided that* $v_1 < v_2 < \cdots < v_h$*. This subdivision yields disjoint, consecutive segments* $\{[v_{min}, v_1), [v_2, v_3), \ldots, [v_h, v_{max})\}$ *in the value axis. The resulting* tile map *is a matrix of tiles over both domains, so that tile* (i, j) *corresponds to time interval* $[t_{min} + i \cdot \tau, t_{min} + (i + 1) \cdot \tau)$*,* $i \in 0, \ldots, \lceil \frac{t_{max} - t_{min}}{\tau} \rceil$*, and to value segment* $[v_j, v_{j+1})$*,* $j \in 0, \ldots, h - 1$.

Essentially, such a tile map has a similar effect as *space-driven* partitionings like quadtrees or grid subdivisions [RSV02]. As in the case of spatial objects, a time series can be checked for containment within each tile. More specifically, a time series data point v_t at time t is contained in tile (i, j) if $t_{min} + i \cdot \tau \leq t < t_{min} + (i + 1) \cdot \tau$ and $v_j \leq v_t < v_{j+1}$. Once all time series are mapped into tiles, this matrix offers a summary of the entire dataset.

However, we may inspect specific portions of a tile map, by checking a group of neighboring tiles. This can be abstracted as a *timebox* [HS03] applied over both domains. Intuitively, such a timebox is a rectangle in the time series domain that fully contains a set of time series in the time and value range that it represents. Figure 5.1b depicts with green color the time series that are contained within a timebox, among a set of time series. More specifically:

Definition 5 (Timebox)**.** *A* timebox p *specifies a time interval* $[t, t')$ *and a value range* $[v, v')$ *in order to identify any qualifying time series. We denote as* $timebox(T, p)$ *once a time series* T *qualifies to this timebox* p *if* $\forall t_i \in [t, t'), v \leq T.v_i < v'$.

Overall, given a timebox p and a spatial area q of interest over a set of geolocated time series, we are interested in identifying:

- The tiles that summarize objects located within q and are also fully included within p, i.e., no data point of the time series falls outside p in the respective time interval.
- In addition, the summary provides also a set of k MBRs covering all qualifying time series; a counter measures the time series contained within each such MBR.

We can now formulate the problem of computing *tile map summaries* as follows:

Problem 2 (Tile Map Summary)**.** *Given a set* $\mathcal{T}$ *of geolocated time series, a timebox* p *and a spatial area* q *of interest, and a number* k *of desired MBRs, a* tile map summary *provides a list of* k *tuples over the subset* $\{T \in \mathcal{T} : within(T.loc, q) \wedge timebox(T, p)\}$ *of time series located within area* q *and qualifying to timebox* p*. Each such tuple in the tile map summary has the following structure:*

$$R_t = \{\langle tmap, \{\langle mbr, cnt \rangle\}\rangle\} \tag{5.2}$$

where $tmap$ *represents the tile map constructed over the qualifying time series and is associated with a list of* k *MBRs that outline their spatial extent; the count* cnt *of objects within each such* mbr *is also available.* □

In Section 5.3, we propose a technique that can efficiently construct tile map summaries by employing an extended variant of the iSAX index.

5.2 Computing Bundle Summaries

Intuitively, the first visualization method displays the bundle summaries for a spatial area of interest, as defined in Section 5.1.1. This may concern the currently visible area on a map, so a set of time series patterns and their respective spatial extents are computed and visualized. Using this process, a user can select the bundle of her preference and the proper spatial summary will appear on the map after acquiring the necessary MBRs from the BTSR-tree index. Whenever the user zooms in/out or pans around the map, the BTSR-tree is traversed, and the corresponding bundles, MBRs, and object counts are obtained to drive the visualization. In each case, the rectangle corresponding to the visible part of the map is used to feed a traversal algorithm that efficiently gathers the results. Next, we first outline the structure of the BTSR-tree index, and then we introduce a novel algrorithm for its traversal in order to compute the bundle summaries for a given area of interest.

5.2.1 Deriving Bundle Summaries from the BTSR-tree Index

To support the summaries required by the visualization method, we further extend the information stored in each node of the BTSR-tree index with the *count* of geolocated time series that are fully contained within each bundle. This is done bottom-up during insertion, while the index is traversed to calculate the bundles. At each leaf node, after the clustering, we propagate the number of members of each cluster to its parent, which in turn calculates its clusters and aggregates the counts it has received for each bundle's members. This procedure continues up to the root of the tree. We stress that this process concerns the building of the index and is carried out once geolocated time series are being inserted.

We now present our summarization technique for producing map-based visualizations of geolocated time series. The process is outlined in Algorithm 6. It takes as input the *input rectangle* q, i.e., the spatial area of interest for which the visualization is produced, the number k of bundles and the number l of MBRs per bundle to be generated. The process comprises three distinct steps. Initially, the BTSR-tree index is traversed to obtain the MBRs contained in the input rectangle, along with their bundles and the number of objects per bundle (Line 1). Next, k-means clustering is applied using the average time series per bundle as centroids (Line 2). Finally, the new bundles are calculated and the proper MBRs and corresponding object counts are assigned to each bundle (Line 3). Next, we describe each step in more detail.

Step 1: BTSR-tree Traversal. During this step, the BTSR-tree index is traversed, with the target being the fast provision of a predefined number k of geolocated time series bundles contained within the given area q, along with l MBRs where these bundles can be found and the total number of geolocated time series that reside within each MBR. All required information is stored within the nodes of the BTSR-tree, thus, when a node that is contained within the input rectangle is found, the relevant information is retrieved and added to the intermediate results, without any need to continue searching in its sub-tree. The output of this step is passed to the next phase of bundle clustering.

In more detail, the traversal is performed as follows. After initializing a queue with the root's children (Line 7), we iterate over it (Line 8) until it is empty. For each inner node's child N', we check whether its MBR is contained within the given input rectangle q (Lines 11–12). If so, its MBR, the time series bundles, and the number of objects per bundle are

added to the intermediate results (Line 13) as a *tuple* with the following components:

$$\langle mbr, \{(mbts_1, cnt_1), ..., (mbts_k, cnt_k)\}\rangle.$$

Each such tuple indicates the MBR of a node (mbr), consisting of the coordinates of the lower left and upper right point, as well as k pairs denoting the bundles of the node along with the corresponding number of objects per bundle. If the MBR is not contained in the input rectangle q, we check whether it overlaps with q and if so, we add the child node to the queue (Line 15). If not, this MBR is located outside the input rectangle, and thus we can skip searching this subtree. Once no more nodes are left to search, the intermediate results are finally returned (Line 16).

Step 2: Bundles Clustering. The aforementioned traversal method returns tuples, each containing the bundles residing in the input rectangle, the corresponding nodes' MBRs and the number of objects per bundle. Next, k-means clustering is executed on the average time series of each bundle. Line 2 of Algorithm 6 calls the clustering procedure. Initially, for each tuple (Line 20), we iterate over its bundles (Line 21) and generate a new tuple per bundle of the following format:

$$\langle T_{avg}, mbts, cnt, mbr\rangle$$

This new tuple contains an average time series, the bundle itself ($mbts$), the number cnt of objects enclosed in this bundle, and the MBR (mbr) this bundle belongs to (Line 23). The average time series T_{avg} is calculated by averaging the upper and lower bound of each bundle (Line 22), i.e., average value at each time point. The resulting collection of tuples is fed to the k-means algorithm (Line 24) in order to return the required number k of bundles to be created. This clustering generates a clustered collection of tuples using the calculated average time series. These results are then forwarded to step 3 (Line 25).

Step 3: Bundles Calculation and MBR Assignment. During this step, the clustered tuples received from step 2 are used to calculate the final bundles, the corresponding l MBRs and total number of objects per MBR are assigned to each bundle. The final bundles are calculated in a similar manner to the MBTS bundles during BTSR-tree construction. More specifically, at each time point, we obtain the maximum and minimum value among the corresponding upper and lower bounds for the bundles of each cluster (see Section 3.2.1). Then, we apply k-means clustering on each bundle's MBRs, obtaining a total of l new MBRs per bundle along with an aggregate count of the time series contained therein. The final result is then used for visualization.

Line 3 of Algorithm 6 calls the corresponding procedure. For each cluster of tuples received from step 2 (Line 28), we loop over its members (Line 31) and we use each tuple's bundle to update the upper and lower bounds (Line 32). Then, we apply k-means clustering on the cluster's MBRs, obtaining l new MBRs along with their counts (Line 33). Once the bounds and the corresponding list of MBRs for the current bundle have been calculated, we issue an aggregated tuple to the final result (Line 34). This tuple has the following components:

$$\langle mbts', \{(mbr_1, cnt_1), ..., (mbr_l, cnt_l)\}\rangle$$

where $mbts'$ is a resulting bundle, along with l MBRs associated with it. Each MBR is accompanied with the corresponding number of objects (i.e., raw time series) therein. The final result with all such tuples is then returned in order to generate the visualization (Line 36).

Algorithm 6: Bundles Summarization of Geolocated Time Series

Input: Input rectangle q; number k of bundles to be generated; number l of MBRs per bundle
Output: A list R_b containing tuples of bundles, MBRs, and object counts

```
1  R ← IndexTraversal(q)                                   // Step 1
2  R_c ← BundlesClustering(R, k)                           // Step 2
3  R_b ← BundlesCalculation(R_c, l)                        // Step 3
4  return R_b
5  Procedure IndexTraversal(q)
6      R ← ∅, Q ← Root.getChildren()
7      while Q ≠ ∅ do
8          N ← Q.getNext()
9          if N is not leaf then
10             foreach N' ∈ N.getChildren() do
11                 if q.contains(N'.mbr) then
12                     R ← R ∪ {⟨N'.mbr, {(N'.mbts, N'.cnt)}⟩}
13                 else if q.overlaps(N'.mbr) then
14                     Q ← Q ∪ N'.getChildren()
15     return R
16 Procedure BundlesClustering(R, k)
17     R_c ← ∅, C ← ∅
18     foreach t ∈ R do
19         foreach b ∈ t.mbts do
20             T_avg ← avg(b.up, b.lo)
21             C ← C ∪ {⟨T_avg, b, t.cnt(b), t.mbr⟩}
22     R_c ← kmeans(C.avg, k)
23     return R_c
24 Procedure BundlesCalculation(R_c, l)
25     R_b ← ∅
26     foreach Cl ∈ R_c.clusters do
27         mbts ← ∅, mbr ← ∅
28         foreach t ∈ Cl do
29             mbts ← updateMBTS(mbts, t.mbts)
30             {(mbr, cnt)} ← kmeans(l, t.mbr, t.cnt)
31         R_b ← R_b ∪ {⟨mbts, {(mbr, cnt)}⟩}
32     return R_b
```

5.3 Computing Tile Map Summaries

In the following, we present our second visualization method, which allows the user to draw one or more timeboxes on the time series domain. This triggers a traversal of our hybrid

geo-iSAX index to obtain the geolocated time series in the currently visible map area and also fully contained within these timeboxes. The result comes in the form of tiles, each spanning between two iSAX breakpoints, along with a count per tile indicating the number of time series whose SAX symbol resides within that tile. This count is used to generate the visualization in the form of tile map. A predefined number of MBRs and corresponding counts is also returned, generated by clustering the locations of the resulting geolocated time series and used for the spatial part of the visualization. Whenever the user zooms in/out or pans over the map, or whenever she draws a new timebox, the procedure is repeated, the index is traversed and the visualization is regenerated. Next, we first outline the original structure of the iSAX index and then we introduce its geo-iSAX variant, which enables evaluation of timebox queries and also maintenance of spatial information in its nodes. As we discuss next, those two extensions provide the necessary support for computing tile map summaries as specified in Section 5.1.2.

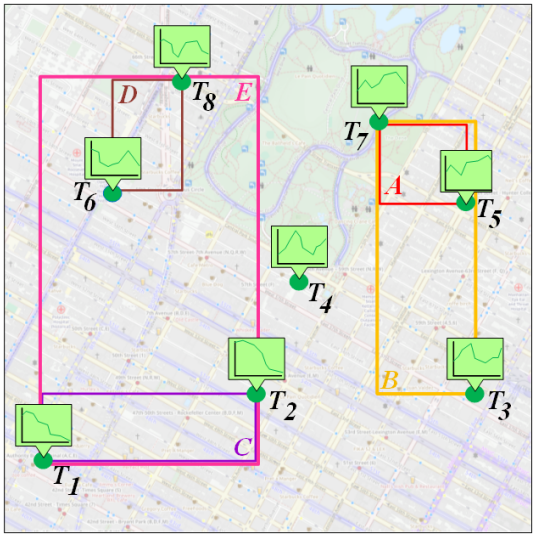
5.3.1 The geo-iSAX Index

We introduce geo-iSAX, a time series-first *hybrid* variant of the iSAX index that allows significant traversing speed-ups by pruning both on the spatial and time series domain. This is achieved by storing in each node of the tree, apart from the SAX word of the geolocated time series it contains, the MBR that they form. Initially, the time series part of the index is built, following the procedure described in Section 2.1. As a next step, similarly to the BTSR-tree index, we traverse the index in a bottom-up fashion, first generating the MBRs of the geolocated time series contained in the leaf nodes. As we go upwards, we update the MBR information of each visited inner node, using the MBRs of its children nodes, until we reach the root, whose MBR will contain the geolocated time series of the whole dataset. Figure 5.3b illustrates the structure of the geo-iSAX tree created over a sample dataset of geolocated time series.

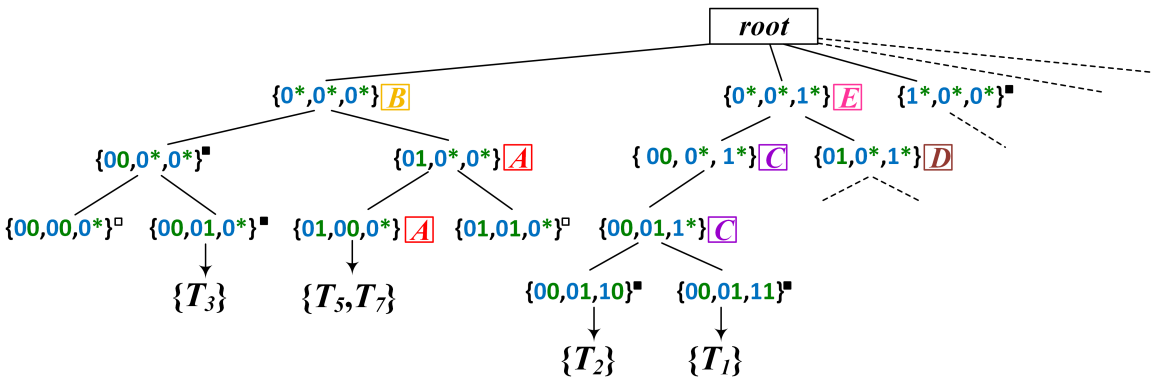
Due to the fact that the iSAX index is created to solely index time series data, the MBRs generated by our hybrid variant may be highly overlapping, mitigating the pruning potential while traversing the index. To alleviate the negative effects of the numerous overlaps, we introduce an alternative *splitting policy* for geo-iSAX, which attempts to minimize the overlapping area, while maintaining the total area covered by the MBRs that occur after a split at the lowest possible levels. Recall that the original iSAX index selects the split dimension (i.e., the segment of a node's word on which the split will occur) for a node using a round robin approach. Our method loops over all split dimensions and for each one, it calculates the SAX word that would occur upon splitting on it. Then, it generates the MBRs for that specific split using the location of the geolocated time series contained in the node to be split. For each split dimension, it computes the sum of the two new MBRs' intersection area and the total area that they cover. The selected split is the one that generates the smaller sum. Due to the rather small number of word segments in an iSAX index, this procedure does not incur high construction costs, only slightly affecting the overall index construction time.

5.3.2 Deriving Tile Map Summaries from the geo-iSAX Index

Next, we present our summarization approach for tile map visualization of geolocated time series, obtained using timeboxes. In order to maintain low latency even for large datasets, we



(a) Sample dataset with MBRs as maintained by geo-iSAX.

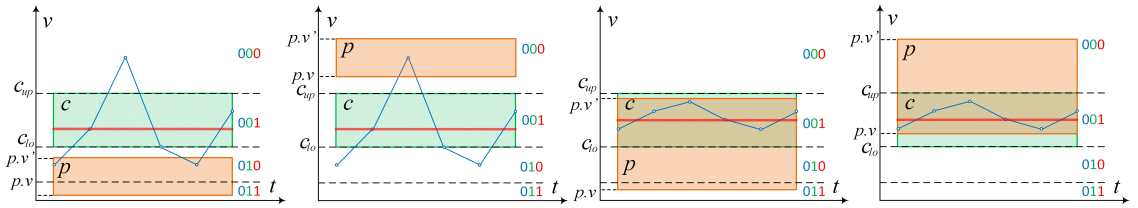


(b) The geo-iSAX index with letters indicating MBRs (depicted in 5.3a) attached to each node. Nodes with filled boxes indicate a degenerate MBR consisting of a single point; nodes with hollow boxes indicate no data. Subtrees under dash lines are not shown for brevity.

Figure 5.3: Sample dataset with MBRs and SAX representations of time series as maintained by the geo-iSAX index.

traverse geo-iSAX to obtain the resulting geolocated time series in a timely fashion. However, to avoid false negatives, we need to ensure that the pruned nodes do not contain any qualifying geolocated time series, as we discuss next.

Pruning Rule for Timebox Queries on geo-iSAX. When traversing the geo-iSAX index, we first evaluate whether its MBR satisfies the spatial constraint q (i.e., intersects with or is within q). If so, we need to check the timebox constraint p, i.e., whether the time series it contains *certainly* have data points that reside outside the given timebox. Consider the time series in Figure 5.4. Without loss of generality, we suppose a SAX word of length $w = 1$ and cardinality $b = 8$, i.e., $SAX(T, 1, 8) = \{010\}$. The solid red horizontal line is the PAA value that classifies this segment of the time series to the SAX value 010. Tile c defined by the two breakpoints that contain the PAA value is hown in green color, while the user-defined timebox p specifying a value range $[v, v')$ is depicted in orange.



(a) Outside below **(b)** Outside above **(c)** Intersecting below **(d)** Intersecting above

Figure 5.4: Cases where a timebox p is either outside or intersecting a given tile c.

As can be noticed in Figure 5.4a, the upper side of the timebox is below the lower side of tile c, i.e., $p.v' < c_{lo}$. In this case, we can safely prune a node that is described by this SAX word, because there is at least one data point outside the timebox, "pulling" the PAA upwards and within c. For the same reasons, we can prune a node whose breakpoint-defined area is completely below the given timebox, as depicted in Figure 5.4b. In this case, the lower side of the timebox is above the upper side of c ($p.v > c_{up}$), indicating that there is at least one data point outside the timebox "pulling" the PAA value downwards and within c.

On the other hand, in case the timebox intersects with tile c, we cannot safely prune the corresponding node. For example, in Figure 5.4c, $c_{lo} < p.v' < c_{up}$, so there may exist a time series that is fully contained in the timebox and thus be part of the result. The same stands for the example in Figure 5.4d, where $c_{lo} < p.v < c_{up}$. Finally, for the cases where timebox p fully contains tile c or vise-versa, it is trivially deduced that no pruning can apply.

We stress that the above observations only hold when in the time axis the given timebox is aligned to the segments of the SAX words in the index. For example, for a time series of length $n = 10$ and a word length of $w = 5$, the resulting segments will have length equal to $n/w = 2$. Thus, the timebox must be aligned to the data points t_0, t_2, t_4, t_6 and t_8.

Traversal Algorithm over geo-iSAX. Algorithm 7 outlines the process for producing the tile map visualizations of geolocated time series. The process takes as input a spatial rectangle q, a user-defined timebox p and the number k of MBRs that will be returned. It produces a list R_t containing k tuples of MBRs along with time series counts and the tile map, as defined in Eq. 5.1. For each inner node, the procedure checks whether its MBR intersects rectangle q and whether its SAX word might represent time series within timebox p. If a leaf node is reached and both constraints are met, we iterate over its raw geolocated time series and add the ones qualifying for the timebox to the final result R_t, after properly updating the breakpoint tile counts.

In more detail, the procedure takes place as follows. Starting from the root of the geo-iSAX index, we iterate over each node's children (Lines 5-6 in Algorithm 7). Next, we check whether the node to be evaluated is a leaf node (Line 8) and if it is not, we iterate over its children (Line 9) and check whether each one's MBR either is contained or intersects the spatial rectangle (Line 10). If so, we check whether its SAX word could represent time series within the timebox (Line 11) and if this is the case, we add it to the queue Q to be evaluated (Line 12). At this point, we should mention that, in order to avoid expensive calculations, we first perform the spatial check as it is less computationally expensive than the timebox check, avoiding the latter in case the node's MBR is outside the input rectangle. If the currently evaluated node is a leaf node (Line 13), we iterate over the geolocated time series that it contains (Line 14) and check whether it is fully contained within the user-defined timebox (Line 15).

Algorithm 7: Tile Map Summarization of Geolocated Time Series

Input: The input rectangle q; the timebox p; the number of MBRs to be generated k
Output: A list R_t containing a tile map and tuples of MBRs and object counts

```
1  R_t ← IndexTraversal(Root, q, p, k)
2  return R_t
3  Procedure IndexTraversal(Root, q, p, k)
4      R_t ← ∅, R ← ∅, tmap ← ∅, L ← ∅, Q ← Root.getChildren()
5      while Q ≠ ∅ do
6          N ← Q.getNext()
7          if N is not leaf then
8              foreach N' ∈ N.getChildren() do
9                  if q.contains(N'.mbr) ∨ q.overlaps(N'.mbr) then
10                     if timebox(N'.sax, p) then
11                         Q ← Q ∪ N'.getChildren()
12         else
13             foreach T ∈ N'.getChildren() do
14                 if timebox(T, p) then
15                     tmap ← updateCounts(tmap, T.sax)
16                     R ← R ∪ T
17             L ← L ∪ {R.mbr, |R|}
18     {⟨mbr, cnt⟩} ← kmeans(L, k)
19     R_t ← {⟨tmap, {⟨mbr, cnt⟩}⟩}
20     return R_t
21 Procedure timebox(X, p)
22     if isSax(X) then
23         foreach s ∈ {p.t_min, p.t_max} do
24             c ← breakpoints(X_s)
25             if p.v_min > c_up ∨ p.v_max < c_lo then
26                 return False
27         return True
28     else
29         foreach t ∈ {p.t_min, p.t_max} do
30             if X_t > p.v_max ∨ X_t < p.v_min then
31                 return False
32         return True
```

If so, we update the tile counts matrix $tmap$ (Line 16) and add the corresponding raw time

series to the set R (Line 17). Finally, we add to a list L the MBR of the set of qualifying time series along with its size (Line 18). Upon exiting the loop, we apply k-means clustering on the resulting centroids and obtain k tuples containing MBRs along with time series counts.

The procedure that checks whether a SAX word could represent time series that are fully contained within the given timebox is described in detail starting from Line 22 in Algorithm 7. It takes as input a timebox and a raw time series or SAX word that we want to check against the given timebox. We initially check whether the given X argument is a SAX word (Line 23). If it is, we itereate over all the segments that are contained (Line 22) in the time range defined by the timebox and for each segment we obtain the iSAX breakpoints that enclose it (Line 25). Afterwards, we check whether the lower side of the timebox is above the upper side of the tile defined by the obtained breakpoints, or vice versa, as depicted in Figure 5.4. If this is true, we return false. If this is not the case for any of the segments, the method returns true (Line 28). A similar procedure is followed for the case that argument X is a raw time series (Lines 29-33).

5.3.3 Experimental Evaluation

In this section, we evaluate the proposed visualization methods. We first describe our experimental setup including the datasets that we use in the evaluation. Next, we present illustrative visualizations over real-world geolocated time series, as well as scalability results against a synthetic dataset containing 4 million geolocated time series.

5.3.4 Experimental Setup

All experiments were conducted on a Dell PowerEdge M910 with 256 GB RAM and 4 Intel Xeon E7-4830 CPUs, each containing 8 cores clocked at 2.13GHz. We assume that all indices fit in memory, hence parameter selection for their construction was based on this assumption. We use the water and taxi real-world datasets also used for experimental evaluation in Section 4.4. To examine the scalability of our algorithms, we generated a synthetic dataset comprising 4 million geolocated time series by inflating the water consumption dataset. This was achieved by using the original time series as seeds and introducing some random variations in their location and pattern. We chose the water dataset so as to generate a more densely populated dataset (Alicante is a medium-sized city) to stress-test our summarization methods. In scalability tests, we also make use of randomly chosen subsets from this synthetic dataset. Table 5.1 lists a summary of the main characteristics for each dataset.

Table 5.1: Datasets used in the experiments.

Dataset	Area (km^2)	Number of time series	Length n of each time series
Water	114	822	168
Taxi	2,500	417,960	168
Synthetic	114	4,000,000	168

Table 5.2 lists the range of values for the parameters used in our tests concerning both methods; default values are shown in bold.

Table 5.2: Parameters tested in the experiments.

Parameter	*Values*
Dataset size	1000K, 2000K, 3000K, **4000K**
Map scale	1:50000, 1:25000, 1:20000, 1:15000, 1:10000, **1:5000**, 1:500

5.3.5 Evaluation of Bundle Summarization

We first present a detailed evaluation of our method concerning bundle summaries. Specifically, we present two visualization examples on two real-world datasets. Then, we evaluate the scalability of our method in terms of different map scales and dataset sizes.

5.3.5.1 Map Visualizations

The visualization for the bundle summary depicts the MBTS derived for the most representative patterns of time series at the currently visible area of the map. Once our summarization method returns the results, the corresponding MBRs contained in the current view and zoom level are drawn on the map, along with the number of the geolocated time series that belong to the selected bundle. This number is depicted using circles, colored green for small numbers, yellow for larger and red for more densely populated MBRs, thus easily conveying the local intensity of this pattern. The bundles are listed on the left of the map, using confidence bands to indicate their upper and lower bounds. The average time series of each bundle is also depicted. A user can scroll this list and select the bundle of their preference. Once a bundle is selected, the contents of the map are updated accordingly with the respective MBRs and aggregates.

Figure 5.5: Visualizing water consumption patterns in the city center of Alicante (map scale 1:5000).

Figure 5.5 shows an example of the bundle summary visualization using the water dataset, for $k = 5$ bundles and $l = 3$ MBRs. The depicted area is in the center of Alicante, in the most

densely populated zone of the city. In this example, `Bundle 4` is selected (indicated with a green colored frame) and the relevant MBRs are shown on the map (using red colored boxes). This indicates that inside each depicted MBR there exists a specific number of geolocated time series that have been clustered to the chosen bundle. As mentioned, each geolocated time series in this dataset represents hourly water consumption of a household across one week. Different consumption behaviors have been grouped together and a daily pattern for each bundle can be noticed which is due to the Circadian rhythmic way that people consume water [Asc65]. The rather large number of geolocated time series in the bundle, considering the zoom level and the extent of the MBRs, intuitively suggests that neighboring families tend to have similar water consumption behavior.

Figure 5.6 illustrates another example, this time using the taxi dataset in New York City, for $k = 5$ bundles and $l = 11$ MBRs. This dataset is significantly larger, and the zoom level selected in this example is lower (a larger geographic area is visible), hence the MBRs contain a larger number of time series. In this figure, we choose `Bundle 1`, which represents the rather quieter taxi dropoff zones in Manhattan, as the total number of dropoffs there is rarely over 60 during any hour of the week. In this example, there is also a clear daily routine in all bundles, with the dropoffs reaching a local maximum twice per day, suggesting the rush hours in New York City, when people commute to and from their work. In almost all bundles, the daily pattern is significantly different on Saturdays and Sundays, which confirms the intuition that during weekends people do not tend to commute in a routinely fashion. Overall, such visual representations of information digested from massive time series data can easily catch users' attention to important phenomena and ongoing trends, confirming the usefulness of our approach.

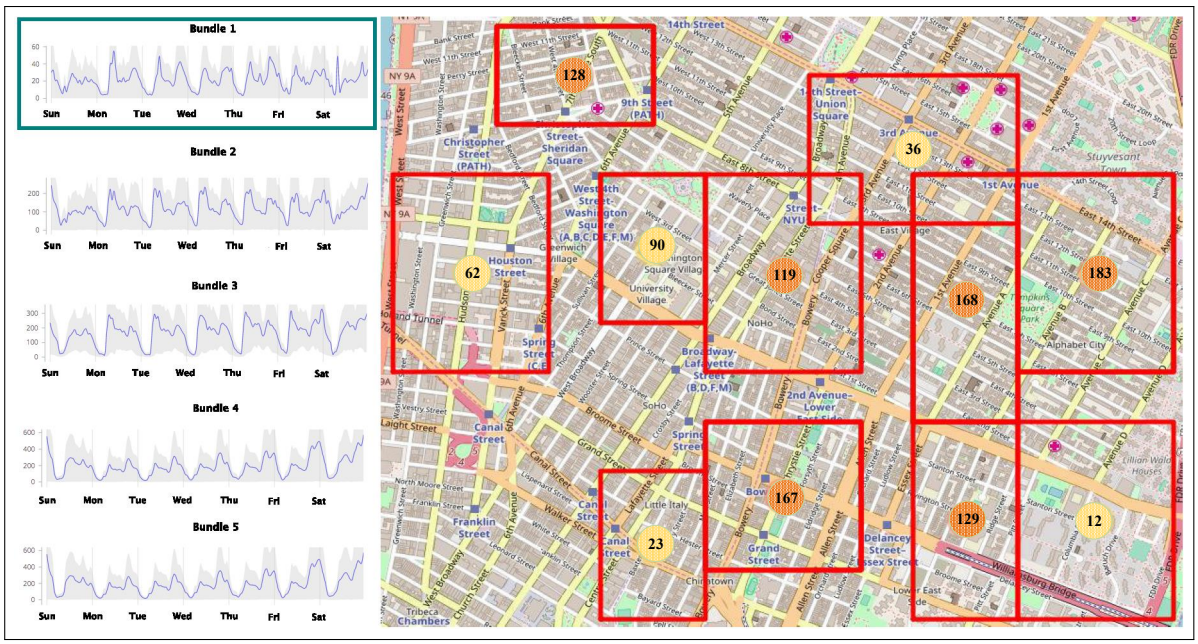


Figure 5.6: Visualization of taxi dropoff patterns in Manhattan, NYC (map scale 1:10000).

5.3.5.2 Performance Evaluation

Parameters. In preliminary tests, we fine-tuned parameters used against the synthetic dataset. Conclusively, for the scalability evaluation, we built the BTSR-tree index setting the minimum and maximum number of entries per node to $m = 750$ and $M = 2000$, respectively. Note that the index fits in memory, so such large parameter values do not have a negative impact on performance. For an evaluation of the BTSR-tree index under different parameter settings, please refer to [CSP+17]. Regarding the number of bundles, we set $k_0 = 5$ for its leaf nodes. The number k of bundles and the number l of MBRs per bundle for the traversal algorithm is set to be equal to the number of bundles at the leafs, i.e., $k = l = k_0 = 5$.

Baseline method for Detailed Bundle Summaries. In order to determine the trade-off between responsiveness and accuracy of our method (Section 5.2.1), we compare it with a baseline approach, which involves more *detailed summarization*. The latter utilizes the raw geolocated time series retrieved from the spatial filtering and generates the summaries by first applying k-means clustering on the time series domain to obtain the bundles, followed by another clustering in the spatial domain within each bundle to obtain its respective MBRs.

Results. We evaluate the performance of our approach on larger datasets in terms of response time for different zoom levels on the map using map scales, since zooming-in requires deeper traversal of the BTSR-tree index. The comparison is performed for subsets of the synthetic dataset of different size. We measure the *accuracy* both in spatial and time series domains, by calculating the mean Euclidean distance of each object located within the spatial area of interest from its closest bundle and MBR, respectively. Since this is intended as an interactive application, where the summarization method is triggered as soon as the user moves the map, *response times* must be adequately small. In our method, this is facilitated by the fact that the search along a path stops once it encounters a node whose MBR is contained in the actual map extent (rectangle). However, for the same reason, the responsiveness is expected to come at the expense of the summaries' level of detail, since the inner nodes contain coarser information regarding the time series they contain in their sub-trees.

Figure 5.7 depicts traversal costs for different map scales over the areas covered by the three datasets. More specifically, the water and synthetic datasets cover the area of the city of Alicante, Spain, whereas the taxi dataset the wider metropolitan area of New York City. Response time in all cases is equal or lower than one second. The synthetic dataset, due to its very high density is significantly slower than the rest, however still the results are obtained in less than a second. The response for the water dataset is almost instant due to its small size and very low density. Initially, in all cases, at the largest scale, the visible area of the map contains all the time series in the dataset, thus it only has to retrieve information from the root of the index. Then, as we zoom in, more nodes have to be visited, as the MBRs of the accessed nodes begin to overlap with the map rectangle and their children have to be retrieved. The worst case for the synthetic dataset is at scale 1:5000, which roughly corresponds to a large neighborhood of the city, where many time series are located. For the taxi dataset, the worst case is at 1:20000, which corresponds to the wider Manhattan area and then the response time gradually drops due to the lower dataset density. The number of nodes accessed in each case is proportional to the response times, ranging from one node (the root) in case of the smaller map scale (all city) up to 165 at scale of 1:5000 for the synthetic dataset, one up to 53

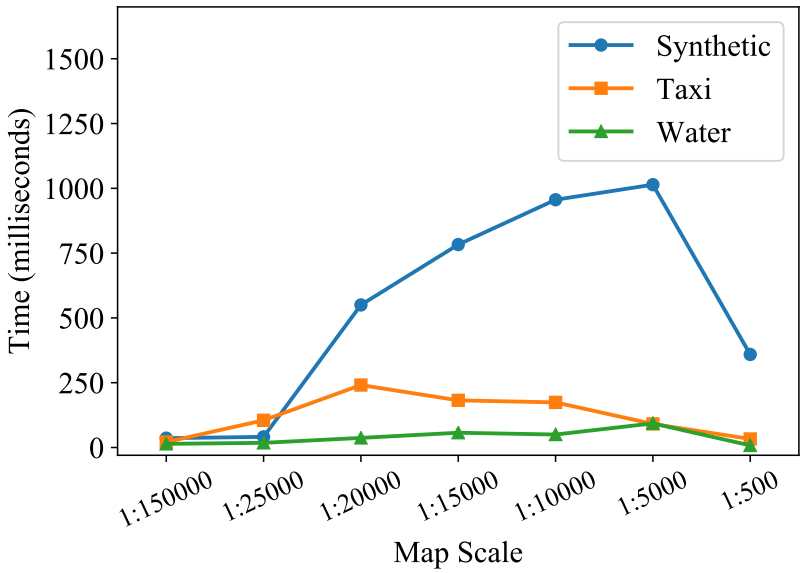
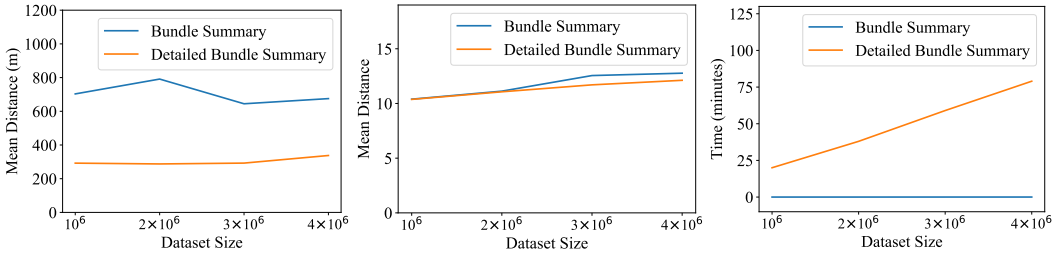


Figure 5.7: Execution time for different map scales.

for the taxi dataset and one up to 15 for the water dataset. Interestingly, fewer node accesses are required in all cases at the very large scale of 1:500, since the respective small map area overlaps with fewer nodes and most of the search space is pruned.



(a) Accuracy in spatial domain. **(b)** Accuracy in time domain. **(c)** Response time.

Figure 5.8: Assessment of bundle summarization.

Figure 5.8 depicts the accuracy and responsiveness comparison between the bundle summarization method and its detailed version. In the spatial domain (Figure5.8a), it is apparent that the mean Euclidean distance of the raw data from the closest MBR centers is close to two times larger for our bundle summary, indicating a loss in accuracy, as expected. The mean distance is not heavily affected by the dataset size at any case, indicating a rather stable summary quality, independent of the amount of the results. The case is quite different in the time series domain (Figure5.8b), where the mean distance of the raw time series from the closest bundle's average time series is only slightly larger for the bundle summary –especially for smaller datasets–, indicating a rather good summary quality. There is a slight worsening trend in both cases as the size of the dataset is increased, possibly due to the tendency of the summary to be more generic as the number of results increases, with a larger number

of BTSR-tree nodes taking part in the summary calculation. However, this loss in accuracy in both domains is largely compensated in terms of execution time (Figure 5.8c), with the bundle summary being close to one second in all cases, while the detailed approach is linearly slowed down as the dataset size increases, requiring more than an hour to generate the summary against 4 million objects. Overall, this test confirms that our proposed method for bundle summaries can offer a really large speedup in terms of response time with tolerable concessions in accuracy, even against a heavily dense synthetic dataset where a large number of time series are contained within a small area.

5.3.6 Evaluation of Tile Map Summarization

Next, we present an evaluation of our method for obtaining tile map summaries. We first demonstrate two examples of visual exploration via summarization over two real-world geolocated time series datasets. Then, we compare the scalability of our method with a baseline detailed summary in terms of different map scales and dataset sizes.

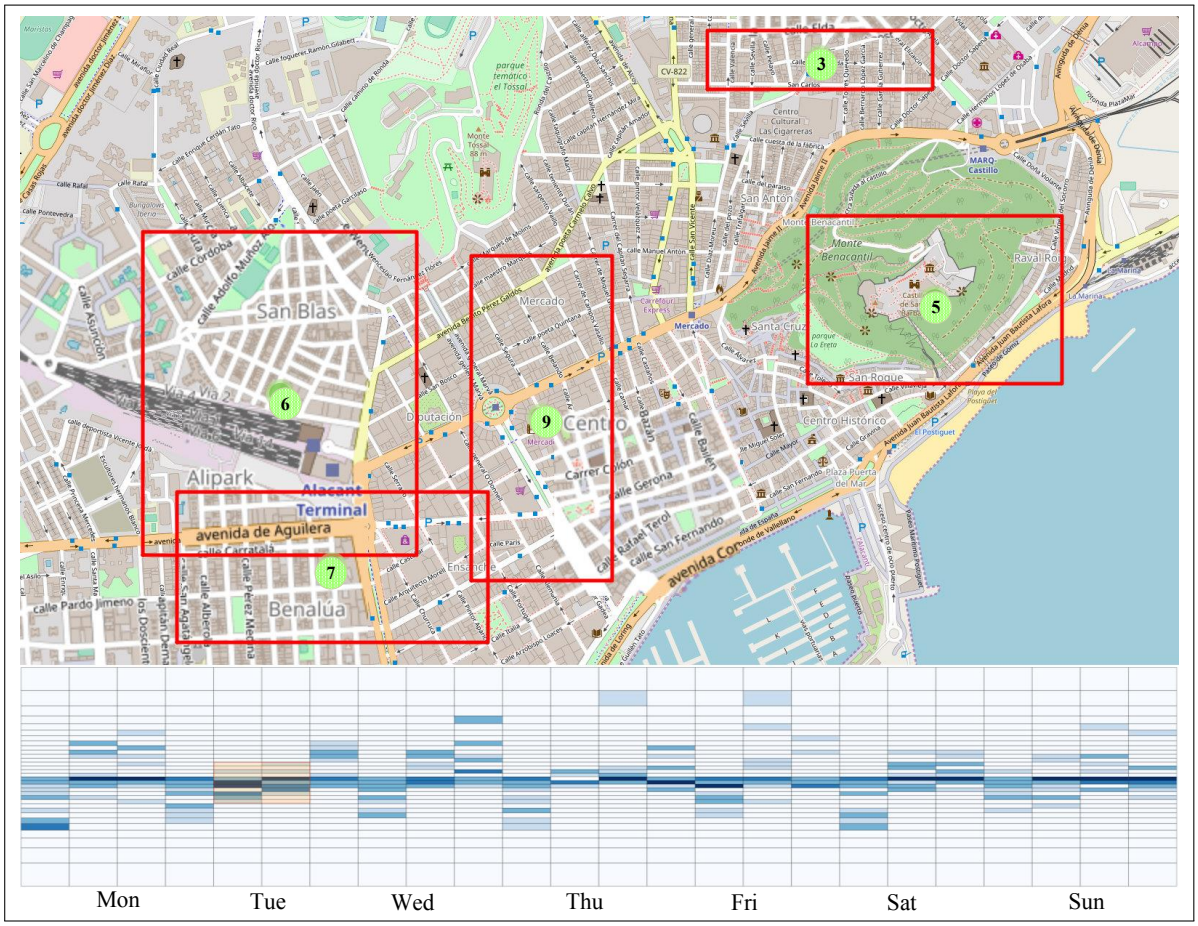


Figure 5.9: Visualization of water consumption tile map summary in the city center of Alicante (map scale 1:5000).

5.3.6.1 Map Visualizations

This visualization depicts a tile map summary of the geolocated time series that are contained within the currently visible part of the map. Whenever the user either moves around, zooms in and out the map, or draws a timebox on the time series domain, our summarization method is invoked to traverse the geo-iSAX index and return the MBRs and tile map that correspond to the visible area. Once the results are returned, they are drawn on the map and time series frames respectively. Similarly to the bundle summary and in order to present the local density, the number of geolocated time series contained within each MBR is depicted using circles colored green for small numbers, yellow for larger and red for more densely populated MBRs. The time series frame is located on the bottom of the map and is essentially a depiction of the returned tile map, using lighter shades of blue for less populated tiles and darker shades for the more dense ones. The timeboxes are drawn on this frame, allowing a selection of arbitrary ranges on the value axis, while, on the time axis, the selection is forced to be aligned with the iSAX segments.

An example of the tile map visualization is illustrated in Figure 5.9, for the central area of Alicante, with $k = 5$ MBRs, a SAX word length of $w = 24$ and a maximum cardinality of $b = 32$. An example timebox (depicted as an orange box) spans two iSAX segments (corresponding to Tuesdays on the time axis) and has a range of one standard deviation on the value axis. The associated MBRs are depicted using red colored boxes. In this example, and within the chosen timebox, there exists a rather small amount of time series, indicating that not many households tend to maintain a lower water consumption behavior during Tuesdays.

Another example of the tile map visualization, depicting an area in Manhattan, New York is illustrated in Figure 5.10, with $k = 5$ MBRs, a SAX word length of $w = 24$ and a maximum cardinality of $b = 64$. This time, the timebox spans three iSAX segments (representing Wednesdays) and a value range of one standard deviation. Since the selected range of taxi dropoffs is rather small, especially considering the fact that Manhattan is a busy area during the whole week, the depicted MBRs indicate that, during Wednesdays, there are quieter zones in these specific areas. Such information could be leveraged for searching a quieter neighborhood within the city.

5.3.6.2 Performance Evaluation

Parameters. Similarly to the bundle summary, we performed preliminary tests against the synthetic dataset to fine-tune the parameters. We built the geo-iSAX index setting the maximum number of entries per node to $M = 200$, a maximum cardinality of $b = 512$ and a default SAX word length of $w = 8$. We have set the number of resulting MBRs to $k = 5$ for its leaf nodes. Table 5.3 lists the range of values for the rest of the parameters.

Table 5.3: Parameters in tests for tile map summaries

Parameter	*Values*
Timebox time range ($\|p_t\|$)	1, **2**, 3, 4, 5
Timebox value range ($\|p_v\|$)	1σ, 1.5σ, $\mathbf{2}\sigma$, 2.5σ, 3σ

Baseline method for detailed summaries. Similarly to the bundle summary, we also compare with a more detailed baseline implementation in terms of accuracy and time required

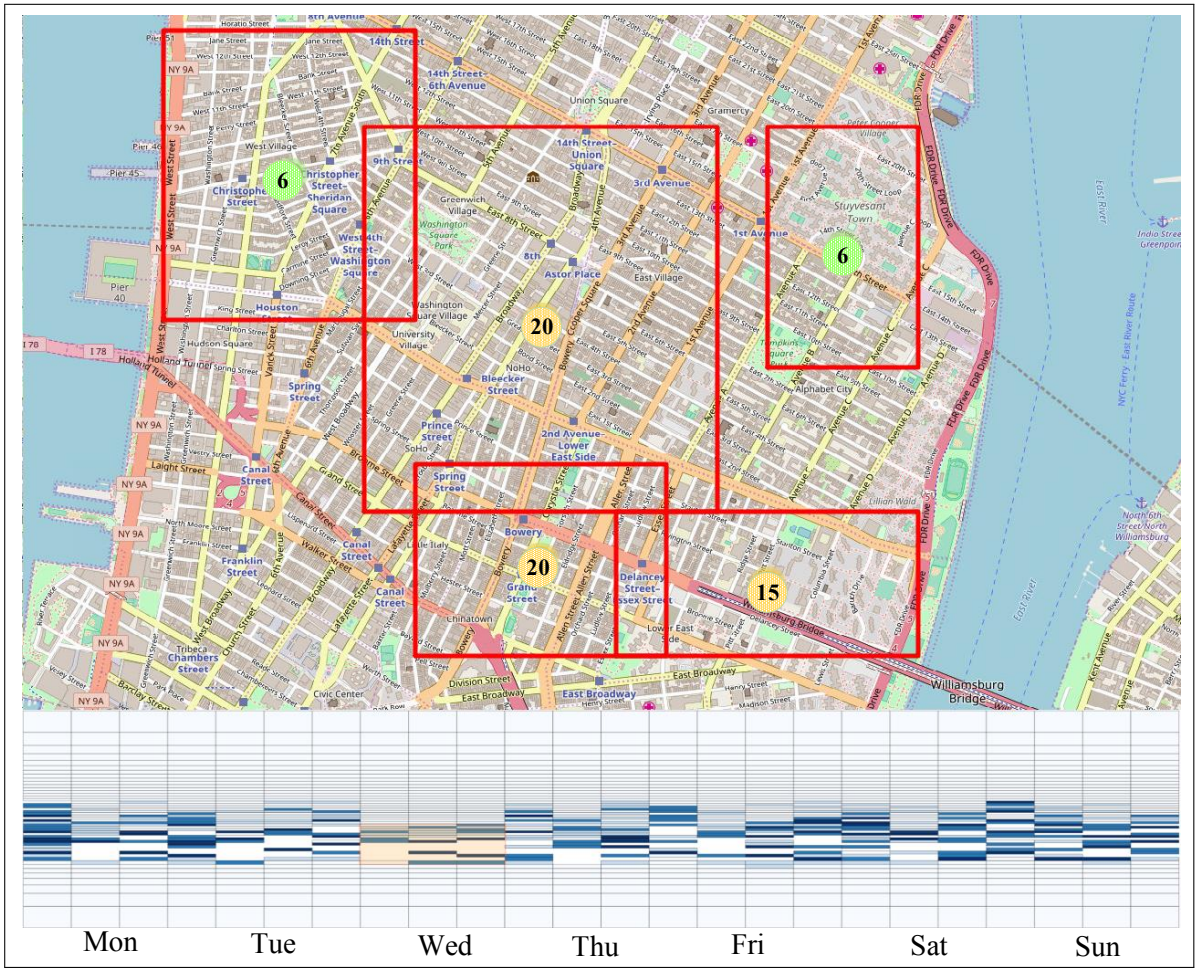


Figure 5.10: Visualizing taxi dropoff tile map in Manhattan, NYC (map scale 1:10000).

to fetch the results. The spatial part of the detailed summary is generated by performing k-means clustering on the filtered raw time series themselves and generating the resulting MBRs, instead of performing the clustering on each node's resulting MBRs. The tile map of the detailed summary has the exact same structure with the one of tile map summary. The difference among the two tile maps, since in the detailed summary the raw time series are used, lies in the way the count of each tile is augmented. Instead of increasing the count of the tiles where the SAX word's segments of a time series reside, we increase the count of the tiles where each time series point falls within.

Results. We evaluate the performance of the tile map summary for different zoom levels and timebox sizes. For the comparison between the two tile maps we use the *root mean squared error* (RMSE) on the difference among the counts between each pair of corresponding tiles. As mentioned, the detailed tile map is generated using the raw time series. This essentially generates tile maps with larger counts, since instead of only incrementing the count once for a SAX word's segment of a time series, it may be incremented multiple times, as each word's segment is derived from n/w time series points (n is the length of time series and w is the SAX word length). To compensate this, we divide the count of each tile of the

detailed summary with n/w.

Conclusively, the root mean squared error among two tile maps is calculated as follows:

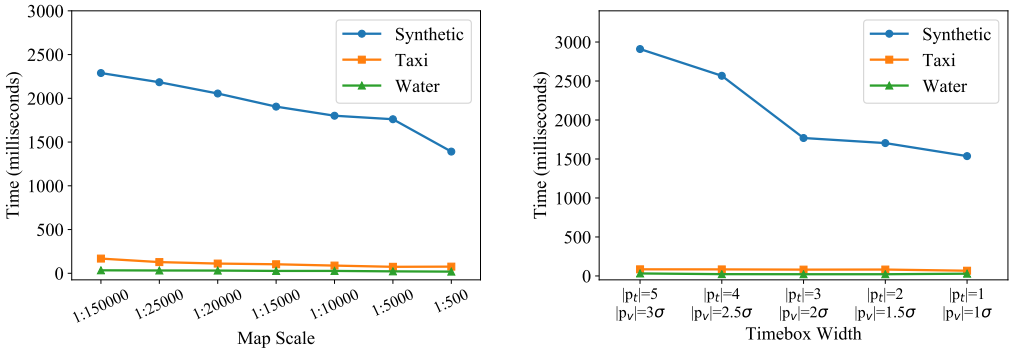
$$RMSE(tmap, tmap') = \sqrt{\frac{1}{w \times h} \sum_{i=1}^{w} \sum_{j=1}^{h} (c[i,j].cnt - \frac{c'[i,j].cnt}{n/w})^2} \tag{5.3}$$

where $tmap$ and $tmap'$ are respectively the tile maps from our method (Section 5.3.2) and the detailed summary, while $c[i,j].cnt$ and $c'[i,j].cnt$ are the corresponding counts of the (i,j) tile in both summaries and h is the number of y-axis breakpoints. For measuring the spatial accuracy, we follow the same rationale as in the evaluation of bundle summaries. Similarly, we evaluate the trade-off between responsiveness and accuracy of our approach. The detailed summary is expected to have a higher accuracy; however, since both spatial and time series summaries are calculated using the raw geolocated time series, we expect a trade-off in responsiveness, especially for larger datasets.

Figure 5.11a shows the traversal costs for different map scales for each of the three datasets. For the taxi and water datasets, the response time is almost instant due to their smaller sizes, ranging up to at most 100 milliseconds. For the case of the synthetic dataset and due to its high density, the response time is higher, starting from approximately 2.25 seconds for larger spatial areas of interest and falling down to 1.5 seconds as the map scale becomes larger (i.e., smaller areas). The rather higher response time for the tile map summary is due to the fact that in order to be calculated, the geo-iSAX's leaf nodes have to be accessed to obtain the locations and maximum cardinality SAX words to generate the summary. However, it should be noted that this is a rather worst case scenario, since the synthetic dataset was generated to be very highly dense as a stress test. The response time is not dramatically reduced for larger map scales, since geo-iSAX is a time series-first hybrid index and high overlapping of its MBRs is expected, so larger spatial areas of interest tend to intersect with the MBRs of many nodes, negatively impacting performance. As it is apparent, at a map scale of 1:500, the performance is more abruptly improved as the nodes of the index begin to be more aggressively pruned.

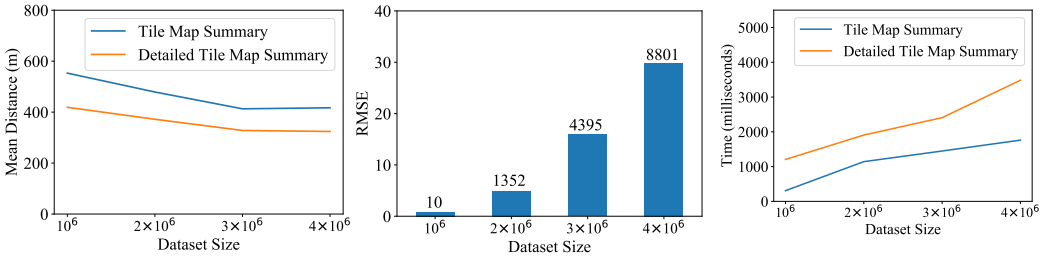
Similar results are also observed for different timebox sizes, as depicted in Figure 5.11b. The taxi and water datasets almost instantly generate the summaries along all timebox sizes. As a reminder, the timebox size is measured in terms of number of SAX segments selected in the time axis and standard deviation range selected in the value axis. In the figure, time range $|p_t|$ denotes the number of SAX segments, whereas value range $|p_v|$ expresses the standard deviation range selected for each timebox. For larger timebox sizes, the index traversal is slower as expected, as more nodes tend to satisfy the rather loose constraints. For the same reasons, the response time is improved up to around 1500 milliseconds as the timebox size gets smaller.

Figure 5.12 demonstrates the trade-off between accuracy and responsiveness by comparing the proposed tile map summary and its detailed implementation. As it can be easily observed, the difference between the spatial mean Euclidean distance among the raw data and their closest MBRs (Figure5.12a) is between 150 and 200 meters for all dataset sizes. In both cases, it is rather larger for smaller datasets, slowly diminishing as their size gets larger, indicating an improvement of both summaries for larger data. It is worth noting that the difference from the baseline implementation also seems to be slightly reduced for larger datasets. Figure 5.12b illustrates the RMSE between the two tile maps using Equation 3.2.



(a) Response time for different map scales. **(b)** Response time for different timebox sizes.

Figure 5.11: Response time for different map scales and timebox sizes.



(a) Accuracy in spatial domain. **(b)** Accuracy in time series domain. **(c)** Response time.

Figure 5.12: Assessment of tile map summarization.

The number at the top of each bar is the count of results returned using the same constraints (spatial rectangle, timebox) over each dataset. It is apparent that the quality of the tile map summary worsens for a larger number of participating time series, increasingly diverging from the detailed version. Still, the loss in both spatial and tile map accuracy is compensated in terms of response time as depicted in Figure 5.12c. The detailed implementation requires up to twice the time to generate the summary for all dataset sizes, while its overhead compared to the tile map summary is increasing with larger datasets. Consequently, there is a clear trade-off between accuracy and response time in both domains, allowing an increase in responsiveness without much sacrifice in accuracy.

5.4 Summary

In this chapter, we introduced methods for map-based visual exploration over large geolocated time series data. To that end, we proposed two summarization approaches over geolocated time series, which allow a visual analytics application to retrieve the required information. The results can be displayed on a map, depicting the spatial distribution of the data in the form of MBRs for both approaches. Each approach also provides a time series summary, via time series bundles or tile maps respectively. To speed up the retrieval of the results, we

employ two hybrid indexing techniques that allow pruning in both the spatial and the time series domains. Our experiments on a large-scale synthetic dataset indicated that the visualizations can be rendered fast, enabling efficient exploration in map-based applications; in the worst case, response time is up to a couple of seconds. Additionally, we examined indicative demonstrations of the visualizations generated from two real-world datasets in different application domains, confirming their helpfulness in jointly exploring both the time series themselves as well as their geographic distribution.

In the next chapter, we deal tackle the problem of discovering pairs or bundles (groups) of co-evolving time series, i.e., time series that contain observation values at the same timestamps all along their duration. Additionally, we focus on local similarity search on geolocated time series, using a modified version of our BTSR-tree index.

Chapter 6

Local Similarity Search

In this chapter, we introduce the measure of *local similarity*, that can be applied on co-evolving (i.e., time aligned) time series. Two co-evolving time series are locally similar if the pairwise distance of their values per timestamp does not exceed a given threshold during a time interval, that lasts at least a pre-defined number of consecutive timestamps. Based on this new similarity measure, we introduce two novel approaches for efficiently detecting all possible pairs and *bundles* (groups) of time series that are locally similar within a given dataset. Additionally, we utilize our hybrid BTSR-tree index to answer several hybrid queries on geolocated time series based on local similarity and introduce an extension to BTSR-tree, named $\mathcal{S}$BTSR-tree that significantly speeds up the process.

The bundle discovery problem we address in this Thesis resembles the problem of *flock discovery in moving objects*, where the goal is to identify sufficiently large groups of objects that move close to each other over a sufficiently long period of time. However, to the best of our knowledge, ours is the first work to address the problems of locally similar pair and bundle discovery over co-evolving time series.

Local Pair and Bundle Discovery. Discovering such pairs and bundles is useful in various applications. For instance, public utility companies employ smart meters to collect time series measuring consumption per household (e.g., for water or electricity). Identifying such bundles of time series (i.e., a number of similar subsequences over certain time intervals) can reveal similar patterns of consumption among users, allowing for more personalized billing schemes. In finance, examining time series of stock prices can identify pairs or bundles of stocks trending similarly at competitive prices over some trading period, hence offering precious insight for possible future investments.

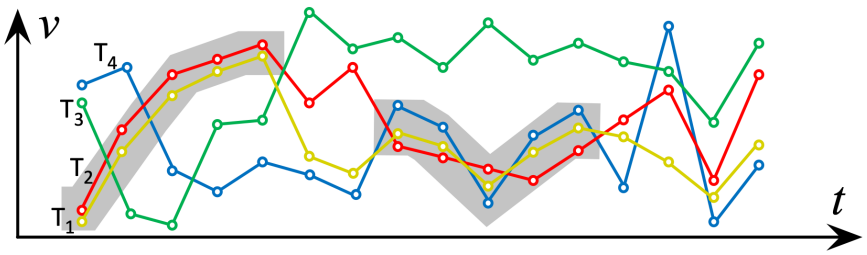


Figure 6.1: A pair and a bundle of locally similar time series.

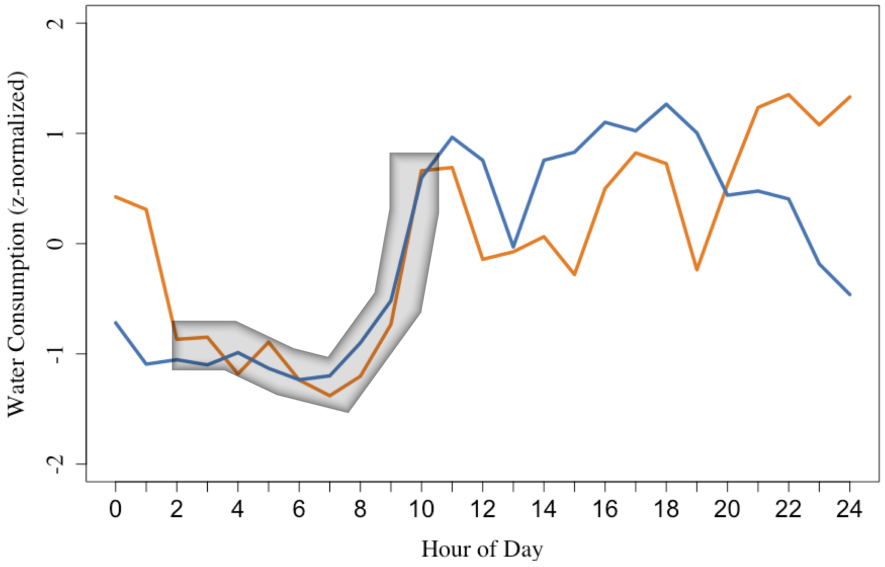


Figure 6.2: Pair of locally (but not globally) similar time series.

Figure 6.1 illustrates an example comprising four time series depicted with different colors. We observe that from timestamp 1 to 5 the values of T_1 and T_2 are very close to each other, thus forming a locally similar pair. Similarly, from timestamp 8 to 12, the values of T_1, T_2 and T_4 are close to each other, forming a bundle with three members. Note that values in each qualifying subsequence may fluctuate along a bundle as long as they remain close to the respective values per timestamp of the other members in that bundle.

A real-world example is depicted in Figure 6.2. These two time series represent per-hour average water consumption during a day of the week for two different households. We can observe that their respective values per timestamp (at granularity of hours, in this example) are very close to each other during a certain time period (hours 2-11), but are farther apart in the rest. Hence, an algorithm that measures the global similarity between two time series might not consider this pair as similar; however, the subsequences inside the gray strip are clearly pairwise similar, and might indicate an interesting pattern. Identifying such local similarities within a sufficiently long time interval is our focus in this chapter.

Furthermore, Figure 6.3 depicts several bundles of locally similar time series detected by our algorithms in a real-world dataset containing smart water meter measurements. The detected bundles represent different per-hour average water consumption patterns during a week. There is a wider pattern detected among 6 households during the first 30 hours of the week indicating reduced consumption (probably no permanent residence). The orange and yellow patterns indicate different morning routines during the third and fourth day of the week. The green and purple patterns represent a reduction in consumption during the late hours of the fourth and sixth day, respectively, with some intermediate consumption taking place during the night. Finally, the shorter red and light blue bundles suggest different evening patterns for two other days (respectively, decreasing and increasing consumption).

Discovering all possible pairs and bundles of locally similar time series, along with the corresponding subsequences, within large sets is a computationally expensive process. To find matches, a filter-verification technique can be applied. At each timestamp, the *filtering* step can discover candidate pairs or groups having values close to each other; then, the

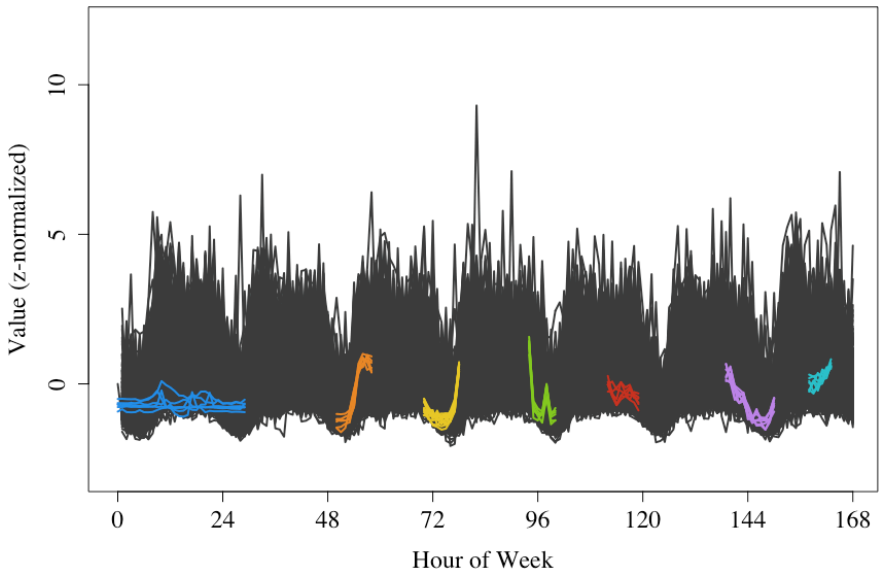


Figure 6.3: Bundles of locally similar time series.

verification step is invoked to determine whether each such candidate satisfies the required conditions, essentially whether this match occurs throughout a sufficiently large time interval. However, both the filtering and the verification steps are expensive. The computational cost becomes especially high for the case of bundle discovery, as it has to examine all possible subsets of locally similar time series that could form a bundle. Hence, such an exhaustive search is prohibitive when the number and/or the length of the time series is large.

We employ a value discretization approach that divides the value axis in ranges equal to the value difference threshold ϵ, in order to reduce the number of candidate pairs or bundles that need to be checked per timestamp. Leveraging this, we first propose two *sweep line* scan algorithms, for pair and bundle discovery respectively, which operate according to the aforementioned filter-verification strategy. However, this process still incurs an excessive amount of comparisons, as it needs to scan all values at every timestamp. To overcome this, we introduce a more aggressive filtering that only checks at selected *checkpoints* across time, but ensuring that no false negatives ever occur. This approach incurs significant savings in computation cost, as we only need to examine candidate matches on those checkpoints only instead of all timestamps. To further reduce the number of examined candidates, we propose a strategy that judiciously places these checkpoints across the time axis in a more efficient manner. We then exploit these optimizations introducing two more efficient algorithms that significantly reduce the execution cost for both pair and bundle discovery.

The bundle discovery problem we address in this chapter resembles the problem of *flock discovery in moving objects*, where the goal is to identify sufficiently large groups of objects that move close to each other over a sufficiently long period of time [GvK06, BGHW08, VBT09, TVK15]. In fact, the baseline algorithm we describe can be viewed as an adaptation of the algorithm presented in [VBT09]. However, to the best of our knowledge, ours is the first work to address the problems of locally similar pair and bundle discovery over co-evolving time series.

Local Similarity Search on Geolocated Time Series. Our approach for hybrid search over geolocated time series (see 3) using the BTSR-tree supports only *global* time series similarity, i.e., similarity measured across the entire length of time series. Specifically, as in other works in this area [EZPB18, LKWL07, CPSK10, CSP+14], the distance between two time series is measured by aggregating the pairwise Euclidean distance of their respective values across the entire sequences. However, in many cases, more fine-grained trends and patterns may exist, which are missed under this global similarity measure. For example, consider two time series representing the hourly energy consumption of two nearby buildings over a week, and assume that the two buildings exhibit a similar consumption pattern during working days but a different one in weekends. A query imposing a similarity threshold over the entire week would fail to identify these two geolocated time series as similar. However, it may be useful to discover that there is a period of up to 5 days during which these two time series are actually similar.

Motivated by this observation, in this work we extend our previous approach on hybrid queries over geolocated time series to support *local similarity* of time series, thus allowing more flexible and fine-grained queries and analyses. The *local similarity score* between two time series T_i and T_j is defined as the maximum number of consecutive timestamps during which the respective values of T_i and T_j do not differ by more than a user-specified threshold ϵ. Notice that, compared to global similarity, this condition is more relaxed, in the sense that it is applied to subsequences of length lower than T_i and T_j, but at the same time stricter, in the sense that the threshold ϵ is required to be satisfied at each individual timestamp during the selected period rather than on the aggregate distance over all timestamps.

Combining this local similarity constraint with a filter on *spatial distance* leads to a novel set of hybrid queries. Figure 6.4 shows an example with a query time series T_q searching over a set of time series $T_1, \ldots, T_9$ for those within radius ρ from its location and also locally similar to T_q. In particular, with respect to a given ϵ, results should also be locally similar to T_q for at least 5 consecutive timestamps. Qualifying results include T_2 with local similarity score $\sigma_2 = 5$ (bottom chart), and T_7 with $\sigma_7 = 7$ (top chart).

It turns out that such hybrid queries involving local similarity can still be evaluated using the BTSR-tree index. We first present a baseline method employing a sweep-line algorithm to check for local similarity, and then describe how this can be optimized by using appropriately placed *checkpoints*, based on the local similarity score threshold specified by the query, in order to skip unnecessary comparisons. Despite the fact that this saves some computations, the resulting time savings are relatively small, since the number of index nodes that need to be probed is not essentially reduced. To overcome this problem, we introduce an improvement to the BTSR-tree index, which is based on temporally segmenting the time series bounds within each node and deriving tighter bounds per segment. Once the time series bounds in each node become more fine-grained, pruning the search space for local similarity queries proves much more effective.

The rest of this chapter is organized as follows. For pair and bundle discovery, Section 6.1.1 describes the problems. Sections 6.1.2 and 6.1.3 introduce our algorithms and Section 6.1.4 reports our experimental results. For local similarity search on geolocated time series, Section 6.2.1 formally defines the problem. Section 6.2.2 presents how query evaluation under local time series similarity can be executed using the BTSR-tree. Then, Section 6.2.4 presents the enhanced $\mathcal{S}$BTSR-tree. Section 6.2.5 reports our experimental results. Finally, Section 6.3 concludes this chapter.

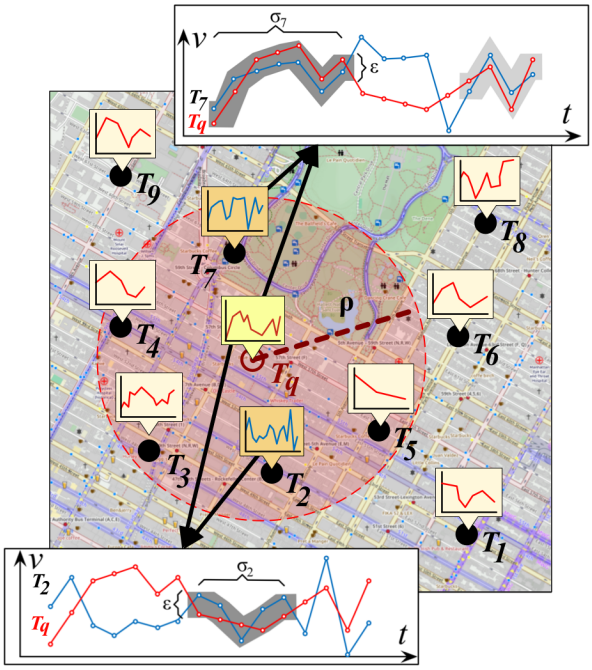


Figure 6.4: Retrieving geolocated time series based on spatial distance and local similarity.

6.1 Local Pair and Bundle Discovery

6.1.1 Problem Definition

We consider a set of *co-evolving* time series, so all time series are *time-aligned* and each series has a value at each of the k timestamps. Given a set of such co-evolving time series, our goal is to find *pairs* of time series that have similar values locally over some time intervals of significant duration. More specifically:

Definition 6 (Locally Similar Time Series)**.** *Two co-evolving time series $_i$ and $_j$ are locally similar if there exists a time interval I spanning at least δ consecutive timestamps such that at every timestamp in I their corresponding values do not differ by more than a given threshold ϵ, i.e., $\forall t \in I, |T_i.v_t - T_j.v_t| \leq \epsilon$.*

Note that threshold ϵ expresses the maximum tolerable deviation per timestamp between two time series, so it actually concerns the absolute difference of their corresponding values. We wish to find all such pairs of time series, so the problem is actually a *self-join* over the dataset, specifying as join criteria the distance threshold ϵ and the minimum time duration δ of qualifying pairs. More formally:

Problem 3 (Pair Discovery over Time Series)**.** *Given a set of r co-evolving time series $\mathcal{T} = \{T_1, T_2, ..., T_r\}$ of equal duration n, a distance threshold $\epsilon > 0$, and a time duration threshold $\delta > 1$ timestamps, $(\delta \in N)$, retrieve all pairs $\{T_i, T_j\}, 1 \leq i < j \leq r$ of locally similar time series along with the corresponding time intervals.*

For example, in Figure 6.5, the detected pairs for specified ϵ and δ would be the locally similar time series within grey ribbons. Since two time series might be locally similar in more

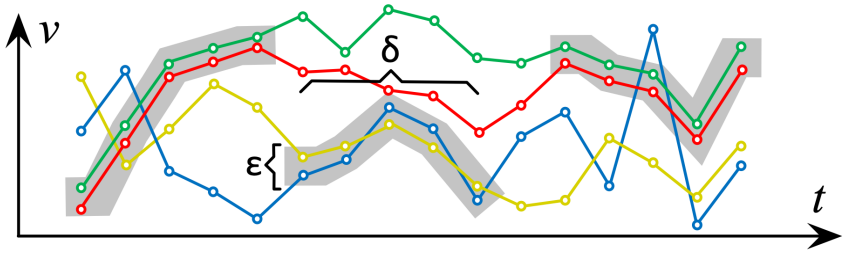


Figure 6.5: Pair discovery over a set of time series.

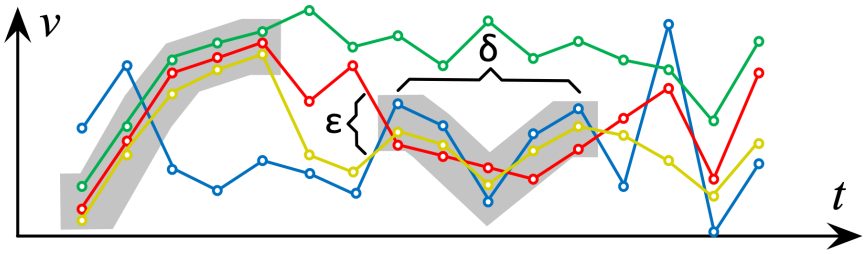


Figure 6.6: Bundle discovery over a set of time series.

than one intervals, their matching subsequences are considered as two different pairs, one for each interval. For instance, in Figure 6.5 the green and red time series yield two matching pairs in different time intervals.

The above problem can be extended to the detection of groups, called *bundles*, of co-evolving time series. Each such bundle of time series contains at least a pre-defined number $\mu > 2$ of members, which are pairwise locally similar to each other over a time interval of sufficient duration. This problem can be formulated as follows:

Problem 4 (Local Bundle Discovery over Time Series)**.** *Given a set of co-evolving time series* $\mathcal{T} = \{T_1, T_2, ..., T_r\}$ *of equal length* n*, a minimum bundle size* $\mu > 2, (\mu \in \mathbb{N})$*, a maximum value difference* $\epsilon > 0$*, and a minimum time duration* $\delta > 1$ *timestamps,* $(\delta \in \mathbb{N})$*, retrieve all groups* $\mathcal{G}$ *of time series such that:*

- *Each group* $G \in \mathcal{G}$ *contains at least* μ *time series.*
- *Within each group* $G \in \mathcal{G}$*, all pairs of time series are locally similar with respect to* ϵ *and* δ*.*
- *Each group* $G \in \mathcal{G}$ *is maximal, i.e., there is no other group* $G' \supseteq G$ *that also forms a bundle for the same time interval.*

An illustration of the above problem is shown in Figure 6.6. Each grey band covers the subsequences of at least $\mu = 3$ time series that constitute a bundle. These subsequences are pairwise locally similar for a specified distance ϵ and duration δ.

6.1.2 Pair Discovery

In this Section, we propose two solutions for the pair discovery problem. The first (Section 6.1.2.2) is a baseline algorithm that uses a sweep line to scan the co-evolving time series

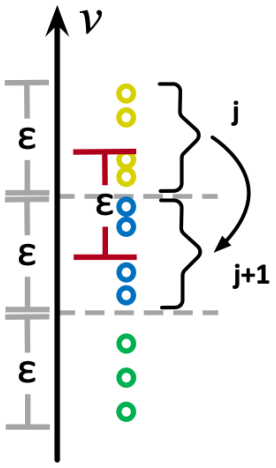


Figure 6.7: Discretization of time series values at timestamp t.

throughout their duration, while validating and keeping all the pairs that satisfy the given constraints employing a value discretization scheme per timestamp (Section 6.1.2.1). The second method (Section 6.1.2.4) employs an optimization that reduces the number of pairs to consider by judiciously probing candidates at selected timestamp values (referred to as *checkpoints*, Section 6.1.2.3). This significantly prunes the search space without missing any qualifying results.

6.1.2.1 Value Discretization

To reduce the candidate pairs that need be checked at each timestamp t, we discretize the values of all time series at t in *bins*, i.e., several consecutive value ranges, each one of size ϵ. Time series with values within the same bin at timestamp t form candidate pairs, but we also need to check adjacent bins for additional candidate pairs whose values differ by at most ϵ. Time series having values at non-adjacent bins are certainly farther than ϵ at that specific timestamp t, so we can avoid these checks.

To detect all candidate pairs and avoid cross-checking in every two adjacent bins we consider a *value range* of size ϵ, whose upper endpoint coincides with each value under consideration at time t. Then, all values of time series contained within this range, form candidate pairs (see Figure 6.7). Obviously, values contained in the same bin j can form candidate pairs. Then, we can cross-check each value in bin j with values in bin $j+1$ for additional candidates with value difference at most ϵ, as indicated with the red (right) range.

At each timestamp t, the process of finding all the pairs consists of: (1) *Filtering* - Search among time series values in adjacent bins to detect candidate pairs using the aforementioned search method. (2) *Verification* - For each candidate pair, check similarity of their respective values at successive timestamps as long as this pair still qualifies to the matching conditions (or the end of time series data is reached). This step resembles to a "horizontal expansion" along the time axis in an attempt to eagerly verify and report pairs.

Algorithm 8: Sweep line scan pair discovery

Input: Set $\mathcal{T}$ of co-evolving time series of length n
Parameters: Threshold ϵ, min duration δ
Output: List P with all locally similar pairs of time series

```
1  B ← CreateBins(T)
2  P ← DiscoverPairs(∅, B, {0, ..., n}, ϵ, δ)
3  return P
4  Procedure DiscoverPairs(P, B, Ω, ϵ, δ)
5      foreach t ∈ ω do
6          foreach z = 0 → B.size do
7              T' ← B_z^t ∪ B_{z+1}^t
8              foreach T ∈ T' do
9                  P^t ← getAdjacentPairs(T.v_t, ϵ)
10                 foreach p ∈ P^t do
11                     if p ∉ P then
12                         p ← VerifyPair(p, t, n, ϵ)
13                         if p.end − p.start ≥ δ then
14                             P ← P ∪ p
15     return P
16 Procedure VerifyPair(p, t, n, ϵ)
17     foreach t' = t + 1 → n do
18         if |p.T1.v_t' − p.T2.v_t'| > ϵ then
19             break
20     p.start ← t
21     p.end ← t'
22     return p
```

6.1.2.2 Pair Discovery Using Sweep Line

A baseline method for pair discovery over a set of time series is to check all the candidate pairs formed at each timestamp, and verify whether the minimum duration constraint δ is satisfied. Algorithm 8 describes this procedure. Pair discovery (Line 5) considers a time duration Ω (as a set of consecutive timestamps) to check for results. Initially $\Omega = \{0, \ldots, n\}$, where n is the total duration of all time series data. For each timestamp t, we obtain only the subset of time series whose values are contained in two adjacent bins (Line 7). Based on these values at t, we obtain all candidate pairs with respect to the threshold ϵ (Line 9). For each such pair, if it is not already part of the resulting pairs at that specific timestamp t, we *verify* it by first expanding it horizontally (Lines 10-12) and checking whether this pair meets the duration constraint δ (Line 13) along subsequent timestamps. If so, we add it to the reported results (Line 14).

For the *horizontal expansion*, we iterate over the subsequent timestamps (after the current one – Line 17) and stop when ϵ is violated (Lines 18–19). Then, we mark the start of this pair with the current timestamp t, whereas its end is marked by the timestamp at which ϵ is crossed, and we return this pair (Lines 20-22).

However, searching over all timestamps in such an exhaustive manner can be expensive, particularly for long time series. Next, we present an optimization that identifies candidate

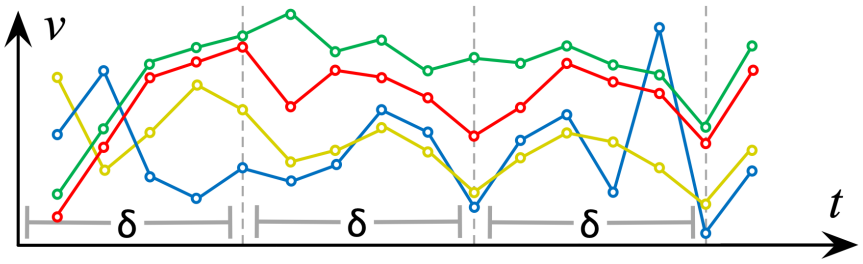
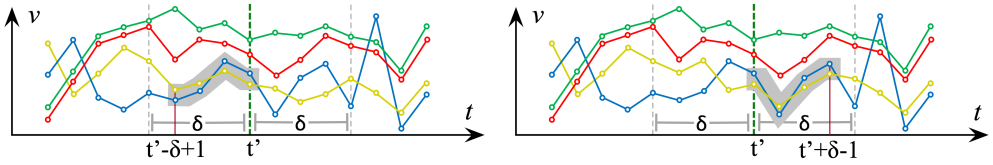


Figure 6.8: Checkpoints placed every δ timestamps.



(a) Pair with a starting point before t'. **(b)** Pair with an ending point after t'.

Figure 6.9: A qualifying pair will be detected on a checkpoint.

pairs at selected timestamps only, so that only those pairs require verification.

6.1.2.3 Optimized Filtering at Checkpoints

To prune the search space, we consider *checkpoints* along the time axis, so that searching for candidate pairs will be performed at these specific timestamps only. If the temporal span between two successive checkpoints does not exceed the minimal duration threshold δ, we can ensure no false negatives, since any qualifying pair starting at an intermediate timestamp between two checkpoints will surely be detected at least on the second one. Figure 6.8 shows an example for a set of time series (checkpoints placed every $\delta = 5$ timestamps).

Assume a set of checkpoints placed at time interval δ from each other, as depicted in Figure 6.9a. Let a checkpoint at timestamp t' and a qualifying pair of duration δ starting at timestamp $t' - \delta + 1$. This pair cannot have smaller duration, otherwise it would not meet constraint δ. Consequently, the pair will be detectable on the checkpoint at t', as shown in the figure. Similarly, if a qualifying pair ends at timestamp $t' + \delta - 1$ (Figure 6.9b), it will be detected at the checkpoint at t'. Hence, all pairs around a checkpoint at t' can be detected as candidates when we check their values at t'. Thus, we can easily conclude to the following observation.

Lemma 2 (Checkpoint Covering Interval)**.** *Let the interval between successive checkpoints not exceed δ. Considering a checkpoint placed at timestamp t', all qualifying pairs starting at $s, t' - \delta + 1 \leq s < t'$ and ending at $f, t' < f \leq t' + \delta - 1$ will satisfy all matching constraints at timestamp t'.*

This lemma entails that it suffices to check for candidate pairs only at checkpoints, i.e., every δ timestamps. We denote the set of checkpoints as C. Since we skip timestamps and in order to avoid false misses, we now have to verify pairs with a horizontal expansion (as in 6.1.2.2), but towards both directions, i.e., before and after a given checkpoint. Overall, at each checkpoint the optimized process performs: (1) *Filtering* - Search for candidate pairs

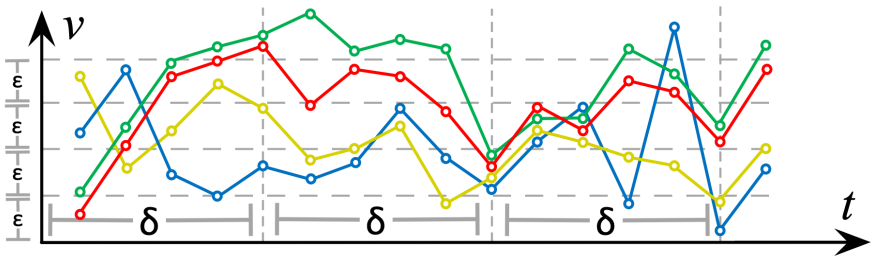


Figure 6.10: Sub-optimal checkpoint placement.

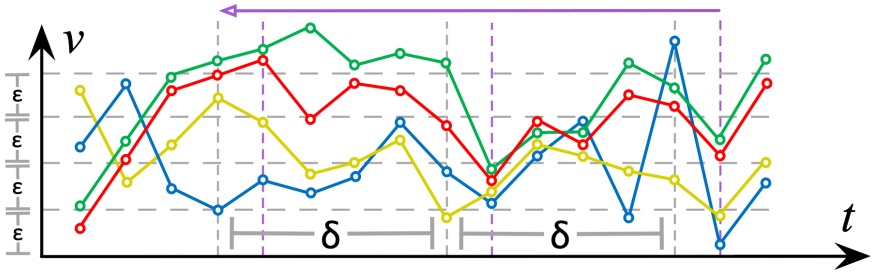


Figure 6.11: Improved checkpoint placement.

among the values of time series in adjacent bins. (2) *Verification* - For each pair, perform a *two-way* horizontal expansion across the time axis.

Improving placement of checkpoints Depending on the dataset, the default checkpoint placement might yield an *increased* number of candidate pairs, resulting in too many verifications. Intuitively, if the time series values at a specific timestamp t' are placed in a more *"scattered"* manner over the bins, less candidates would be generated. This is because the values of time series at t' would differ from each other by more than ϵ and thus can be pruned as described in Section 6.1.2.1. Figure 6.10 depicts such a case of sub-optimal placement of checkpoints, where the second checkpoint is placed at a rather *dense* area and as a result, six candidate pairs are considered. We can avoid this by shifting all checkpoints together either to the left or to the right, yet maintaining their temporal span every δ. As shown in Figure 6.11, all three checkpoints are collectively shifted to the left, avoiding the dense area and reducing the total number of candidate pairs. An extra checkpoint can be inserted before the first or after the last one, guaranteeing that there is no interval longer than δ without checkpoints.

Clearly, the placement of the set C of checkpoints influences the amount of candidate pairs. We wish to find the best such placement, which provides the least number of candidate pairs. The amount of candidates depends on the cardinality of the bins (i.e., the number of values in each one, as shown in Fig. 6.7) at any particular checkpoint $c \in C$. Given that r is the total number of time series in the dataset (and hence the number of values at each checkpoint), we can identify the most populated bin at checkpoint c by calculating the maximal density $\frac{\max\{\mathcal{B}_c\}}{r}$, where $\mathcal{B}_c$ represents the set of bin cardinalities at checkpoint c. Therefore, for a given configuration C of checkpoints, we can estimate an overall cost r by taking the sum of such maximal densities over all checkpoints, i.e., $g = \sum_{c \in C} \frac{\max\{\mathcal{B}_c\}}{r}$. The less this total cost, the smaller the cardinality per bin at each checkpoint and, thus, the less the candidates that will be generated. Consequently, we seek to find the minimum g. To do so, we shift all

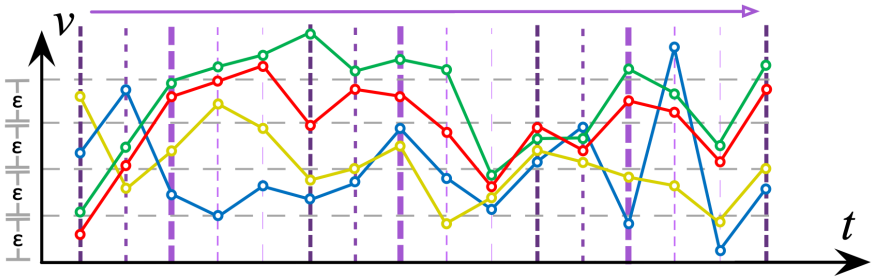


Figure 6.12: Best checkpoint placement (thick vertical lines).

checkpoints together to the right, one timestamp at a time, we estimate ratio g again and repeat δ times. This procedure is illustrated in Figure 6.12, where the checkpoints symbolized with similar lines belong to the same set as we move them to the right in order to identify the best placement (indicated with the thickest vertical dashed lines).

6.1.2.4 Pair Discovery Using Checkpoints

After identifying the best checkpoint placement, we can discover pairs of locally similar time series by applying the exhaustive algorithm presented in Section 6.1.2.2, but iterating over the defined checkpoints instead of all timestamps. To speed up the verification step, we introduce an optimization that reduces the number of checks. For each candidate pair p that started at (a possibly previous) timestamp $p.start$, we first expand its verification to the left. In case the current duration of pair p is still less than δ, we jump at timestamp $p.start + \delta$ in order to eagerly prune candidates that will not last at least δ. There, we check directly whether the values of these two time series qualify, and we continue this check backwards in time. If at an intermediate timestamp the ϵ constraint is not met, we can stop the verification process and discard the candidate pair.

The procedure for pair discovery is listed in Algorithm 9. Initially, we calculate the best possible checkpoint set (Line 2). Then, we run the procedure described in Section 6.1.2.2 but instead of probing over all timestamps we iterate over the resulting checkpoint set (Line 3).

Regarding verification, we first move towards the leftmost endpoint (Line 6) and detect the starting timestamp of the pair (Line 9). Then, we iterate from the timestamp $t_c = p.start + \delta$ towards the initial timestamp t (Line 11). If the pair does not qualify at a timestamp during this interval (Line 12), we set t as its final timestamp and return the pair (Line 14). Otherwise, we continue from timestamp t_c and towards the rightmost timestamp until the pair ceases to qualify (Lines 15-19).

Cost analysis. Let N_{tc} be the number of examined timestamps (i.e., checkpoints) considered by the algorithm. For each such timestamp, the algorithm needs to perform two operations, namely first to generate the set of candidate pairs and second to verify each pair. Let C_{cg}, C_{cv} and N_c denote, respectively, the candidate generation cost, the verification cost per candidate and the number of generated candidates. Then, the total cost is $C = N_{tc} \times (C_{cg} + N_c \times C_{cv})$. The candidate generation cost C_{cg} is proportional to the number of candidates N_c, which in turn is $\mathcal{O}(|\mathcal{T}|^2)$. However, in practice, the algorithm only needs to generate candidate pairs from the time series corresponding to the same bin. Let β denote the number of bins ($\beta \leq (y_{max} - y_{min})/\epsilon$) and N_β the maximum number of time series associated with any bin. Then, the expected cost in practice would be $\beta \times N_\beta^2$. As ϵ increases, β decreases but N_β increases (in the worst case, $N_\beta = |\mathcal{T}|$, i.e., there is a single

Algorithm 9: Checkpoint scan pair discovery

Input: Set $\mathcal{T}$ of co-evolving time series of length n
Parameters: Threshold ϵ, min duration δ
Output: A list P containing all the locally similar time series

$\mathcal{B} \leftarrow CreateBins(\mathcal{T})$
$C \leftarrow GetCheckpoints(n, \delta)$
$P \leftarrow DiscoverPairs(\emptyset, \mathcal{B}, C, \epsilon, \delta)$
return P

Procedure $VerifyPair2Way(p, t, n, \epsilon)$
  foreach $t' = t - 1 \rightarrow 0$ **do**
    if $|p.T_1.v_{t'} - p.T_2.v_{t'}| > \epsilon$ **then**
      break
  $p.start \leftarrow t'$
  $t_c \leftarrow p.start + \delta - 1$
  foreach $t' = t_c \rightarrow t$ **do**
    if $|p.T_1.v_{t'} - p.T_2.v_{t'}| > \epsilon$ **then**
      $p.f \leftarrow t$
      return p
  foreach $t' = t_c + 1 \rightarrow n$ **do**
    if $|p.T_1.v_{t'} - p.T_2.v_{t'}| > \epsilon$ **then**
      break
  $p.end \leftarrow t'$
  return p

bin that contains all time series). Moreover, any candidate that has been already generated and verified in a previous timestamp need not be verified again, so the number of candidate pairs to be verified in the worst case is $\mathcal{O}(|\mathcal{T}|^2)$. Regarding the verification cost C_{cv}, consider a pair of time series (T_i, T_j) that is generated at timestamp τ. The algorithm needs to check for each subsequent timestamp $t \in (\tau, n)$ whether $|T_i.v_t - T_j.v_t| \leq \epsilon$, as long as this condition holds; hence, the cost is $\mathcal{O}(n)$. Of course, in practice, this will also require much fewer comparisons in most cases. Notice that the difference between Algorithm 9 and Algorithm 8 is that the former generates and verifies candidates only at checkpoints, i.e., $N_{tc} = n/\delta$, while the latter at every timestamp, i.e., $N_{tc} = n$.

6.1.3 Bundle Discovery

We now consider the bundle discovery problem. We propose two algorithms: an exhaustive one using a sweep line (Section 6.1.3.1), and one using checkpoints (Section 6.1.3.2), following the same principles as in pair discovery.

6.1.3.1 Bundle Discovery Using Sweep Line

To exhaustively detect bundles of time series, we can follow a similar procedure employing a sweep line as in Section 6.1.2.2. However, this time we detect candidate bundles at each timestamp before verifying whether constraints concerning minimal duration δ and minimum membership μ are satisfied. Essentially, this can be thought of as an adaptation of the flock discovery approach in [VBT09] to the 1-dimensional setting in the case of time series.

Algorithm 10 describes this exhaustive process. Similarly to pair discovery, at each timestamp t (Line 5) we obtain only the values of time series contained in adjacent bins (Lines 6–7). Then, each such value is considered the origin of a search range ϵ, which returns a candidate group at time t (Line 9). Of course, such a candidate group may have been already included in the result bundles previously, during a horizontal expansion at a previous timestamp. In this case, its examination is skipped. Otherwise, if this group contains more than μ members, we proceed to verify it as a candidate bundle over subsequent timestamps via horizontal expansion (Lines 10-12). As will see next, this expansion may return one or more candidate bundles; each one is checked against the duration constraint δ before adding it to the result bundles (Lines 13-16).

Regarding the verification step of a candidate bundle $\mathcal{G}^t$, we apply its horizontal expansion over all subsequent timestamps (Line 19). For each member of such candidate bundle (Lines 21-28), we find the group $\mathcal{G}^{t'}$ of time series having values within range ϵ at time t'. Many such new groups may be created, as each member may yield one group. Such a group $\mathcal{G}^{t'}$ may become a new candidate bundle if it satisfies the μ constraint; if so, it is added to the resulting bundles with an updated duration (Lines 24–27). As we look at subsequent timestamps, it may happen that the same bundle may be added to the results multiple times, but with increasing duration. In the end, we eliminate duplicates and only keep the one with the longest duration (Line 28). Expansion stops once no new candidate bundles can be found in the next timestamp (Lines 29–30).

6.1.3.2 Bundle Discovery Using Checkpoints

For bundle discovery using checkpoints, we apply a similar sweep line approach, but this time we only filter at checkpoints and then verify towards both directions in the time axis. Algorithm 11 describes this procedure. After initializing the checkpoints (Line 2), we run the bundle discovery process as in Algorithm 10, but this time looking at checkpoints (Line 3) instead of all timestamps. For the two-way horizontal expansion, we first examine all candidate bundles in set Q_1 detected from the current checkpoint towards the origin of time axis (Line 7). This is done because a qualifying bundle could have started earlier, before the current checkpoint. Afterwards, we apply the same eager pruning strategy as in Section 6.1.2.4. So, we verify each such candidate bundle jumping forward at timestamp $t_c = q.start + \delta$ and continue backwards in time to its currently known start (Lines 8–10). Among the candidate bundles (in set Q_2) returned from the forward verification (Line 11), we care only for those that satisfy the minimal duration constraint δ (Line 12). These are further verified from timestamp t_c and forward in time, obtaining all subsequent qualifying bundles (Line 13).

Cost analysis. The algorithm follows a similar approach as for the case of pairs. At selected timestamps, first the candidate bundles are generated, and then each bundle is verified. In this case, since we are seeking groups of at least m time series, the number of candidates N_c is $\mathcal{O}(|\mathcal{T}|^{\mu})$, although in practice it is again expected to be much lower since only those time series corresponding to the same bin need to be considered. Finally, verification is similar with only slightly higher cost. Indeed, when checking the condition $|T_i.v_t - T_j.v_t| \leq \epsilon$, we first need to determine the time series T_i and T_j inside the bundle that have the highest and lowest values, respectively.

Algorithm 10: Sweep line scan bundle discovery

Input: Set $\mathcal{T}$ of co-evolving time series of length n
Parameters: Threshold ϵ, min duration δ, min members μ
Output: A list P containing all the discovered bundles

$\mathcal{B} \leftarrow CreateBins(\mathcal{T})$
$P \leftarrow DiscoverBundles(\emptyset, \mathcal{B}, \{0, ..., n\}, \epsilon, \delta, \mu)$
return P
Procedure $DiscoverBundles(P, \mathcal{B}, \Omega, \epsilon, \delta, \mu)$
  foreach $t \in \Omega$ **do**
    foreach $z \rightarrow \mathcal{B}.size$ **do**
      $\mathcal{T}' \leftarrow \mathcal{B}_z^t \cup \mathcal{B}_{z+1}^t$
      foreach $T \in \mathcal{T}'$ **do**
        $\mathcal{G}^t \leftarrow getAdjacent(T.v_t, \epsilon)$
        if $\mathcal{G}^t \notin P$ **then**
          if $\mathcal{G}^t.size \geq \mu$ **then**
            $\mathcal{P} \leftarrow VerifyBundle(\mathcal{G}^t, \{t, .., n\}, \epsilon, \mu)$
            foreach $\mathcal{G} \in \mathcal{P}$ **do**
              if $\mathcal{G}.end - \mathcal{G}.start \geq \delta$ **then**
                $P \leftarrow P \cup \mathcal{G}$
  return P
Procedure $VerifyBundle(\mathcal{G}^t, \{t_1, ..., t_\phi\}, \epsilon, \mu)$
  $P \leftarrow \mathcal{G}^t$
  foreach $t' = t_2 \rightarrow t_\phi$ **do**
    $expanded \leftarrow False$
    foreach $\mathcal{G} \in P$ **do**
      foreach $T \in \mathcal{G}$ **do**
        $\mathcal{G}^{t'} \leftarrow getAdjacent(T.v_{t'}, \epsilon)$
        if $(\mathcal{G}^{t'}.size \geq \mu$ **then**
          Rearrange duration of $\mathcal{G}^{t'}$ accordingly
          $P \leftarrow P \cup \mathcal{G}^{t'}$
          $expanded \leftarrow True$
      $P \leftarrow keepLongestBundles(P)$
    if $expanded = False$ **then**
      $break$
  return P

6.1.4 Experimental Evaluation

6.1.4.1 Experimental Setup

We evaluate the performance of our methods on pair and bundle discovery both qualitatively and quantitatively. We compare our checkpoint (CP) scan approaches for each problem with the respective sweep line (SL) methods. We use the water real-world dataset also used for experimental evaluation in Section 4.4 and a synthetic dataset, as listed in Table 6.1. The latter is consisted of 50,000 time series, each with a length of 1,000 timestamps. So, this dataset contains 50 million data points in total. The dataset was generated in a similar manner to the

Algorithm 11: Checkpoint scan bundle discovery

Input: Set $\mathcal{T}$ of co-evolving time series of length n
Parameters: Threshold ϵ, min duration δ, min members μ
Output: A list P containing all the discovered bundles

$\mathcal{B} \leftarrow CreateBins(\mathcal{T})$
$C \leftarrow GetCheckpoints(n, \delta)$
$P \leftarrow DiscoverBundles(\emptyset, \mathcal{B}, C, \epsilon, \delta, \mu)$
return P
Procedure $VerifyBundle2Way(\mathcal{G}^t, t, n, \epsilon, \mu)$
    $P, Q_1, Q_2 \leftarrow \emptyset$
    $Q_1 \leftarrow VerifyBundle(\mathcal{G}^t, \{t-1, ..., 0\}, \epsilon, \mu)$
    foreach $q \in Q_1$ **do**
        $t_c \leftarrow q.start + \delta - 1$
        $Q_2 \leftarrow Q_2 \cup VerifyBundle(q, \{t_c, ..., q.start\}, \epsilon, \mu)$
    foreach $q \in Q_2$ **do**
        if $q.end - q.start \geq \delta$ **then**
            $P \leftarrow P \cup VerifyBundle(q, \{t_c, ..., n\}, \epsilon, \mu)$
    return P

synthetic dataset used in [KP99].

Table 6.1: Datasets used in the experiments.

Dataset	*Size*	*Time series length*
Water	822	168
Synthetic	50,000	1,000

All experiments were conducted on a Dell PowerEdge M910 with 4 Intel Xeon E7-4830 CPUs, each containing 8 cores clocked at 2.13GHz, 256 GB RAM and a total storage space of 900 GB.

6.1.4.2 Evaluation Results

We conducted two sets of experiments, using the water and synthetic datasets. The water dataset was used for qualitative and quantitative assessment of our methods on pair and bundle discovery, while the synthetic dataset was used for efficiency evaluation.

6.1.4.2.1 Pair and Bundle Discovery over Real Data

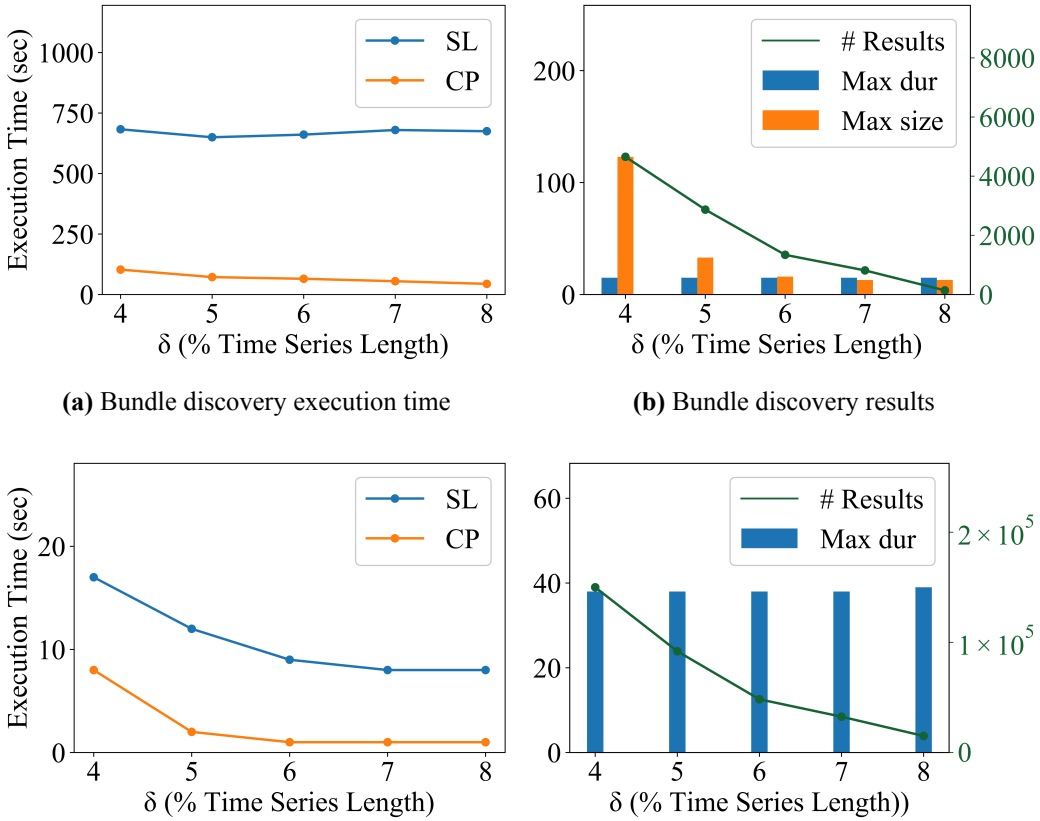
We performed several experiments using the water dataset for various parameter values to detect pairs and bundles using both the SL and CP approaches. The dataset was *z-normalized* to eliminate amplitude discrepancies among time series and focus on structural similarity.

To evaluate our methods against different parameters, we performed preliminary tests to extract ranges of values where the algorithms would return a reasonable number of results. Table 6.2 lists the range of values for the parameters used for bundle and pair discovery tests (recall that parameter μ is not applicable in pair discovery); default values are in bold. Parameter δ is expressed as a percentage of the duration of the time series, ϵ is expressed as

a percentage of the value range (i.e., difference $max - min$ in values encountered across the dataset) and μ is expressed as a percentage of the number of time series in the dataset.

Table 6.2: Parameters for tests over the water dataset

Parameter	*Values*
δ (% of time series length, i.e., 168)	6%, 5%, **6%**, 7%, 8%
ϵ (% of value range, i.e., approx 11.4)	4%, 5%, **6%**, 7%, 8%
μ (% of dataset size, i.e., 822)	0.5%, 0.75%, **1%**, 1.25%, 1.5%



(a) Bundle discovery execution time

(b) Bundle discovery results

(c) Pair discovery execution time

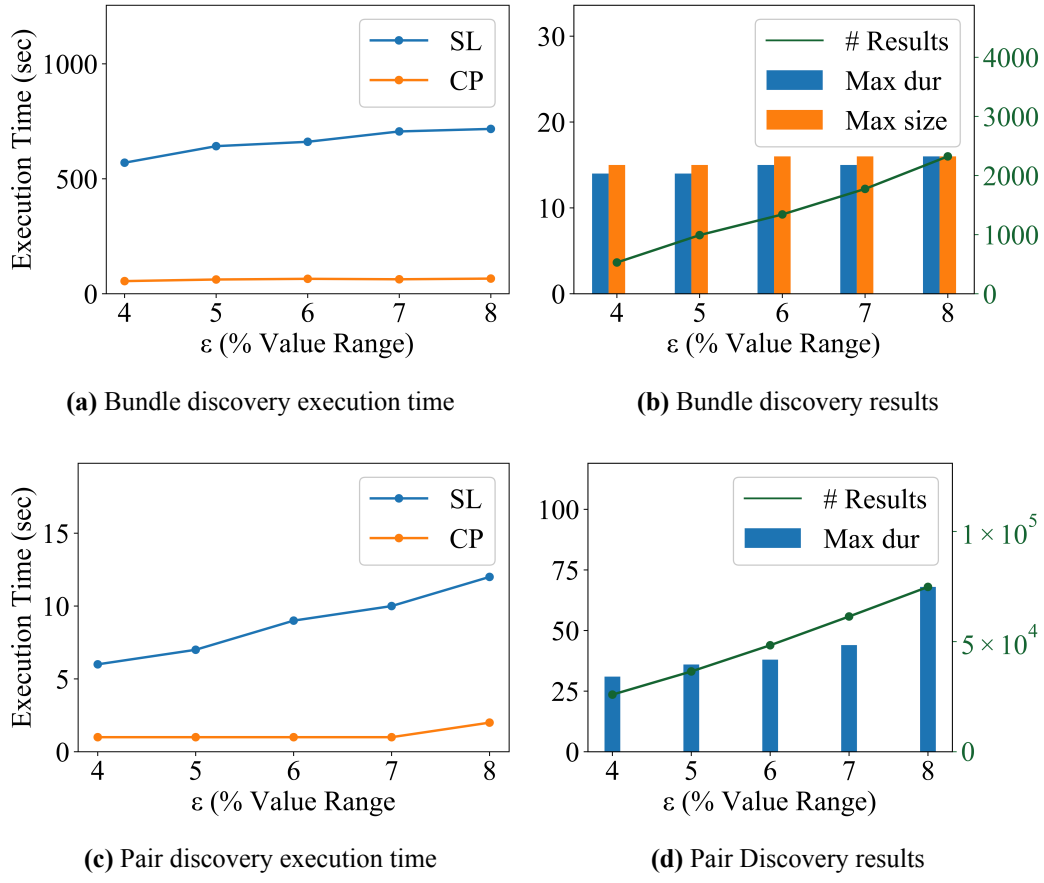
(d) Pair Discovery results

Figure 6.13: Assessment against real data for varying δ.

Varying δ Figure 6.13 depicts the results for varying minimal duration δ. In bundle discovery, the CP algorithm outperforms SL up to an order of magnitude in terms of execution time (Figure 6.13a). As the threshold δ gets larger, performance is improved due to less checkpoints being specified and less candidates needing verification. On the contrary, the SL approach performs similarly irrespective of δ, as time series must be checked at all timestamps. From Figure 6.13b, it turns out that the number of detected bundles is reduced as the δ value increases, which is expected as fewer bundles can last longer. In this plot, the blue bars indicate the maximum bundle duration among the ones that were detected, while the orange

bars indicate the larger detected bundle in terms of membership. It is clear that the maximum bundle size is drastically reduced as the number of results diminish, while the maximum duration among bundles remains the same with the increase of δ, as the longest bundle is the same in these results.

Regarding pair discovery, since it is an overall faster process, the differences in terms of efficiency are smaller, but still apparent. In this case, the execution time (Figure 6.13c) is more abruptly reduced in both SL and CP methods, since less subsequences qualify as pairs. The number of results (Figure 6.13d) is now naturally much larger, as far more pairs are expected to be verified if bundles exist. The same stands for the maximum duration among pairs, which tend to last longer compared to bundles. Since a pair is actually a bundle with μ=2 members, it is easier to find local similarity over longer intervals between two subsequences rather than an increased number of them. The maximum duration, as in bundle discovery, remains the same as δ increases, since this corresponds to the same pair in the results.



(a) Bundle discovery execution time

(b) Bundle discovery results

(c) Pair discovery execution time

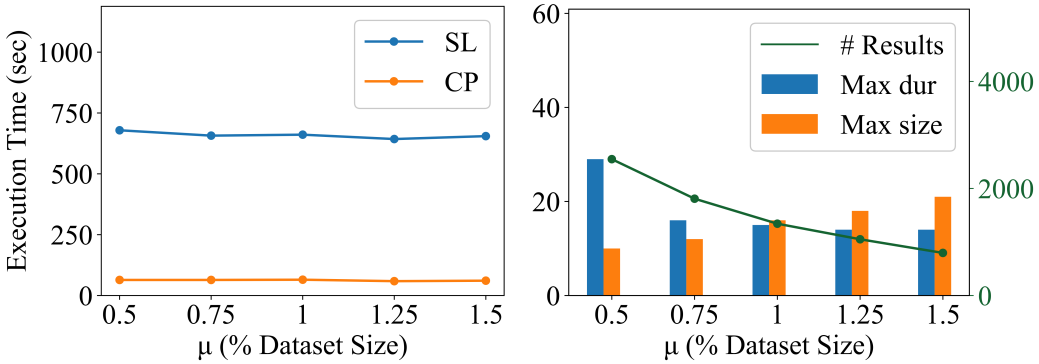
(d) Pair Discovery results

Figure 6.14: Assessment against real data for varying ϵ.

Varying ϵ Varying threshold ϵ for bundle discovery slightly incurs more execution cost for both SL and CP approaches. This is due to the increased number of bundles that need to be verified. Nonetheless, the difference in cost remains at levels of at most an order of magnitude, as shown in Figure 6.14a. As expected, the number of results is also increased

(Figure 6.14b. So does the maximum duration bundle, which is also expected due to more qualifying bundles, hence a higher probability to find longer ones. The maximum bundle size (i.e., membership) is also increased, as more time series can form a bundle when allowing a wider threshold ϵ in deviation of their respective values.

Regarding pair discovery, the results are again similar to bundle discovery for varying ϵ. For very small ϵ values of up to 7% of the value range, the CP algorithm returns results almost instantly. The SL approach is at least five times slower, with its performance deteriorating more rapidly with increasing ϵ values. Again, as in bundle discovery, the results are growing with greater ϵ, as does the maximum duration among pairs, especially for ϵ equal to 8% of the value range.



(a) Bundle discovery execution time

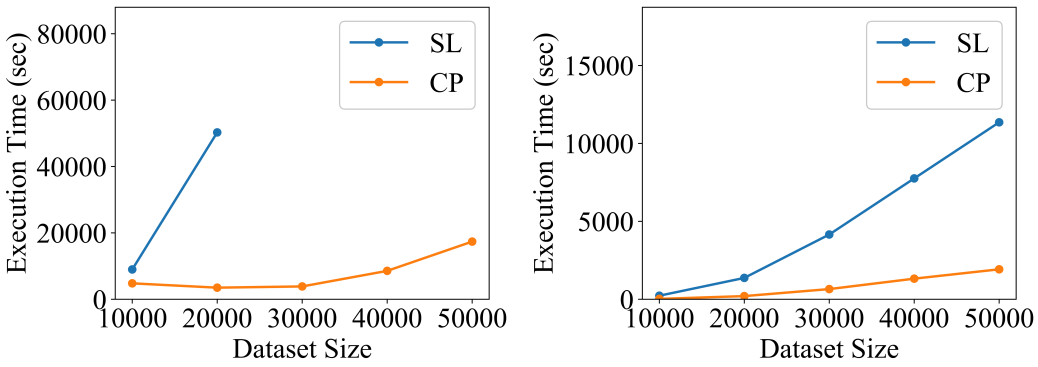
(b) Bundle discovery results

Figure 6.15: Assessment against real data for varying μ.

Varying μ When varying the minimum membership parameter μ in bundle discovery (Figure 6.15), the results regarding execution time are again very similar to the rest of the tests as indicated in Figure 6.15a. Again, the CP algorithm outperforms SL up to one order of magnitude. The execution time is very slightly decreased for larger μ values in both algorithms, as more candidate bundles are pruned. Results from CP are reported almost instantly, since the number of time series is rather small and scanning through the limited number of checkpoints is very fast. This explains why performance of SL does not get drastically improved as μ gets larger, since filtering and verification has to be repeated at every timestamp. The number of results (Figure 6.15b) is reduced as μ increases, which is expected, as less bundles get detected with a larger membership. The maximum duration among bundles also decreases; interestingly, the maximum size detected among bundles increases as the number of results diminishes, due to the growing number μ of required number of members per bundle.

6.1.4.2.2 Efficiency against Synthetic Data

To evaluate the efficiency of our methods, we used the synthetic dataset. Regarding parameter values, as in the previous experiments, we performed preliminary tests to extract ranges of values where the algorithms return a reasonable number of results. Table 6.3 lists the range of values for all parameters used in these efficiency tests for bundle and pair discovery, with the default values emphasized in bold (again, μ is not applicable in pair discovery).



(a) Bundle discovery **(b)** Pair discovery

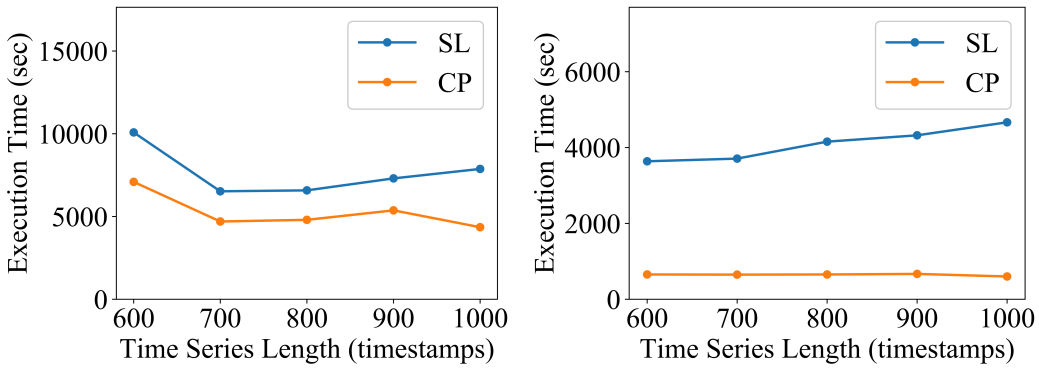
Figure 6.16: Efficiency with varying numbers of time series.

Table 6.3: Parameters for tests against synthetic data

Parameter	*Values*
Dataset Size	**10000**, 20000, 30000, 40000, 50000
Time Series Length	600, 700, **800**, 900, 1000
δ (% of time series length)	**2.5%**
ϵ (% of value range)	**0.2%**
μ (% of dataset size)	**1.25%**

Varying Dataset Size Figure 6.16 depicts the performance comparison between CP and SL algorithms for bundle and pair discovery. We omit cases where execution of an algorithm was taking more than 15 hours (cutoff). As illustrated in Figure 6.16a, an increase in the dataset size leads to a very abrupt deterioration of performance for the SL algorithm of up to several hours of execution for 20,000 time series. For larger dataset sizes, the execution time was significantly longer than the cutoff time. On the other hand, CP reports results in all cases. Its execution time increases for larger dataset sizes, but manages to finish in a few hours in the worst case (for 50,000 time series). It is worth noting that membership parameter μ is more relaxed for larger dataset sizes, as it is expressed in terms of percentage of the total number of time series in the dataset. This explains the slight improvement in performance for dataset sizes of 20,000 and 30,000 for the CP method. In general, CP is more than an order of magnitude faster than SL, which also stands for the case of pair discovery, as illustrated in Figure 6.16b. As the number of time series in the dataset grows, it is natural that more pairs will be detected, hence the linear increase in the execution cost for the CP algorithm. Of course, baseline SL requires more time, as it must check many more combinations of time series.

Varying Time Series Length For time series with increasing length (Figure 6.17a), the CP algorithm for bundle discovery again constantly outperforms SL. Similarly to previous experiments, δ is expressed as a percentage of the time series length. We observe that the execution time initially decreases for both algorithms, as more bundles are pruned. However, as the time series length (and δ) gets larger, the performance of both algorithms slightly



(a) Bundle discovery **(b)** Pair discovery

Figure 6.17: Efficiency with varying length of time series.

worsens. Only in the case of 1,000 timestamps the execution time starts to drop for the CP algorithm due to the even larger δ. This is not the case with the SL method, which has to evaluate more timestamps. Notice that the difference between the SL and CP algorithms is smaller, compared to the real dataset. This is due to the larger number of existing bundles in the synthetic dataset, which were detected on the checkpoints and had to be verified. Similar observations stand for pair discovery, with the CP algorithm significantly outperforming SL in all cases (Figure 6.17b).

6.2 Local Similarity Search on Geolocated Time Series

6.2.1 Problem Definition

We first define the local similarity score between time series, and then present the query variants we consider in this section.

Definition 7 (Local Similarity Score)**.** *The* local similarity score σ *between two time series* T *and* T' *is the maximum count of consecutive timestamps during which the respective values of* T *and* T' *do not differ by more than a given margin* ϵ*, i.e.,* $\sigma(T, T', \epsilon) = |I_{max}|$*, where* I_{max} *is the longest consecutive time interval* I *such that* $\forall i \in I, |T.v_i - T'.v_i| \leq \epsilon$.

Our goal is to efficiently support hybrid queries on geolocated time series that retrieve the results based both on spatial proximity and local similarity. Specifically, we focus on the following types of queries (hereafter referred to as *LS-queries*):

- $Q_{rr}(T_q, \rho, \epsilon, \delta)$: Given a geolocated time series T_q, retrieve every geolocated time series T such that T is located within range ρ from T_q, i.e., $dist_{sp}(T_q, T) \leq \rho$ and has local similarity score against T_q at least δ, i.e., $\sigma(T_q, T, \epsilon) \geq \delta$.
- $Q_{kr}(T_q, k, \epsilon, \delta)$: Given a geolocated time series T_q, retrieve the spatial k-nearest neighbors to T_q that also have local similarity score against T_q at least δ.
- $Q_{rk}(T_q, \rho, \epsilon, k)$: Given a geolocated time series T_q, retrieve the top-k geolocated time series that have the highest local similarity score against T_q with respect to ϵ and are located within range ρ from T_q.

Example 6. *Figure 6.4 depicts an example of the $Q_{rr}(T_q, \rho, \epsilon, \delta)$ query. Given the geolocated time series T_q as query, we seek the spatially close ones (i.e., within a circle of radius ρ) that are also locally similar within margin ϵ for at least δ timestamps. In this example, despite five geolocated time series being within range, only T_2 and T_7 qualify for the final result, since these are the ones that are also locally similar for at least one time interval of length at least δ.*

6.2.2 LS-Queries Using the BTSR-tree

A straightforward approach for answering LS-queries would be to use a spatial index to first filter by spatial distance and then perform a sequential scan across each result to filter out those having local similarity score below the given threshold. This suffers from generating an unnecessarily large number of intermediate results which are then discarded. Instead, we propose to process LS-queries by leveraging the BTSR-tree index [CSP+17], which can prune the search space simultaneously according to both criteria.

While traversing the BTSR-tree, *spatial filtering* is performed at each node N by computing the *bounding distance* $mindist_{sp}$ between the location of T_q and the MBR of N, as in R-Trees [RKV95].

For *time series similarity*, we exploit the MBTS stored within each node. Considering an MBTS B at a node N, we calculate its distance $mindist^i_{ts}$ from T_q at each timestamp i as:

$$mindist^i_{ts}(T_q, B_N) = \begin{cases} T_q.v_i - B^{\sqcap}_N.v_i, & \text{if } T_q.v_i > B^{\sqcap}_N.v_i \\ B^{\sqcup}_N.v_i - T_q.v_i, & \text{if } T_q.v_i < B^{\sqcup}_N.v_i \\ 0, & \text{if } B^{\sqcap}_N.v_i \leq T_q.v_i \leq B^{\sqcup}_N.v_i \end{cases} \tag{6.1}$$

where $B^{\sqcap}_N.v_i$ and $B^{\sqcup}_N.v_i$ are the upper and lower values of the MBTS at timestamp i. By definition of MBTS, no time series indexed under N can differ from T_q by less than $mindist^i_{ts}$ at timestamp i. Hence, only at those timestamps that $mindist^i_{ts} \leq \epsilon$, it is possible that a time series indexed under N is locally similar to T_q. Subsequently, we can compute a *local similarity bound* σ_B:

$$\sigma_B(T_q, B_N, \epsilon) = max\{|I|; \forall i \in I, mindist^i_{ts}(T_q, B_N) \leq \epsilon\}. \tag{6.2}$$

that reflects the *maximum* interval I of consecutive timestamps where the distance computed by Eq. 6.1 does not exceed margin ϵ. This value is an upper bound of the local similarity scores of T_q with any time series enclosed in this MBTS. Figure 6.18 shows that T_q deviates from the given MBTS by no more than ϵ during two intervals: one consisting of $|I_1| = 5$ consecutive timestamps and a smaller one with only $|I_2| = 2$ timestamps (shown as square points). So, the local similarity bound for this MBTS is $\sigma_B = 5$.

By construction, the MBTSs of a child node N' get tighter bounds compared to those of its parent N as we descend the BTSR-tree. It is easy to verify that

$$\sigma_B(T_q, B_N, \epsilon) \geq \sigma_B(T_q, B_{N'}, \epsilon) \tag{6.3}$$

hence local similarity bounds can only diminish when descending the index. This bound provides a useful pruning condition during search with a cutoff threshold δ. Any node where

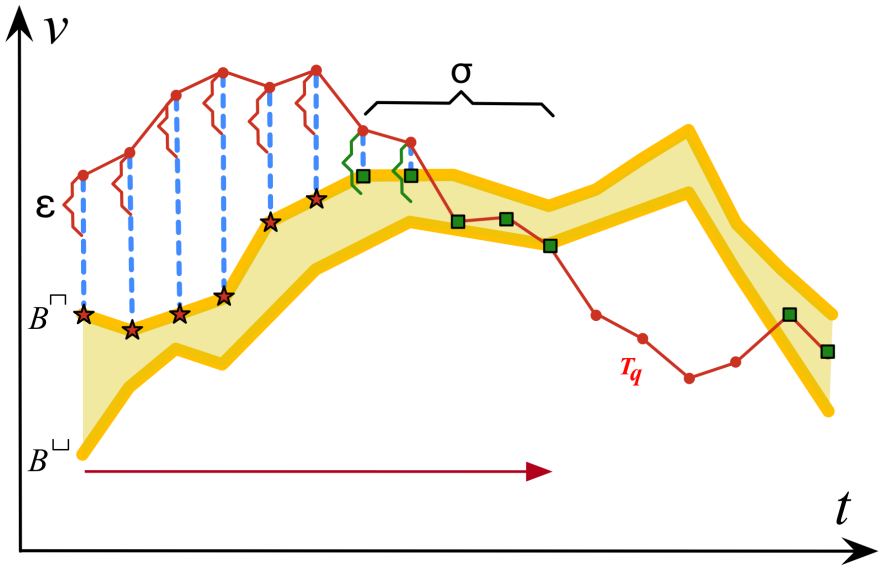


Figure 6.18: Local similarity check against an MBTS.

all its MBTSs have local similarity bound σ_B below δ can be safely pruned.

Next, we describe a baseline approach that employs a sequential scan over MBTSs, and then we present an optimization that prioritizes selected *checkpoints* to avoid many point-wise comparisons.

6.2.3 Sweep Line Approach

We explain how the BTSR-tree can be used, in conjunction with a simple sweep-line algorithm, to answer each of the three LS-queries, taking advantage of the two types of bounds, $mindist_{sp}$ and $mindist_{ts}$, described above.

$\boldsymbol{Q_{rr}(T_q, \rho, \epsilon, \delta)}$: We traverse the BTSR-tree starting from its root. At each inner node N, we first check whether $mindist_{sp}(T_q, MBR_N) \leq \rho$. If so, we employ a sweep line across the time axis to compute the local similarity bound $\sigma_B(T_q, B_N, \epsilon)$ for every MBTS included in N. If *all* resulting bounds σ_B are below δ, the subtree under N is pruned. Otherwise, the search continues at the children. Upon reaching a leaf node, we fetch the geolocated time series contained therein, and verify the query constraints against each one. Each T such that $dist_{sp}(T_q, T) \leq \rho$ and $\sigma(T_q, T, \epsilon) \geq \delta$ is added to the results.

$\boldsymbol{Q_{kr}(T_q, k, \epsilon, \delta)}$: We maintain a priority queue P containing both inner nodes (sorted by ascending $mindist_{sp}$) and geolocated time series (sorted by ascending spatial distance to T_q). We start by adding to P the root of BTSR-tree. In each iteration, we retrieve the top element from P. If it is an inner node, we visit its children to calculate local similarity bounds σ_B according to Eq. 6.2. For any child N that σ_B of one of its MBTSs satisfies threshold δ, we search the subtree of N. Then, we calculate the corresponding spatial distance ($mindist_{sp}$ for a node N or Euclidean distance for a geolocated time series T) and insert it back to P. Once we encounter a geolocated time series T at the top of P, we add it to the results. The process terminates once k geolocated time series have been obtained.

$\boldsymbol{Q_{rk}(T_q, \rho, \epsilon, k)}$: This query is evaluated similarly to the previous one, with two differences. The first difference is that the priority queue P is now sorted based on local similarity bounds in descending order, instead of spatial distance bounds in ascending order. The second is that

before inserting an item (node or time series) to P, its spatial distance ($mindist_{sp}$ or exact) is calculated, and if it is higher than ρ the item is skipped. The traversal starts again from the root, and terminates once k time series have been retrieved from the top of P. These are the top-k results with respect to local similarity (if another time series T had higher local similarity, it would have been retrieved from P first), and they are located within range ρ from T_q (otherwise, they would not have been admitted to P).

6.2.3.1 Checkpoint Approach

The drawback of the sweep-line approach is that it needs to perform a comparison for each individual timestamp to eventually determine the exact or maximum local similarity of a given time series or node, respectively. In the following, we explain how we can use *checkpoints* along the time axis to avoid this exhaustive search. These checkpoints prioritize specific timestamps when checking for candidate matches to eagerly filter out non-qualifying items.

Figure 6.8 shows an example with checkpoints placed along the time axis every $\delta = 5$ timestamps. For clarity, we consider a single time series T. Assume a checkpoint at timestamp t' and a minimal duration δ starting at timestamp $t' - \delta + 1$ for asserting local similarity with query T_q, as shown with the grey strip in Figure 6.9a. This interval cannot have smaller duration, as it would not satisfy the δ constraint. Thus, the local similarity condition will evaluate to true at checkpoint t'. Similarly, if such an interval ends at timestamp $t' + \delta - 1$ (Figure 6.9b), it will be detected at the checkpoint at t'. This observation entails that it suffices to check for local similarity only at checkpoints, i.e., every δ timestamps. We denote the set of checkpoints as C, determined at query time. If a checkpoint satisfies the condition, then we need to scan both forward and backward from it to determine the actual local similarity score, i.e., to find the exact extent of the time interval for which the condition holds.

Figure 6.19 exemplifies the use of checkpoints for comparing T_q to an MBTS of a node for $\delta = 5$ timestamps. Instead of sequentially performing 11 comparisons until verifying that local similarity score σ is at least δ (i.e., we stop the verification at $t = 11$, once $\sigma = 5$), we check only around the checkpoints. At the leftmost checkpoint c_1, no local similarity is found (T_q is farther than ϵ from the MBTS), so we skip directly to checkpoint c_2. Since T_q differs by less than ϵ at c_2, we need to compare values backward and forward, up to the previous and next checkpoint, respectively. This requires only 6 comparisons instead of 11 to decide that this node may contain candidates. Next, we describe how probing with checkpoints is applied during evaluation of LS-queries.

$\boldsymbol{Q_{rr}(T_q, \rho, \epsilon, \delta)}$: Algorithm 12 outlines the procedure. Initially, we obtain the children of the root node in a list and place the checkpoints every δ timestamps (Line 1). We iterate over each item N in this list. If N is an inner node, we have to examine whether both constraints with respect to ρ and δ are met for each of its children. Verification of MBTS against query T_q will be discussed shortly. If this is the case, we traverse the sub-tree of each child in the same manner, by adding it to the list (Lines 5-9), thus descending the tree. If the examined node is a leaf (Line 10), we iterate over each contained time series T to check the constraints ρ and δ. If T qualifies, it is added to the results (Lines 11-13). Note that now the calculation of local similarity scores for geolocated time series is based on checkpoints (Line 12), as discussed above.

Verification of MBTS against the local similarity constraints ϵ, δ is applied using checkpoints (Lines 15-36). This verification concerns each MBTS in a given node N'. At each

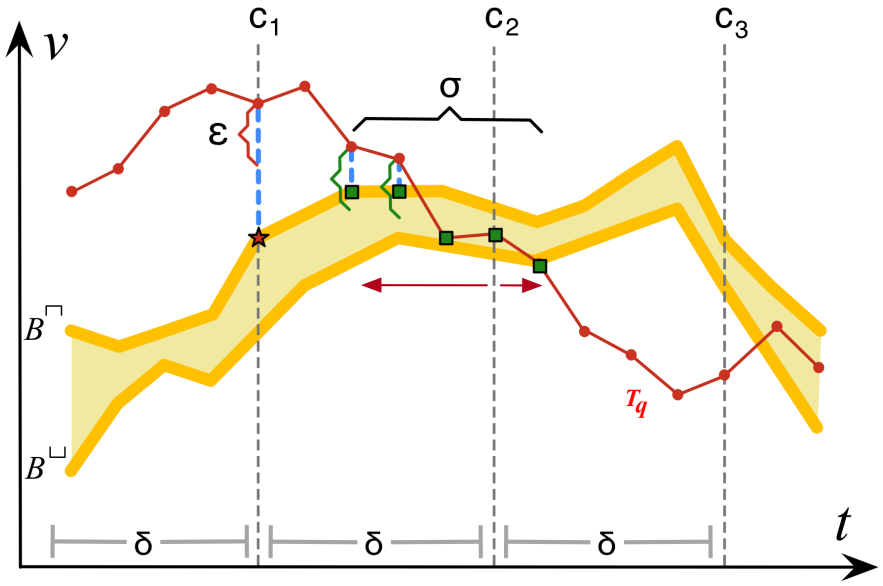


Figure 6.19: Local similarity with a MBTS using checkpoints.

checkpoint c, we first verify whether its $mindist^c_{ts}$ to query T_q is at most ϵ (Line 18). If so, we first scan backward to inspect whether there are at least δ consecutive timestamps where T_q deviates by at most ϵ from this MBTS (Lines 20-27). Similarly, we probe forward from checkpoint c (Lines 28-35). In either case, once local similarity no longer holds at a timestamp, probing skips to the next checkpoint. If the check fails for all checkpoints of all MBTSs, then this node cannot contain any results (Line 36).

$\boldsymbol{Q_{kr}(T_q, k, \epsilon, \delta)}$: We follow a similar procedure to the one in Section 6.2.3 for query Q_{kr}, employing the same verification process over MBTSs and time series as in Algorithm 12. Algorithm 13 describes the procedure. We start by adding the root node to a priority queue P based on spatial distance (Line 2). After determining the checkpoints using the given δ (Line 3), we iteratively retrieve elements from P (Line 5). Then, three cases may occur:

(i) If this element is a time series (Lines 6-9), it is guaranteed to be a result, given that P is sorted based on spatial distance from T_q. Indeed, any subsequent element must be located farther than the current. When list R obtains the required number k of results, the search terminates.

(ii) The element is a leaf node (Lines 10-14): In this case, we obtain each time series T contained in this leaf, and verify the local similarity score of T against δ. If the condition is met, we calculate the spatial distance of candidate T from query T_q and push T into the priority list along with its spatial distance (Lines 10-14).

(iii) If the element is an inner node, we iterate over its children and only push back to the queue the ones whose MBTSs are verified against ϵ and δ using checkpoints (Lines 15-19).

$\boldsymbol{Q_{rk}(T_q, \rho, \epsilon, k)}$: The procedure for this query is listed in Algorithm 14. Notice that for employing checkpoints, we need a local similarity threshold δ, so as to determine their placement, but this query does not specify a fixed δ. To be able to obtain one during search, we now maintain two priority queues: P holds inner nodes sorted by local similarity bounds

Algorithm 12: $Q_{rr}(T_q, \rho, \epsilon, \delta)$

```
1  R ← ∅, List ← Root.entries, C ← determineCheckpoints(δ)
2  while List ≠ ∅ do
3      N ← List.getNext()
4      if N is not leaf then
5          foreach N' ∈ N.getChildren() do
6              if mindist_sp(T_q, MBR_N') ≤ ρ then
7                  count ← ∅
8                  if VerifyMBTS(T_q, N', C, ε, δ) then
9                      List ← List ∪ {N'.getChildren()}
10     else
11         foreach T ∈ N.getObjects() do
12             if dist_sp(T_q, T) ≤ ρ ∧ σ^C(T_q, T, ε) ≥ δ then
13                 R ← R ∪ {T}
14         return R
15 Procedure VerifyMBTS(T_q, N', C, ε, δ)
16     foreach B ∈ N' do
17         foreach c ∈ C do
18             if mindist^c_ts(T_q, B) ≤ ε then
19                 count++, c' ← c
20                 while True do
21                     c'--
22                     if mindist^c'_ts(T_q, B) ≤ ε then
23                         count++
24                         if count ≥ δ then
25                             return True
26                     else
27                         break
28                 while True do
29                     c++
30                     if mindist^c_ts(T_q, B) ≤ ε then
31                         count++
32                         if count ≥ δ then
33                             return True
34                     else
35                         break
36     return False
```

(Eq. 6.2), while R keeps up to k geolocated time series sorted by local similarity scores (as in Def. 1). We initially set $\delta = 1$, so checkpoints are trivially placed at every timestamp. This

Algorithm 13: $Q_{kr}(T_q, k, \epsilon, \delta)$

```
1  R ← ∅
2  P.push(Root)
3  C ← determineCheckpoints(δ)
4  while P is not empty do
5      N ← P.poll()
6      if N is raw then
7          R ← R ∪ {N}
8          if |R| = k then
9              break
10     else if N is leaf then
11         foreach T ∈ N.getObjects() do
12             if σ^C(T_q, T, ε) ≥ δ then
13                 T.dist ← dist_sp(T_q, T)
14                 P.push(T, T.dist)
15     else
16         foreach N' ∈ N.getChildren() do
17             if VerifyMBTS(T_q, N', C, ε, δ) then
18                 N'.dist ← mindist_sp(T_q, MBR_N')
19                 P.push(N', N'.dist)
20 return R
```

implies that computation of local similarity scores with $\delta = 1$ is equivalent to the sweep line approach. However, δ increases with the detection of qualifying results, hence checkpoints will progressively get placed more sparsely. The search starts by adding the BTSR-tree root in P (Line 2). We iteratively poll the top element from P, and there are two possible cases:

(i) The top element is a leaf node. Then, we iterate over the contained time series and add the ones that satisfy the spatial condition (ρ) to R, along with their corresponding local similarity score σ if it exceeds the current value of δ (Lines 8-12). Once R exceeds capacity k, its last element is evicted to make room for the newly inserted one and δ is updated according to the local similarity score σ_k of the k-th element in R. In this case, the placement of checkpoints is re-adjusted according to the increased δ value (Lines 13-16).

(ii) The top element is an inner node. In this case, we iterate over each child N' and check if $mindist_{sp}(T_q, MBR'_N) \leq \rho$. If N' qualifies, we calculate the local similarity bound σ_B of all its MBTSs using checkpoints. If the maximum among these bounds $max(\sigma_B) \geq \delta$, then N' is inserted to P with this maximum score (Lines 17-25).

The process terminates once the top element in P has local similarity less than δ (Lines 6-7). The result is the contents of R.

Algorithm 14: $Q_{rk}(T_q, k, \rho)$

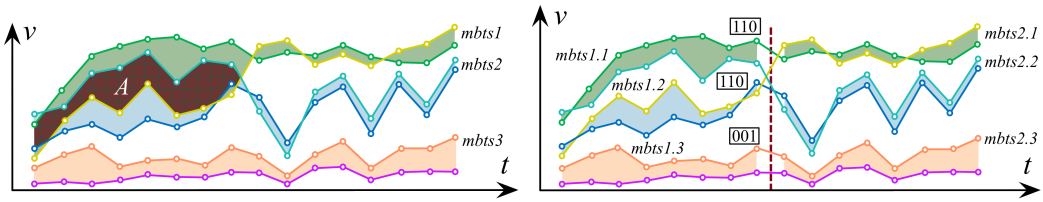
```
1  R ← ∅
2  P.push(Root)
3  δ ← 1
4  C ← determineCheckpoints(δ)
5  while P is not empty do
6      if P.peekFirst.σ_B < δ then
7          break
8      if N is leaf then
9          foreach T ∈ N.getObjects() do
10             if dist_sp(T_q, T) ≤ ρ then
11                 if σ^C(T_q, T, ϵ) ≥ δ then
12                     R.push(T, σ^C(T_q, T, ϵ))
13             if R.size > k then
14                 R.pollLast
15                 δ ← R.peekLast.σ
16                 C ← determineCheckpoints(δ)
17     else
18         foreach N' ∈ N.getChildren() do
19             if mindist_sp(T_q, MBR_N') ≤ ρ then
20                 σ_B ← 0
21                 foreach B ∈ N' do
22                     if σ_B^C(T_q, B, ϵ) ≥ σ_B then
23                         σ_B ← σ_B^C(T_q, B, ϵ)
24                 if σ_B ≥ δ then
25                     P.push(N', σ_B)
26 return R
```

6.2.4 The $\mathcal{S}$BTSR-tree Index

6.2.4.1 Index Structure

The BTSR-tree index uses k-means clustering to cluster the time series under each node and then stores the MBTSs of those clusters. However, clustering entire time series typically generates many overlapping MBTSs, incurring much dead space. This has a negative impact on the pruning power of the index, especially when considering local similarities. Figure 6.20a depicts such a case of six time series indexed in a node. A k-means clustering with $k = 3$ will form the depicted MBTSs denoted with shaded colors. As a result, the dark area A represents the overlap between $mbts.1$ and $mbts.2$ and actually makes those bounds less tight. Hence, such MBTSs inflate estimates for local similarity bounds, and thus lead to unnecessarily descending further down the index.

To reduce the amount of overlap within the MBTSs of nodes, we introduce an extended version of the BTSR-tree, named $\mathcal{S}$BTSR-tree. $\mathcal{S}$BTSR-tree attempts to eliminate as much



(a) Example of a node's MBTS. **(b)** Segmenting can eliminate whitespace.

Figure 6.20: Segmenting time series yields tighter MBTS.

overlap as possible, through segmentation of time series. Figure 6.20b depicts the intuition. If we segment the time series before applying k-means, the resulting MBTSs for each segment tend to be tighter, eliminating the excessive overlap A from Figure 6.20a. The $\mathcal{S}$BTSR-tree is built similarly to BTSR-tree. The only difference is that the MBTSs of each node are calculated *per segment*. In this method, we assume a pre-defined number s of segments, but segmentation is orthogonal to our problem and can be carried out by applying existing methods like [BGH+06]. Ultimately, $\mathcal{S}$BTSR-tree allows for more aggressive pruning when traversing the index.

6.2.4.2 Cross-Segment Continuity Via Bit-Vectors

A downside of the segmentation approach is the loss of the MBTS continuity across time, which results in MBTSs enclosing different time series in neighboring segments. For example, in Figure 6.20b, there are no MBTSs in the right segment containing the same time series as $mbts1.1$ and $mbts1.2$, a fact which hinders the calculation of local similarity on the segment boundaries (the vertical line). To overcome this, we introduce a *bit-vector* V along each MBTS of a segment, having one bit for each MBTS created. If in the current segment a bit in vector V of a given MBTS is set, this indicates that this MBTS encloses at least one common time series with another MBTS′ in the next segment. In the example shown in Figure 6.20b, $V = 110$ for $mbts1.1$ indicates common time series with $mbts2.1$ and $mbts2.2$ in the next segment, while $V = 001$ for $mbts1.3$ signifies common time series with only $mbts2.3$. This way, to calculate local similarity, we can easily identify all the MBTSs that share common time series among two successive segments.

To evaluate LS-queries, traversal of the $\mathcal{S}$BTSR-tree index follows a similar rationale to the procedure in Section 6.2.3.1. For each checkpoint c, we first obtain the segment where it falls in, and we scan each MBTS leftward and rightward from c, as discussed in Section 6.2.3.1. If we cross the border to another segment, the available bit-vectors directly identify the MBTS that need be examined in this neighboring segment. This propagates until the local similarity constraints (ϵ and δ) are satisfied. Figure 6.21 illustrates an example of a node verification. Let us consider a predetermined number of three segments and the corresponding MBTS of each segment for that node. Suppose that there exists a checkpoint c on the second segment. To verify whether this node satisfies the local similarity constraints, we start from checkpoint c and we check leftwards whether $mindist^i_{ts} \leq \epsilon$ for each timestamp. If the currently examined timestamp falls in the first segment, we fetch the corresponding MBTS and bit-vectors and continue checking whether $mindist^i_{ts} \leq \epsilon$ in both MBTS (green shaded), as their bit-vectors both indicate common members with the first one in segment 2. A similar procedure is followed rightwards, where we only have to check the first MBTS, according to the bit-vectors.

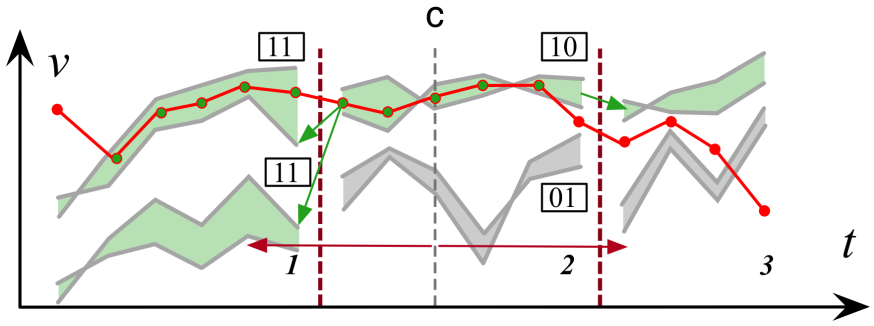


Figure 6.21: Example of verifying a $\mathcal{S}$BTSR-tree node.

6.2.4.3 Cost Analysis

Next, we analyze the cost of the Q_{rr} query (the other queries have similar costs). For index traversal, since the index is an augmented R-tree, the basic cost for searching over an R-tree applies here as well [Gut84]. However, there is an extra cost which involves two parts. The first part concerns MBTS verification. Assume a query time series T_q of length n that is verified against the MBTS of a node N. For each checkpoint, the algorithm checks for each timestamp t among two segments, whether the mindist at t between T_q and the node's MBTS is less than ϵ (see Equation 6.3). This is repeated at each neighboring segment for each MBTS whose bit vector is 1, until threshold δ is satisfied, or rejected for all checkpoints. Thus, this extra cost is $O(c * b^2 * s * g)$ in the worst case, where c is the number of checkpoints, b is the number of MBTS, s the number of segments, and $g = n/s$ the number of timestamps between two segments. In practice, this will typically require much fewer comparisons, since the node is traversed only when a qualifying interval is found. The second part of extra cost concerns time series verification. To verify T_q against T, the algorithm needs to check for each timestamp t whether the value difference between T_q and T is less than ϵ, and keep the largest detected one; hence, this extra cost is $O(n)$.

6.2.5 Experimental Evaluation

Next, we report results from a comprehensive evaluation of our methods against real-world datasets.

6.2.5.1 Experimental Setup

Datasets. We use the real-world datasets also used for experimental evaluation in Section 4.4, containing diverse types of geolocated time series as detailed in Table 6.4. To test scalability, we generated a synthetic, augmented version of the Flickr dataset by slightly moving each location in a random manner and altering each time series value by a random number between 1 and 10. We produced three additional synthetic datasets each containing $\times 2$, $\times 3$, $\times 4$ the number of time series from the original dataset.

Index and Query Parameters. To evaluate the performance benefits observed in the experiments only based on pruning, we tuned the index parameters to fixed values. The minimum (m) and maximum (M) number of entries stored in each node are set to 40 and 100, respectively. For both BTSR-tree and $\mathcal{S}$BTSR-tree, the number of MBTS is set to 10 and

Table 6.4: Datasets and parameters used in the experiments.

Dataset	Area (km^2)	Number of locations	Length of timeseries	Default query parameters			
				ρ	ϵ	δ	k
Flickr	Earth	414,967	96	30%	7.5%	20	30
Crime	392,000	362,215	76	30%	7.5%	25	30
Taxi	2,500	417,960	168	30%	10%	20	30

for $\mathcal{S}$BTSR-tree, the number of segments s is also set to 10. The query parameters involve the spatial distance and local similarity thresholds, i.e., ρ, ϵ, δ and k. The values of these parameters are set differently for each dataset, based on their characteristics; default values are shown in Table 6.4. The value of ρ is set relatively, by setting the covered area as a percentage of the total area. Similarly, ϵ is set as a percentage of the maximum difference between the observed values.

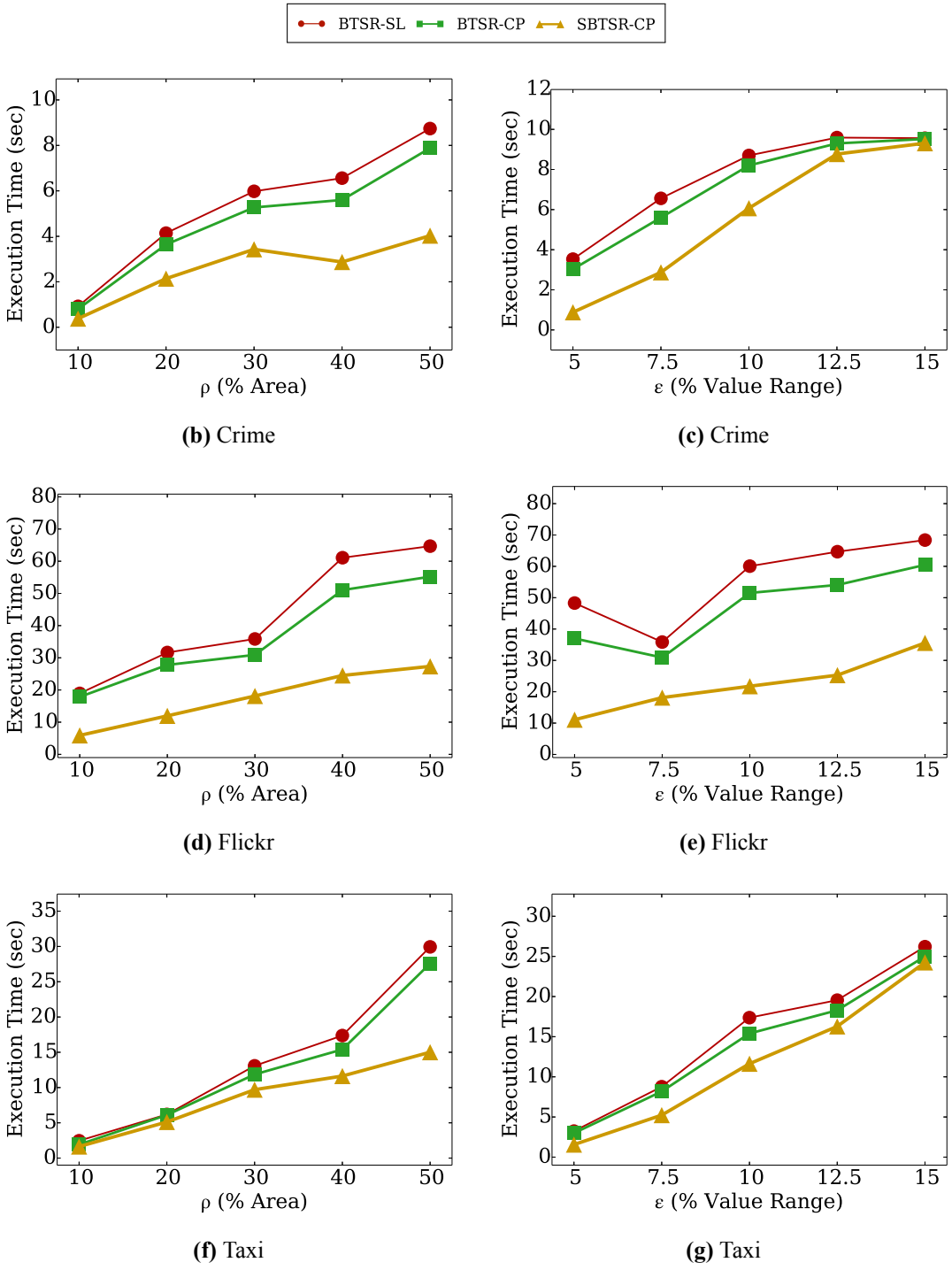
Evaluation Setting. Each experiment is performed using a randomly selected workload of 100 queries for each dataset and we report the average response time. All indices are held in memory, while the leafs contain pointers to files with geolocated time series stored on disk. All methods were developed in Java. Tests were executed on a server with 4 CPUs, each containing 8 cores clocked at 2.13GHz, and 256 GB RAM running Debian Linux.

6.2.5.2 Query Performance

We compare the average per query execution time for all three queries using sweep line and checkpoint methods on BTSR-tree and the checkpoint method on $\mathcal{S}$BTSR-tree.

$Q_{rr}(T_q, \rho, \epsilon, \delta)$. Figure 6.22 illustrates the query performance for varying thresholds ρ and ϵ and the first column of Figure 6.23 for varying δ, on all three datasets. It is apparent that the $\mathcal{S}$BTSR-tree with the checkpoint approach outperforms the rest in all cases. Its superior pruning power is attributed to the segmentation, which yields tighter bounds within the nodes and consequently less disk accesses. The sweep line and checkpoint methods over BTSR-tree perform similarly in all cases. Both methods access the same nodes, but the checkpoint approach needs to examine significantly less values across time to determine local similarities. However, since all local similarity calculations take place in-memory, computation cost does not make a big difference, compared to the less node accesses required with the $\mathcal{S}$BTSR-tree.

More specifically, for the *crime* dataset, relaxing ρ (Figure 6.22b) has a negative impact on all three methods as more nodes have to be accessed and pruning depends mostly on the ϵ value. $\mathcal{S}$BTSR-tree increasingly outperforms the rest as ρ increases, due to its more aggressive pruning on local similarity. For the case of increasing ϵ (Figure 6.22c), the result is the opposite, as this way the parameter is relaxed and more nodes get accessed. For very large ϵ values, pruning is solely based on spatial distance and all approaches perform similarly. Finally, increasing δ (Figure 6.23b) also increases the difference in performance among the three approaches, while it also reduces the average query response time. This is due to large numbers of subsequences qualifying for small δ values, resulting in more node accesses. As δ increases, pruning is more rapidly improved in the case of $\mathcal{S}$BTSR-tree due to its tighter bounds.



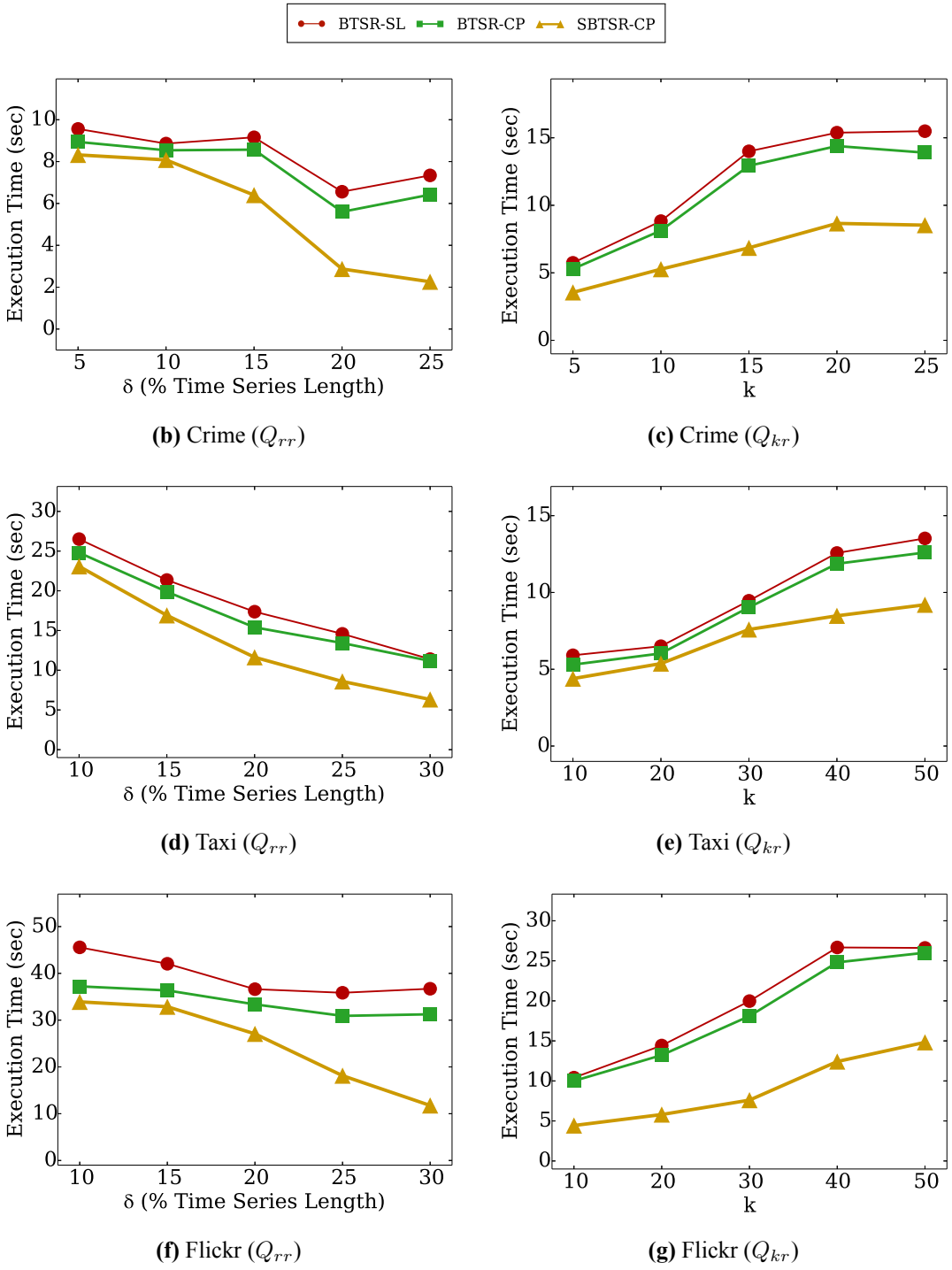
(b) Crime **(c)** Crime

(d) Flickr **(e)** Flickr

(f) Taxi **(g)** Taxi

Figure 6.22: Query $Q_{rr}(T_q, \rho, \epsilon, \delta)$ for varying ρ and ϵ.

The results are similar but with larger differences for the *Flickr* dataset (Figures 6.22d, 6.22e and 6.23f). Intuitively, the less periodicity in a dataset, the more the benefit from segmentation; if the time series in the dataset exhibit periodicity, the bounds that will occur from applying k-means clustering on the whole sequences will be relatively tighter than otherwise. The Flickr dataset, due to its nature, is more random than the crime dataset, which justifies the larger differences. This explanation is also supported by the results for the *taxi* dataset, illus-



(b) Crime (Q_{rr})

(c) Crime (Q_{kr})

(d) Taxi (Q_{rr})

(e) Taxi (Q_{kr})

(f) Flickr (Q_{rr})

(g) Flickr (Q_{kr})

Figure 6.23: Per column: $Q_{rr}(T_q, \rho, \epsilon, \delta)$ for varying δ – $Q_{kr}(T_q, k, \epsilon, \delta)$ for varying k.

trated in Figures 6.22b, 6.22c and 6.23b. Despite a similar behavior in varying all thresholds, the differences in average query response time among the different approaches are smaller than in the crime and Flickr datasets, due to the high daily periodicity of taxi drop-offs.

Another observation is that the execution cost for queries against the Taxi dataset is lower than that against Flickr. Although these two datasets have a similar number of locations, their spatial distribution and extent differ substantially (Taxi data spans New York city, while

Flickr data spans the entire planet), which may significantly affect pruning during search. To verify this, we ran a test with a random Q_{rr} query, $\rho = 30\%$ and the default parameters, and we measured the number of pruned nodes. For the query against the Taxi dataset, 3017 nodes were pruned in the tree as opposed to only 360 nodes in the tree built for the Flickr data. Since spatial filtering is much faster with our approach, this explains the difference in execution cost against these two datasets.

$Q_{kr}(T_q, k, \epsilon, \delta)$. Figures 6.23c, 6.23g and 6.23e depict the results for the $Q_{kr}(T_q, k, \epsilon, \delta)$ query for the three datasets. As k increases, more nodes have to be traversed in order to fetch the additional results, and the execution time increases for all methods. Nevertheless, $\mathcal{S}$BTSR-tree still clearly outperforms the other two algorithms.

$Q_{rk}(T_q, k, \rho)$. Finally, Figures 6.24f, 6.24b and 6.24d depict the results for the $Q_{rk}(T_q, k, \rho)$ query. In this case, the performance deterioration as k increases is less abrupt, especially for the crime dataset, as usually the top-k results are spatially closely located and are retrieved quickly. Again, the largest and smallest differences are spotted on the Flickr and taxi datasets, respectively.

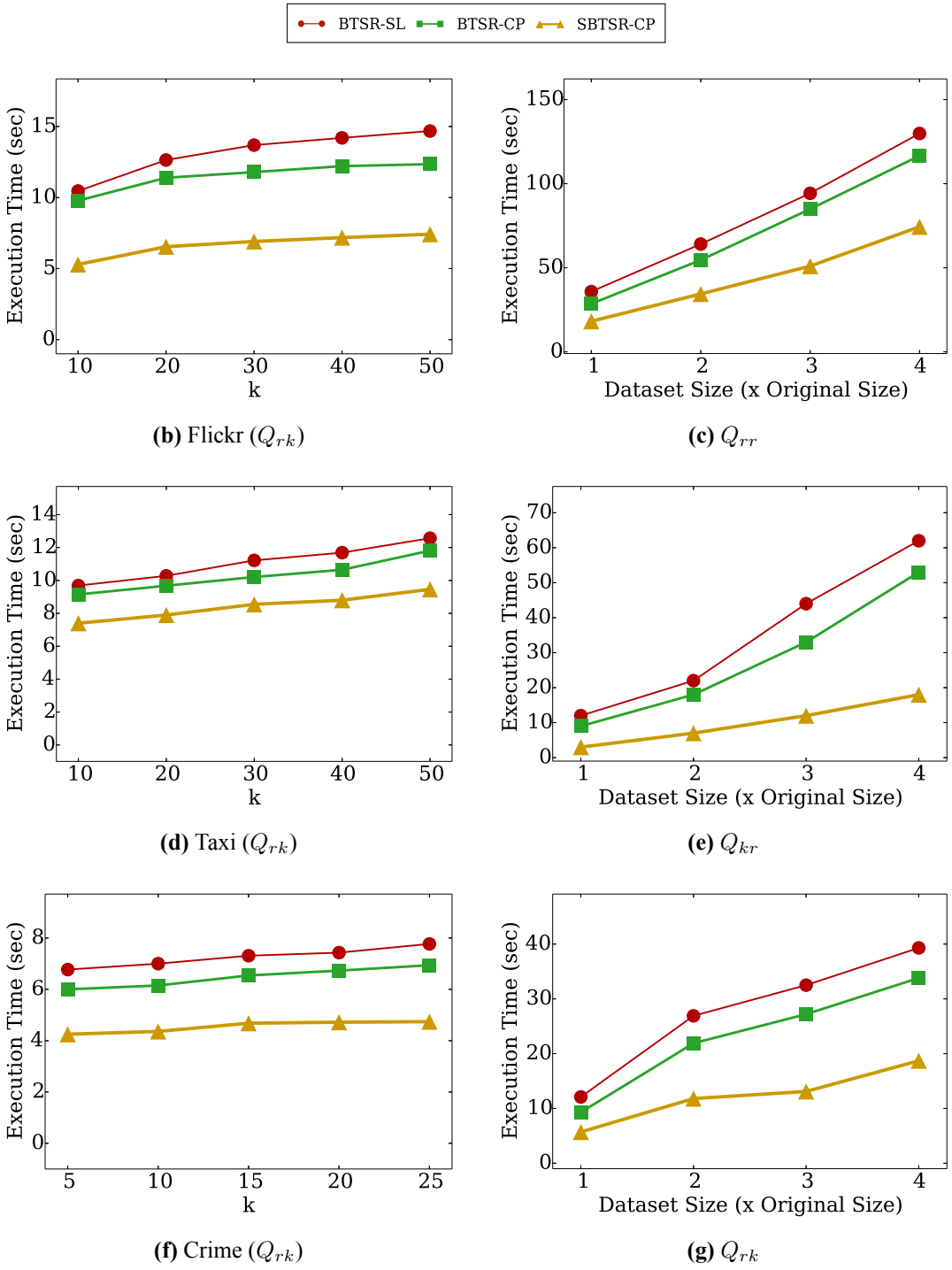
6.2.5.3 Scalability

We performed a scalability evaluation for all three queries using the Flickr-based synthetic datasets, again measuring the average query response time for the same query workload. The results for increasing dataset size (up to four times) are depicted in Figure 6.24. In all cases, the $\mathcal{S}$BTSR-tree-based approach scales better, especially in the top-k queries (Figures 6.24e and 6.24g), where the larger difference observed in Figures 6.23g and 6.24b is further augmented.

6.3 Summary

In this chapter, we addressed the problems of pair and bundle discovery over co-evolving time series, according to their local similarity. We introduced two efficient algorithms for pair and bundle discovery that utilize checkpoints appropriately placed across the time axis in order to aggressively prune the search space and avoid expensive exhaustive search at each timestamp. Our methods successfully detect locally similar subsequences of co-evolving time series, as demonstrated in our experimental evaluation over real-world and synthetic data. Also, they were orders of magnitude faster compared to baseline methods that apply a sweep line approach, confirming their effectiveness in pair and bundle discovery.

Furthermore, we have studied three variants of hybrid queries on geolocated time series, involving both range and top-k search, and combining spatial distance with local time series similarity. The latter allows to measure similarity of time series over subsequences instead of their entire length, and thus enables the identification of more fine-grained trends and patterns. The queries are evaluated by hybrid index structures, in order to allow for simultaneous pruning by both criteria. We first discuss query evaluation using the previously proposed BTSR-tree, and then we further extend it to derive the $\mathcal{S}$BTSR-tree which exhibits even better performance, by using temporal segmentation of time series to derive tighter



(b) Flickr (Q_{rk})

(c) Q_{rr}

(d) Taxi (Q_{rk})

(e) Q_{kr}

(f) Crime (Q_{rk})

(g) Q_{rk}

Figure 6.24: Per column: $Q_{rk}(T_q, k, \rho)$ for varying k – Scalability.

bounds. Our evaluation against several real-world datasets has shown that $\mathcal{S}$BTSR-tree can compute results much faster for all query variants.

In the following chapter, we present our distributed machine learning framework for fast and efficient classification and regression, using an optimized parallelization approach of the k-Nearest Neighbors joins algorithm.

Chapter 7

Scalable Time Series Forecasting

Forecasting time series is crucially useful in various applications, such as resource demand management (e.g., water, electricity, natural gas), stock market and supply demand forecasting (e.g., in super markets). An electricity provider, for example, could forecast future demand on its power supply network, based on the historical consumption time series of its customers. This way, precautionary measures could be taken when a larger electricity demand is anticipated, to avoid a potential power outage. Depending on the dataset, time series forecasting can be a rather complex and computationally intensive task due to the high dimensionality and usual uncertainty in such data. Classical statistical methods (e.g., autoregressive integrated moving average, simple exponential smoothing), or more data-driven, machine learning-based (e.g., recursive neural networks) approaches could be employed for such a task. In this chapter, we present a data-driven k-Nearest Neighbors (kNN) method for large-scale analytics on Big Data.

During the past few years, new database management and distributed computing technologies have emerged to satisfy the need for systems that can efficiently store and operate on massive volumes of data. *MapReduce* [DG08], a distributed programming model which maps operations on data elements to several machines (i.e., *reducers*), set the foundation for this technology trend. This laid the groundwork for the development of open source distributed processing engines such as the Apache *Hadoop*, *Spark* and *Flink* [ABE+14], that efficiently implement and extend MapReduce. These engines offer a handful of tools that operate on Big Data stored in distributed storages, supported by distributed file systems such as the *Hadoop Distributed File System* (HDFS). Among them, Flink provides a mechanism for automatic procedure optimization and exhibits better performance on iterative distributed algorithms [GZ14]. It also exhibits better overall performance, as it processes tasks in a pipelined fashion [fli]. This allows the concurrent execution of successive tasks, i.e., a reducer can start executing as soon as it receives it's input from a mapper, without requiring all the mappers to finish first.

Data analysis and knowledge extraction from Big Data collections is often performed by applying specific machine learning techniques. The simplicity along with the effectiveness of the kNN algorithm, have motivated many research communities over the years with numerous applications and scientific approaches which exploit or improve its potential, especially in spatial databases and data mining. Of particular interest are the kNN *joins* methods [BK04], which retrieve the nearest neighbors of *every* element in a testing dataset (R) from a set of elements in a training dataset (S). Each data element consists of several *features*, which con-

stitute the preliminary knowledge on which the neighbor retrieval is conducted. However, computing kNN joins on vast amounts of data can be very time consuming when conducted by a single CPU, as it requires computing kNN for each element in dataset R. Additionally, a possible extension of such methods to perform machine learning tasks such as classification or regression, magnifies the complexity. Various studies have been also carried out towards *approximate* solutions of kNN, where there is a trade-off between the algorithm's precision and complexity.

In this chapter, we introduce a framework of methods for scalable management, analysis and mining on Big Data collections. We present the *Flink Machine Learning kNN* (FML-kNN for short) framework which implements a probabilistic classifier and a regressor. Specifically, we introduce a MapReduce-based version of kNN joins, which reduces file operations for large amounts of data and is uniquely initialized upon launch. Our approach is unified in a single session to reduce space occupation and cluster overloading. Through an experimental evaluation on real-world water consumption data, we show that the proposed method achieves high prediction precision and useful knowledge extraction.

The rest of this chapter is organized as follows. Section 7.1 presents some preliminaries and essential basic concepts. Section 7.2 presents FML-kNN. In Section 7.3, we evaluate the framework's methods in terms of wall-clock completion time and scalability. We also present and discuss two case studies on knowledge extraction tasks over large amounts of water consumption data. Finally, Section 7.4 concludes the paper and outlines our future research directions.

7.1 Preliminaries

In the following, we present basic concepts regarding classification and regression based on kNN joins, as well as methods for dimensionality reduction, essential for the implementation of FML-kNN. We also briefly describe Apache Flink, the distributed processing engine that we used.

7.1.1 Classification

A kNN joins classifier algorithm categorizes new elements in a *testing* dataset (R). It detects the nearest neighbors of each elements in a *training* dataset (S) via a similarity measure, expressed by a distance function (i.e., Euclidean, Manhattan, Minkowski). In FML-kNN we used the Euclidean distance, through which, for each query element in the testing dataset R we obtain the dominant class (i.e., class membership) among its kNNs' classes. kNN classification in most cases is performed by a voting scheme, according to which, the class that appears more times among the nearest neighbors will be the resulting class. The voting scheme can be *weighted* when someone takes into account the distances between the nearest neighbors. Then each nearest neighbor has a weight according to its distance to the query element.

Let us consider the set of the k-nearest neighbors as $X = \{x_1^{NN}, x_2^{NN}, ..., x_k^{NN}\}$ and the class of each one as a set $C = \{c_1^{NN}, c_2^{NN}, ..., c_k^{NN}\}$. The weight of each nearest neighbor, indicating its impact on the final result, is calculated as follows:

$$w_i = \begin{cases} \frac{d_k^{NN} - d_i^{NN}}{d_k^{NN} - d_1^{NN}} & : d_k^{NN} \neq d_1^{NN} \\ 1 & : d_k^{NN} = d_1^{NN} \end{cases}, \quad i = 1, ..., k \tag{7.1}$$

where d_1^{NN} is the closest neighbor and d_k^{NN} the furthest one. By this calculation, the closest neighbors will be assigned a greater weight. We extend the approach to perform probabilistic classification (more details in Section 7.2).

7.1.2 Regression

Regression is a statistical process, used to estimate the relationship between one dependent variable and one or more independent variables. In the machine learning domain, regression is a supervised method, which outputs continuous values (instead of discrete values such as classes, categories, labels, etc.). These values represent an estimation of the target (dependent) variable for the new observations. A common use of the regression analysis is the prediction of a variable's values (e.g., future water/energy consumption, product prices, web pages visibility/prediction of potential visitors), based on existing/historical data. There are numerous statistical processes that perform regression analysis, however, in the case of kNN, regression can be performed by averaging the numerical target variables of the kNN as follows.

Considering the same set of kNNs ($X = \{x_1^{NN}, x_2^{NN}, ..., x_k^{NN}\}$) and the target variable of each one as $V = \{v_1^{NN}, v_2^{NN}, ..., v_k^{NN}\}$, the value of the new observation will be calculated as:

$$v_r = \frac{\sum_{i=1}^{k} v_i^{NN}}{k}, i = 1, ..., k \tag{7.2}$$

FML-kNN regressor implements the above procedure. At this point we should note that k-nearest neighbors performs non-linear classification and regression, as it does not seek a decision hyperplane to separate the data or a straight line to fit them. Instead, it seeks the closest elements in the neighborhood of the query element based on the Euclidean distance, which results to a non-linear traversal of the space.

7.1.3 Dimensionality reduction

In order to avoid expensive distance computations caused by high dimensional input data, we reduce their dimensionality to one. This is accomplished by three different SFC implementations, namely the z-order, the Gray-code and the Hilbert curve, all supported by FML-kNN in order to provide the flexibility of tuning according to specific needs, w.r.t. time performance or accuracy. Each curve scans the n-dimensional space in a dissimilar manner and exhibits different characteristics in terms of scanning "fairness" and computation complexity [MAK02].

z-order curve The z-order curve (Figure 2.3a) is computed by interleaving the binary codes of an element's features. This procedure takes place starting from the most significant bit (MSB) towards the least significant (LSB). For example, the z-value of a 3-dimensional element with feature values 3 (011_2), 4 (100_2) and 5 (110_2), can be formed by first interleaving

the MSB of each number (0, 1 and 1) going towards the LSB, thus, forming a final value of 011101100_2. This is a fast procedure, not requiring any costly CPU execution.

Gray-code curve The Gray-code curve (Figure 2.3b) mapping computation is very similar to the z-order curve as it requires only an extra step. After obtaining the z-value as described above, it is transformed to Gray-code by performing exclusive-or operations to successive bits. For example, the Gray-code value of 0100_2 would be calculated as follows. Initially, the MSB is left the same. Then, the second bit would be an exclusive-or of the first and second ($0_2 \oplus 1_2 = 1_2$), the third and exclusive-or of the second and third ($1_2 \oplus 0_2 = 1_2$) and the fourth an exclusive-or of the third and fourth ($0_2 \oplus 0_2 = 0_2$). Thus, the final Grey-code value would be 0110_2.

Hilbert curve Finally, the Hilbert curve (Figure 2.3c) requires more complex computations in order to be calculated. The intuition behind Hilbert curve is that two consecutive points in the sequence are nearest neighbors, thus, avoiding "jumping" to farther elements, as in the z-order and Gray-code curves. The curve is generated recursively by rotating the two bottom quadrants at each recursive step. There are several algorithms that map coordinates to Hilbert coding. In this work, we employ the methods described in [Law00], offering both forward and inverse Hilbert mapping.

7.1.4 Apache Flink

Our framework was implemented using the Apache Flink distributed processing engine. Flink offers a variety of *transformations* on datasets and is more flexible than similar engines (i.e., Apache Hadoop and Apache Spark), due to the fact that it executes its jobs in a pipelined manner, thus, gaining in performance. Also, it efficiently supports iterative algorithms, which are extremely expensive in the standard MapReduce framework. The parallel tasks are executed by task managers, each one usually denoting a single machine with a number of further parallel processing slots, usually set to be the same as the number of available CPUs.

Flink is more appropriate for demanding computations performance-wise, as it does not require key-value pairs during the transitions that take place between the transformations. Instead, Java plain objects or just primitive types are used, optionally grouped in *tuples*. The grouping (partitioning) and sorting can be applied directly according to specific tuple elements or object variables, thus, avoiding the need of generating key-value pairs, which are required i.e., by Hadoop in order to properly partition and sort the elements during the shuffle and sort phase. Furthermore, Flink is equipped with built-in automatic job optimization, which achieves better performance compared to other engines.

7.2 Methods

This section outlines the design and implementation of FML-kNN. One of the main contributions of FML-kNN is the unification of the three different processing stages into a single Flink session. Multi-session implementations, regardless of the distributed platform on which they are developed and operated, are significantly inefficient due to the following reasons:

- **Multiple initializations of the distributed environment**: They increase the total wall-clock time needed by an application in order to produce the final results.
- **Extra I/O operations during each stage**: They introduce latency due to I/O operations and occupy extra HDFS space.

We avoid the above issues by unifying the distributed sessions into a single one. Figure 7.1 illustrates the unified session. The stages are executed sequentially and I/O operations on HDFS take place only during the start and end of the execution.

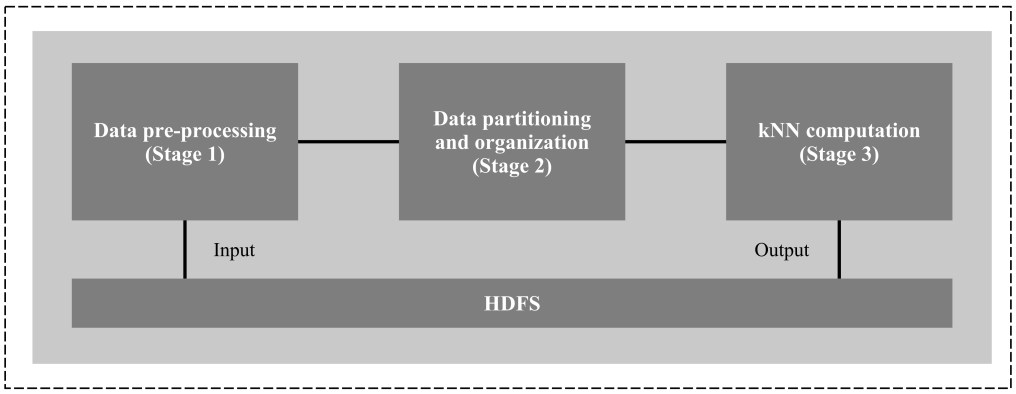


Figure 7.1: Single Flink session.

7.2.1 Dimensionality reduction and shifting

The input dataset is consisted of points in a d-dimensional space. To perform dimensionality reduction, we transform each point into SFC values via either z-order, Gray-code, or Hilbert curve. By taking a closer look on the way the SFCs fill a two-dimensional space from the smallest value to the largest, one could easily notice that some elements are falsely calculated being closer than others, as the curve scans them first. This can have a negative impact on the result's accuracy.

To diminish the negative effect of such false approximations, we generate a pre-determined number of alternate versions of the dataset, where each element is randomly shifted by a pre-calculated random vector. As an example, let us suppose that we have a dataset whose elements consists of three features and a random vector $v = \{3, 5, 2\}$. Then, all the elements of the dataset are shifted using this vector, e.g., an element $el = \{2, 6, 9\}$ will be altered to $el' = \{2 + 3, 5 + 6, 2 + 9\} = \{5, 11, 11\}$. This is demonstrated for z-order curve in Figure 3, where the four bottom-left elements are shifted twice in the x-axis and once in the y-axis, altering the sequence in which they are scanned by the curve. This way, neighboring points that are distant on the curve will possibly be closer in the shifted dataset. This procedure compensates the lost accuracy, as it enables scanning the space in an altered sequence. During execution, the shifted dataset is concatenated with the original one and the algorithm is executed, producing *multiple* groups of kNNs, from which the final kNNs are determined. The limitation of this approach is the fact that the size of the input dataset is increased according to the number of shifts $i \in [1, \alpha]$, where α is the total number of shifts. Finally, we should mention that the above process is similar for all SFCs.

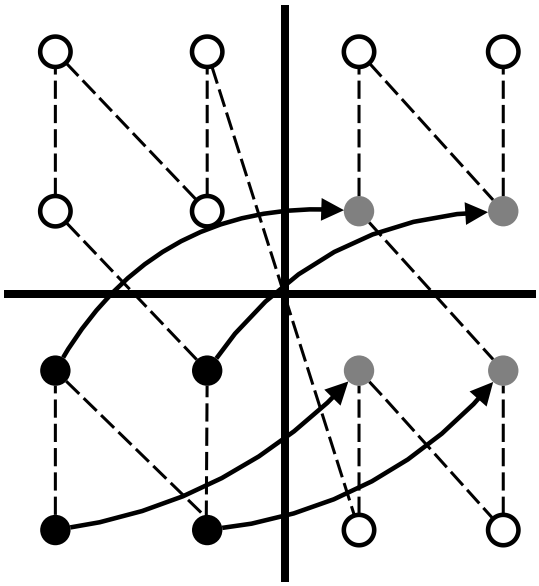


Figure 7.2: Data shifting.

7.2.2 Partitioning

A crucial part of developing a MapReduce application is the way input data are partitioned in order to be delivered to the required reducers. Similar baseline distributed approaches of kNN joins problem perform partitioning on both R and S datasets in n blocks each and cross-check for nearest neighbors among all possible pairs, thus requiring n^2 reducers. We avoid this issue by computing n overlapping partitioning ranges for both R and S, using each element's SFC values. This way, we make sure that the nearest neighbors of each R partition's elements will be detected in the corresponding S partition. We calculate these ranges after properly sampling both R and S, due to the fact that this process requires costly sorting of the datasets.

7.2.3 The FML-kNN Distributed Processing Framework

7.2.3.1 FML-kNN workflow

FML-kNN has the same workflow as other similar approaches [SRHM15], and consists of three processing stages. The workflow of the algorithm is depicted in Figure 7.3. The operations that each stage of FML-kNN performs are enumerated below (the Flink operation/-transformation is in parentheses):

- **Data pre-processing (stage 1)**:
  - Performs dimensionality reduction via SFCs on both R and S datasets (Flat Map R/S).
  - Shifts the datasets (Flat Map R/S).
  - Unifies the datasets and forwards to the next stage (Union).
  - Samples the unified dataset (Flat Map Sampling).

  - Calculates the partitioning ranges and broadcasts them to the next stage (Group Reduce).

- **Data partitioning and organization (stage 2)**:
  - Partitions the unified dataset into n partitions, using the received partitioning ranges (Flat Map).
  - For each partition and each shifted dataset, the kNNs of each element in dataset R are calculated (Group Reduce).

- **kNN computation (stage 3)**:
  - The final kNNs for each element in dataset R are calculated and classification or regression is performed (Group Reduce).

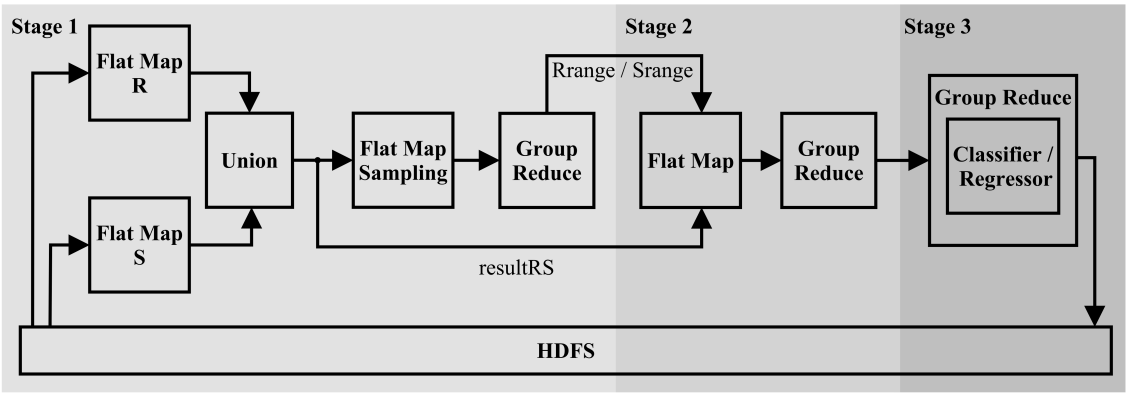


Figure 7.3: Single-session FML-kNN.

During data pre-processing, the sampling process is performed by a separate `flatMap` operation, which in return feeds the reducers with a smaller, sampled dataset used to calculate the partitioning ranges. The unified transformed datasets are directly passed to the mappers of stage 2, which also receive the partitioning ranges as a broadcast dataset via the `withBroadCastSet` operation. The data partitioning and organization stage directly feeds the input of the kNN computation. Flink's agility allows for the entire removal of the mapping procedure of stage 3, as it only propagates the results from the stage 2 to the reducers of stage 3. This increases the efficiency as it reduces the algorithm's resource requirements. In the following, we present each stage in more details.

7.2.3.2 Data pre-processing (stage 1)

The R and S datasets are read as plain text from HDFS and delivered to two separate concurrent `flatMap` transformations, identifiable by the input source file. During this stage, the SFC values (z-values for z-order, g-values for Grey-code curve and h-values for Hilbert) and possible shifts, are calculated and passed to a `union` transformation, which creates a union of the two datasets. The unified and transformed datasets are then forwarded to the next stage

(stage 2) and to a sampling process, which is performed by a separate `flatMap` transformation. The sampled dataset is then passed on a `groupReduce` transformation, grouped by a shift number. This way, α (number of shifts) reducers will be assigned with the task of calculating the partitioning ranges for R and S datasets, which are then broadcast to the next stage (data partitioning and organization).

Broadcasting the partitioning ranges significantly reduces the computational resources required by the reducers compared to F-zkNN, as it avoids the race condition between stages 1 and 2 (Figure 7.4a). During the race condition, the transformed dataset is forwarded to the second stage before the partitioning ranges are calculated by the reducers, causing its mapping operation to be initiated, which would result in an error, as the partitioning ranges are required. To avoid this race condition, F-zkNN's reducers have to locally cache the transformed dataset while it is being sampled for the calculation of ranges, as depicted in Figure 7.4b. FML-kNN overcomes this issue, by broadcasting the partitioning ranges, as this procedure blocks the execution of the second stage until the transformed dataset is ready. The left part of Figure 7.3 depicts the whole process.

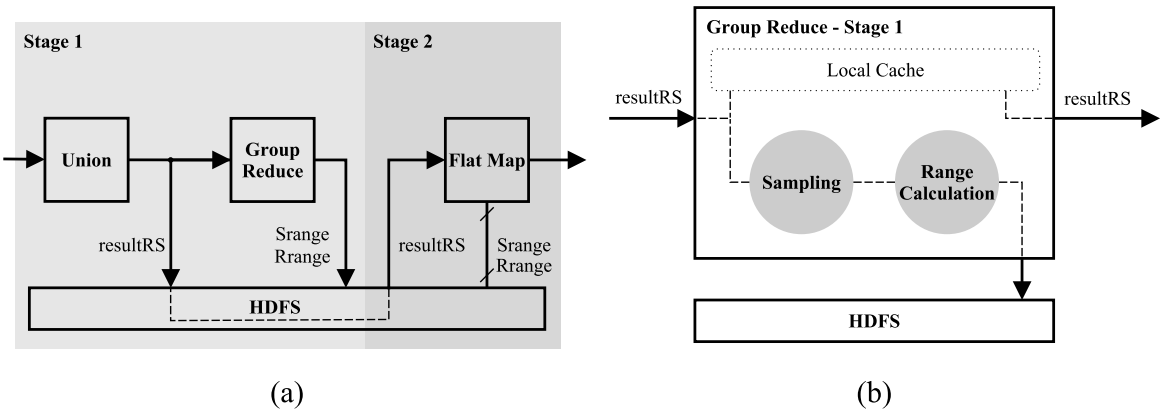


Figure 7.4: F-zkNN race condition and solution.

Algorithm 15 presents the pseudocode of stage 1. The random vectors are initially generated and cached on HDFS in order to be accessible by all the nodes which take part in the execution. They are then given as input to the algorithm, along with datasets R and S. Notice that $\mathbf{v}_1 = \overrightarrow{0}$ indicating that the datasets are not shifted during this iteration. This process takes place α times, where α is the number of shifts (Line 5). After shifting the datasets, during the first mapping phase (Lines 5-9), the elements' SFC values are calculated and collected to $\hat{R}_i^T$ and $\hat{S}_i^T$, where $i = 1, ..., \alpha$. Then, the sampling is performed by the second mapping phase (Lines 10-17). During the reduce phase (Lines 18-22), the partition ranges ($Rrange_i$ and $Srange_i$) for each shift are calculated using the sampled datasets and broadcast to stage 2 (Lines 18-20). The output consists of the unified transformed datasets, which finally feed the data partitioning and organization stage (Line 22).

7.2.3.3 Data partitioning and organization (stage 2)

The transformed datasets of stage 1 are partitioned to $n \times \alpha$ blocks via a custom partitioner, after fetching the previously broadcast partitioning ranges. Each block is then delivered to a different reducer through a `groupBy` operation. Finally, the nearest neighbors for each query

Algorithm 15: FML-kNN (stage 1).

1 ▷ The pre-processing stage's input
Input: Datasets R, S, random vectors $\mathbf{V} = \{\mathbf{v}_1, ..., \mathbf{v}_\alpha\}, \mathbf{v}_i = \overrightarrow{0}$ and sampling threshold σ
2 ▷ The output, that will be emitted to the next stage
Output: Transformed datasets R_i^T and $S_i^T, i = 1, ..., \alpha$
3 **begin**
4 ▷ The same procedure is repeated for each shift
5 **for** $i = 1, ..., \alpha$ **do**
6 $R_i = R + \mathbf{v}_i$
7 $S_i = S + \mathbf{v}_i$
8 $R_i^T \leftarrow$ calcSFC(R_i)
9 $S_i^T \leftarrow$ calcSFC(S_i)
10 **foreach** $x \in R_i^T \cup S_i^T$ **do**
11 ▷ Sampling
12 $r \leftarrow$ Random$(0, 1)$
13 **if** $r < \sigma$ **then**
14 **if** $x \in R_i$ **then**
15 InsertSample$(s, \hat{R}_i^T)$
16 **else if** $x \in S_i$ **then**
17 InsertSample$(s, \hat{S}_i^T)$
18 $Rrange_i \leftarrow$ calcRange$(\hat{R}_i^T)$
19 $Srange_i \leftarrow$ calcRange$(\hat{S}_i^T)$
20 Broadcast$(Rrange_i, Srange_i)$
21 ▷ Emit to the partitioning and organization stage
22 **return** $R_i^T \cup S_i^T$

element are calculated via proper range search operations and passed on the next stage (kNN computation). The middle part of Figure 7.3 depicts this process.

Algorithm 16 presents the pseudocode of stage 2. During the map phase (Lines 7-18), after having read each shift's broadcast partition ranges (Line 6), the received transformed datasets are partitioned into $n \times \alpha$ buckets ($R^{g \times i}$ and $S^{g \times i}$, $i = 1, ..., \alpha$, $g = 1, ..., n$, Lines 13 & 18), α being the number of shifts and n the number of partitions. The partitions $S^{g \times i}$ are then sorted and emitted to the reducers (Lines 23-30) along with the corresponding partitions $R^{g \times i}$. There, for each $x \in R$ element, a range search is performed on the proper sorted partition in order for its kNN to be determined (Line 23) and its initial coordinates are calculated (Line 24). The initial coordinates of all neighboring elements' coordinates are then calculated (Line 26) and their distance to the $x \in R$ element is computed (Line 27). Finally, all nearest neighbors are integrated into the proper dataset ($R_{k \times \alpha}$, Line 28) along with the calculated distance and feed the stage 3, grouped by $x \in R$ elements.

7.2.3.4 kNN computation (stage 3)

The calculated $\alpha \times k$-nearest neighbors of each R element, are fetched from HDFS and mapped to $|R|$ reduce tasks. The final k-nearest neighbors are determined and passed to the

Algorithm 16: FML-kNN (stage 2).

```
▷ This stage's input are the datasets emitted during the partitioning and organization stage
Input: Datasets R_i^T, S_i^T, i = 1, ..., α
▷ The output, that will be emitted to the next stage
Output: Dataset R_{k×α}
begin
    ▷ Initialization of the dataset to be emitted
    R_{k×α} = ∅
    receiveBroadcast(Rrange_i, Srange_i)
    ▷ Again, repeat for each shift
    for i = 1, ..., α do
        ▷ Partition the R elements
        foreach x ∈ R_i^T do
            for g = 1, ..., n do
                if zval(s) ∈ Rrange_i(g) then
                    addintopartition(s, R^{g×i})
        ▷ Partition the S elements
        foreach x ∈ S_i^T do
            for g = 1, ..., n do
                if zval(s) ∈ Srange_i(g) then
                    addintopartition(s, S^{g×i})
        for g = 1, ..., n do
            ▷ Sorting is needed to properly perform range search
            sort(S^{g×i})
            foreach x ∈ R^{g×i} do
                RES ← rangeSearch(x, k, S^{g×i})
                CC_x ← calcCoords(x)
                foreach neighbor ∈ RES do
                    CC_neighbor ← calcCoords(neighbor)
                    CD ← calcDist(CC_x, CC_neighbor)
                    R_{k×α} ← add(x, neighbor, CD)
    ▷ Emit to the final stage, grouped by element
    return R_{k×α}
```

classifier or regressor, depending on user preference. The latter calculate either the probability for each class (classifier) or the final value (regressor) for each element in R.

Classification In the case of classification, we extend the voting scheme, to perform probabilistic classification. We consider the set $P = \{p_j\}_{j=1}^{l}$, containing the probability that the query element will belong to each class, where l is the number of classes. The final probability for each class will be derived as follows:

$$p_j = \frac{\sum_{i=1}^{k} w_i \cdot I(c_j = c_i^{NN})}{\sum_{i=1}^{k} w_i}, \quad j = 1, ..., l \tag{7.3}$$

where $I(c_j = c_i^{NN})$ is a function which takes the value 1 if the class of the neighbor x_i^{NN} is equal to c_j.

Finally, the element will be classified as:

$$c_r = \arg\max_{c_j} P, \quad j = 1, ..., l \tag{7.4}$$

which is the class with the highest probability. The final result for each element is appended along with the calculated probabilities for each class in a result entry. Listing 7.1 showcases a four class classification where the element XYZ has been assigned to class C.

Listing 7.1: The element XYZ is classified to class C, which has the highest probability.

$XYZ|Result : C|A : 0.06|B : 0.03|C : 0.71|D : 0.2$

Regression For the case of regression, the final result for each element is calculated as described in Section 7.1.2 and contains its predicted value and has the following format (Listing 7.2):

Listing 7.2: The value estimation of element XYZ.

$XYZ|Result : 19.244469$

In both cases, the results for each query element are stored on HDFS in plain text.

Algorithm 17 presents the pseudocode of stage 3. During this stage, which consists of only a reduce operation, kNNs of each R element are fetched from the grouped set of $R_{k\times\alpha}$. Finally, for each query element either classification (Line 9) or regression (Line 11) is performed, after determining its final nearest neighbors (Line 7). The results are added to the resulting dataset (Line 9), which is then stored on HDFS (Line 12) in the proper format.

7.2.3.5 Spark implementation

We implemented a three-sessions and a single-session Spark version of the probabilistic classifier, named *S-kNN*, in order to conduct a comparative evaluation among the different distributed processing engines. The architecture of the implementation is the same as described above. The main difference lies to the transformations and actions that were used in order to achieve similar functionality to the Flink implementations. The code for both implementations is open source and it can be found online [dai].

7.2.3.6 Cost analysis

Zhang et al. [ZLJ12] showed that the overall communication cost of H-zkNN is $O(\alpha(1/\epsilon^2 + |S| + |R| + k|R| + nk))$, where α is the number of shifts, ϵ the sampling rate, n the number of partitions and k is the number of required nearest neighbors.

FML-kNN introduces some further communication within the stage 1 and between the stage 1 and 2. More specifically, $|R|$ and $|S|$ elements have to be communicated from the

Algorithm 17: FML-kNN (stage 3).

```
▷ The input is the grouped by element dataset emitted during the previous stage
Input: Datasets R, R_{k×α}
▷ The algorithm's results
Output: Dataset R_f
begin
    ▷ Initialization of the final dataset
    R_f = ∅
    foreach x ∈ R do
        RES ← kNN(x, R_{k×i})
        if classification then
            FIN ← classify(RES)
        else if regression then
            FIN ← regress(RES)
        R_f ← add(s, FIN)
    ▷ Store the final results on HDFS
    return R_f
```

initial `flatMap` of stage 1 to the sampling `flatMap`. Similarly, the same number of elements have to be sent to the mappers of the stage 2. There is no extra communication from the unification of stage 3, due to the removal of its mappers. Thus, the rest of the communication cost remains unchangeable. Consequently, the overall communication cost of our method is $O(\alpha(1/\epsilon^2 + 3|S| + 3|R| + k|R| + nk))$.

As far as the CPU cost is concerned, Zhang et al. [ZLJ12] estimate it to be $O(1/\epsilon^2 log(1/\epsilon^2) + nlog(1/\epsilon^2) + (|R| + |S|)log|S|)$ for H-zkNN. The FML-kNN introduces extra calculations in the stage 3, for classification and regression, which are $O(3k|R| + 2c|R|)$ and $O(2k|R|)$ respectively, where c is the number of classes for classification. However, since both are significantly less than $(|R|+|S|)log|S|$, the total CPU cost is finally equal to $O(1/\epsilon^2 log(1/\epsilon^2) + nlog(1/\epsilon^2) + (|R| + |S|)log|S|)$.

7.3 Experimental Evaluation

FML-kNN was assessed in terms of wall-clock completion time and scalability, by conducting a comparative benchmarking among similar implementations, executed over different distributed processing engines. The latter are either executed in one (where possible), or three Spark or Hadoop (i.e., an extension of the original H-zkNNJ algorithm) sessions and are compared with the corresponding versions of the probabilistic classifier. Similar results are expected for the regressor which we have omitted to avoid repetition.

We also present two case studies which exhibit the framework's efficiency over useful insights extraction from water consumption events on a city scale level. Throughout our experiments, we used one synthetic and two real-world water consumption time-series datasets.

7.3.1 Experimental setup

In the following, we present the environmental setting of the experiments and the qualitative and quantitative metrics that we used to assess the performance of the classification and regression processes. We also present the parameters that were used and how they were determined in the context of the experimentation process.

System The algorithms were assessed on a pseudo-distributed environment. The setup includes a system with 4 CPUs, each containing 8 cores clocked at 2.13GHz, 256 GB RAM and a total storage space of 900 GB. The CPUs support hyper-threading technology, running 2 separate threads per core. The total parallel capability of the system reaches the 64 threads ($4 \cdot 8 \cdot 2$). All (i.e., Flink-, Spark- and Hadoop-based) implementations were evaluated on the same HDFS, over a local *YARN* (Yet Another Resource Negotiator) resource manager. This way, each Flink task manager, Spark executor or Hadoop daemon runs on a different YARN container, represented by a separate Java process able to run one or more threads, simulating the distributed cluster.

Despite the significant differences in the distributed processing engines' configuration settings, they were all configured in order to use the same amount of system resources. The level of parallelism of all tasks for each engine was set to 16, in order to exploit the fact that, in an optimally determined setting and during the experiments, the stage 2 partitions the dataset into 8 subsets and the total number of shifts is 2. Thus, a maximum of 16 simultaneous tasks are executed in all cases. For Flink and Spark, a total of 4 task managers (one per CPU) and executors respectively were assigned 32768 MB of Java Virtual Machine (JVM) heap memory. Each task manager and executor was given a total of 4 processing slots (Flink) or executor cores (Spark). For Hadoop, the parallelism ability was set to 16 by assigning the mappers and the reducers to the same number of YARN containers, with 8192 MB of JVM each. Thus, the total amount of JVM heap memory assigned to each session is always 131072 MB (either $4 \cdot 32768$ MB, or $16 \cdot 8192$ MB).

Despite our attempt to assign similar amount of system resources to each distributed processing engine and due to the fact that each one offers a handful of configuration settings regarding execution, memory allocation, and job scheduling behavior, the performance of the different implementations may differ from the optimal one. However, after performing numerous benchmarking sessions, we believe that for the current system setting, the presented configuration is fair, achieving the highest possible performance for all three engines, while maintaining the level of parallelism at 16. The configuration was performed by taking into consideration the corresponding guide of each engine.

Parameters FML-kNN uses a variety of input parameters required by the underlying distributed kNN algorithm, in order to support the classification and regression processes. Regarding the value of the k parameter that was used throughout the experiments and due to the fact that the optimal value is problem specific, we performed a separate *cross-validation*-based evaluation for each of the case studies (see Section 7.3.4). The best k parameter choice was performed in a way that maintains the best balance between completion time and accuracy. Most parameters were similarly chosen after performing appropriate cross-validation-based benchmarking.

Among the rest of the parameters, FML-kNN utilizes a vector of size equal to the input

dataset's dimensions, indicating the weight of each feature according to its significance to the problem. Each feature is multiplied with its corresponding weight before the execution of the algorithm in order to perform the required scaling, according to the feature's importance. To automatically determine an optimal feature weighting vector, we provide the option of executing a genetic algorithm, which uses a specific metric, described in the next paragraph, as cost function. The parameters of the genetic algorithm, such as the size of the *population*, the probability to perform *mutation* or *crossover*, the *elite* percentage of the population and the number of the iterations can be directly determined by the user. This approach was applied to produce the optimal feature weighting vector for each of the cases that we studied.

Metrics Four different well known performance measures were used to evaluate the quality of the results obtained by the classifier and the regressor. These performance measures are included in the framework in order to offer the ability to assess the various data analysis tasks. *Accuracy* and *F-Measure* are implemented for classification, while *Root Mean Square Error* (RMSE) and *Coefficient of determination* (R^2) are used to evaluate the quality of regression. A short description of what each of these metrics represents in our experimentation is listed below:

- **Accuracy**: The percentage of correct classifications (values from 0 to 100). It indicates the classifier's ability to correctly guess the proper class for each element.
- **F-Measure**: The weighted average of *precision* and *recall* of classifications (values from 0 to 1). Using this metric, we ensure good balance between precision and recall, thus, avoiding misinterpretation of the accuracy.
- **Root Mean Square Error (RMSE)**: Standard deviation between the real and predicted values via regression. This metric has the same unit as the target variable. It provides us with the insight of how close the guessed values are to the real ones.
- **Coefficient of determination (R^2)**: Indicates the quality of the way the model fits the input data. It takes values from 0 to 1, with 1 being the perfect fit. A good fit means that the regressor is able to properly identify the variations of the training data.

The completion time comparison of the distributed processing engines is performed after simply obtaining the system time elapsed before and after each execution.

7.3.2 Datasets

For the experimental evaluation of FML-kNN we used two real-world water consumption related datasets coming from Switzerland and Spain. We also generated a synthetic dataset, based on an extended version of the Spain dataset, with a much larger amount of entries. Since the framework's algorithm is kNN-based, we normalized all the datasets' features from values ranging from 0 to 1, in order to avoid broader ranged features heavily affecting the result.

SWM dataset As in the previous chapters, this dataset is a courtesy of the DAIAD project (http://daiad.eu/). Specifically, it contains hourly time-series of 1000 Spanish households' water consumption, measured with smart water meters. It covers a time interval of a whole year, i.e., from the 1^{st} of July 2013 to the 30^{th} of June 2014. The records include a household identifier, a timestamp and a meter measurement in liters. This dataset has a total number of 8.7M records.

Shower dataset The second dataset is also provided by DAIAD and includes shower events from 77 Swiss households collected with shower water meters. Each record contains a household and shower identifier, a meter measurement in liters, the number of times that the faucet was turned off during the shower and the corresponding duration, the total duration of each shower as a whole, as well as the average water temperature and flow rate. It also contains demographic information related to the age, income, number of males or females and total number of household members. This dataset counts 5795 records. While this dataset is not on a Big Data scale, we perform a case study based on it, in order to assess the framework's data analysis capability on water consumption data regarding a single water fixture and consumer activity.

Synthetic dataset In order to evaluate completion time performance on Big Data, we created a synthetic dataset via a Big Data generator. The latter is a part of the *BigDataBench* benchmark suite [bdb] and operates via .xml files, in which the user can determine the number of records and their features. Based on an extended version of the SWM dataset produced after proper feature extraction (see Section 7.3.4.1), the synthetic dataset's entries consist of an id, 10 features and two target variables of continuous (floating point with 10 decimals) and binary data representation, respectively. Each feature is ranged from 1 to 99 and the id is alphanumeric. The data representation, number and range of features was selected in order to maintain a relatively high number of dimensions, while being able to create a large number of records and at the same time avoiding exceeding the memory limits of the setup (approx. 8192 MB per mapper or reducer for the Hadoop case). Thus, the total size of the synthetic dataset has 100M records (approx. 4.1 GB).

7.3.3 Benchmarking

In the following, we perform a comparative benchmarking of FML-kNN, in terms of scalability and wall-clock completion time, using the synthetic dataset and the probabilistic classifier. Similar results are expected for the regressor. The comparison involved:

- **FML-kNN (single session)**: The proposed implementation, presented in the previous section.
- **FML-kNN (three sessions)**: A three-sessions version of FML-kNN, where each stage is executed by a different Flink process.
- **S-kNN (single session)**: An Apache Spark version with the same architecture as FML-kNN.

- **S-kNN (three sessions)**: A three-sessions version of S-kNN, where each stage is executed by a different Spark process.
- **F-zkNN**: The single-session algorithm on which the core algorithm of FML-kNN was based.
- **H-zkNNJ**: An extended version of the algorithm to perform probabilistic classification executed in three separate sessions. This is the baseline method.

It is important to note that a single-session version of the H-zkNNJ algorithm is not possible, as a Hadoop session can only execute one map, followed by one reduce procedure. A single session requires the three stages to be executed in a sequential manner, which is not possible in Hadoop, as it would require mapping to be performed after reducing procedures several times.

7.3.3.1 Wall-clock completion time

Table 7.1 shows the probabilistic classifier's wall-clock completion time of all Flink, Spark and Hadoop versions, run in either three, or one sessions (possible only for FML-kNN, S-kNN and F-zkNN). The three-session F-zkNN is identical to the three-session FML-kNN implementation and its measurements are omitted. We used the synthetic dataset and we followed a 10%–90% testing–training split scheme, resulting in 10M records being used as the testing set (R) and 90M as the training set (S). It is apparent that the Flink-based implementations perform significantly better than all implementations, in both three and single-session versions. This improved performance is due to Flink's ability to process tasks in a pipelined manner, which allows the concurrent execution of successive tasks, thus, gaining in performance by compensating time spent in other operations (i.e., communication between the cluster's nodes).

The unified version implemented with Spark is significantly faster than the total wall-clock time of the corresponding three-sessions setting. This is due to the reduction of the I/O operations on HDFS during the beginning and end of each session. A critical role is also played by the omission of the mappers of stage 3, which introduced the additional overhead of forwarding the results of the second session to the proper reducers. For FML-kNN, the total time of the three-sessions version is similar to the unified one, again due to the pipelined way of execution, which compensates the time lost during HDFS I/O operations. The wall-clock completion time of S-kNN is only slightly lower for each stage than the baseline H-zkNNJ implementation, except during the third session, where it executes almost twice as fast. This is caused by the fact that the stage 3 does not include costly operations such as sorting and transforming the dataset coming from stage 2. The latter is partitioned according to the id of each query element (i.e., elements in R dataset), thus, taking advantage of all system resources and possibly Spark's documented better performance over Hadoop. F-zkNN performs worse than the rest of the implementations due to its resource-hungry propagation of the transformed dataset during the first stage.

It should be noted that the results do not suggest that Flink is superior to the other distributed processing engines. The comparison is limited to the current FML-kNN's core algorithm and the implementation and experimental setting are as fair as possible, considering the engines' different operational support. It proves, however, that our algorithm performs

Table 7.1: Wall-clock completion time

Version	3 Sessions				1 Session
	1st	2nd	3rd	Total	Total
FML-kNN	03m 45sec	24m 59sec	01m 48sec	30m 32sec	30m 25sec
S-kNN	07m 12sec	29m 15sec	03m 01sec	39m 28sec	33m 03sec
F-zkNN	N/A	N/A	N/A	N/A	46m 09sec
H-zkNNJ	06m 00sec	31m 19sec	05m 54sec	43m 13sec	N/A

significantly better with Flink and supports our choice to use it as an underlying execution environment.

7.3.3.2 Scalability

The various implementations were also evaluated in terms of scalability. Figure 7.5 shows the way each version scales in terms of completion time for subsets of the synthetic dataset of different size, using the same 10%–90% testing–training split scheme. The subset sizes varied from 10M (1M testing–9M training), to 100M (10M testing–90M training). As illustrated in the figure, the FML-kNN implementations exhibit similar performance and scale better as the dataset's size increases. However, the unified version, i.e., the one used at FML-kNN framework, has the advantage of not requiring user input and session initialization between the algorithm's stages. Flink's pipelined execution advantages are once again apparent if we compare the scalability performance of all the three-sessions implementations: The I/O HDFS operations cause the Spark and Hadoop versions to scale significantly worse than Flink, which performs similarly to the unified version. Finally, due to the aforementioned dataset propagation, F-zkNN scales worse than all the implementations.

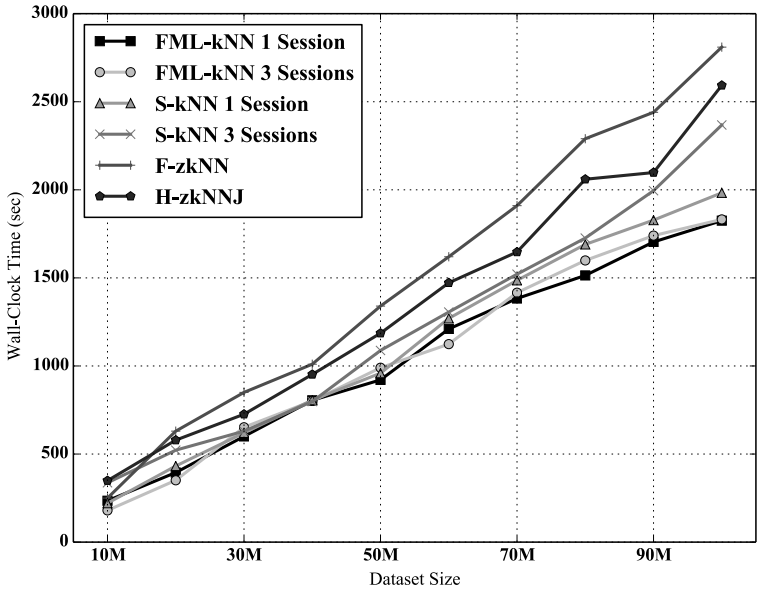


Figure 7.5: Scalability comparison for 10%–90% split.

We have executed the algorithm using the 100M synthetic dataset for different split schemes, more specifically for 10%–90%, 30%–70%, 50%–50%, 70%–30% and 90%–10% testing–

training. The results are depicted in Figure 7.6. FML-kNN performs better in all cases. Due to the fact that the CPU cost of all methods similarly depends on the size of R and S datasets (please refer to Section 7.2.3.6), there are no large variations between the differences in the performance among all the implementations, and for all split cases. More time is required to finish execution as the size of the R dataset is increased, due to the fact that the communication cost more heavily depends on the size of the R dataset. However, this is compensated in each split case by the reduced size of dataset S, causing the execution time to be increased in a similar to logarithmic manner.

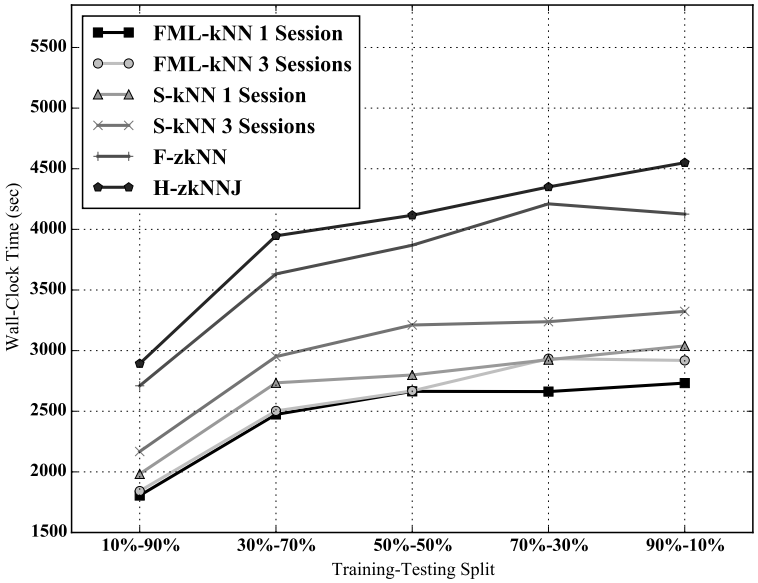


Figure 7.6: Wall-clock completion time for different split sizes.

7.3.4 Case studies

The framework's data analysis potential was evaluated using two real-world water related datasets in two independent case studies presented below. We focus on knowledge extraction from large volumes of historical water consumption data, towards the facilitation of useful insights acquisition for consumers and resource utilities in the water domain, which could induce lifestyle changes by promoting sustainability.

7.3.4.1 SWM dataset

The first case study showcases the potential of the framework's machine learning algorithms on a water consumption time-series forecasting task. Time-series data on water consumption pose several challenges on applying machine learning algorithms for forecasting purposes. Apart from the actual measurement values, one must take into account their correlation with previous values, their seasonality, the effect of contextual factors, etc. [HPC11]. Thus, proper features that represent and correlate different aspects of the data need to be defined in order to take advantage of the time-series nature of such data.

Feature extraction The dataset consists of 8.7M smart water meter hourly readings for 1000 households during a year, with each reading representing a record. The id of each element was set to be the smart meter id, together with the date and hour during which the consumption occurred. Initially, we removed any outlier records that could affect the final result. Such records belong to customers who do not exhibit normal water consumption behavior, i.e., they are frequently absent during the year or they continuously consume water (i.e. indication of possible leakage). We considered as frequently absent the users who did not consume any water for more than 40 days during a year, which indicates that they were away from their households. As a next step, we generated a total of nine different features for each data element.

Due to the qualitative nature of the extracted features, the Euclidean distance of FML-kNN cannot be applied directly on them. To overcome this, we assign each feature with a new value, which is derived during a pre-processing step, after we sort according to average water consumption. For example, for the feature indicating the season of the year, we obtain the average per season water consumption and we sort the seasons according to this value. Then, each season is assigned with a numerical value corresponding to its position in this sequence. This way, if supposedly winter was assigned with 1, summer with 2, autumn with 3 and spring with 4, we know that winter is closer to summer both in terms of consumption and Euclidean distance. The extracted features are presented in more details in the following.

- **Hour**: The hour during which the consumption occurred.
- **Time Zone**: We grouped the hours into four time zones of consumption: 1am - 4am (sleeping), 5am - 10am (morning activity), 11am - 7pm (working hours) and 8pm - 12am (evening activity).
- **Day of week**: The day of the week from Monday to Sunday.
- **Month**: The month of the year, from January to December.
- **Season**: The season of the year, from winter to autumn.
- **Weekend**: A binary feature indicating whether or not the consumption occurred during a weekend. We decided to include this feature after noticing differences between average hourly consumption during weekdays and weekends.
- **Customer group**: We run a *k-means* clustering algorithm, configured to run for time-series data, on the weekly average per-hour consumption time-series for each customer (for details, see below).
- **Customer ranking**: For each hour, we calculated the average hourly consumption of each customer and sorted according to it.
- **Customer class**: This feature represents a grouping of the customers to one of four classes according to their monthly average consumption, i.e. “environmentally friendly”, “normal”, “spendthrift”, “significantly spendthrift”.

Each record also contained two target variables, being the exact meter reading at each timestamp and a binary flag, indicating whether consumption occurred during that hour or not (i.e., if the meter reading was positive or zero).

The reason for choosing k-means for extracting the customer group feature was its significantly lower complexity compared to other clustering methods. As several evaluating works in the literature suggest [SS12, PSJC12, SG14], k-means achieves the best performance in terms of execution time and scalability. This is crucial in our case, as in a real world scenario the feature extraction procedure, as a pre-processing step is usually required to be as time efficient as possible. Additionally, k-means can produce accurate and tight clusters compared to similar methods [JKH14], which renders it the best choice for this case study.

Since the input data to be clustered are time-series, we used *Dynamic Time Warping* (DTW) as a distance metric. DTW is a very commonly used metric for comparing time series. As opposed to other distance metrics (e.g., Euclidean distance), it can handle the case where two time series have similar form but are slightly shifted in the time axis. For more details and a formal definition of DTW, the reader can refer to [YJF98]. Finally, we applied k-means on the average consumption time-series, with 10 as the number of clusters, which was determined as follows. Initially, we used a rule of thumb which provides a very fast estimation of the optimal number of clusters, described by Kodinariya et al. [KDM13], which calculates k as follows:

$$k = \sqrt{\frac{n}{2}} \tag{7.5}$$

where n is the number of elements to be clustered. Considering the number of the customers (1,000), the above equation yielded $k = 22$. Starting from this value, we run the k-means algorithm several times, using lower (higher) number of clusters than 22. We decreased (increased) the number of clusters by 1 and repeated the clustering and measured the *Davies-Bouldin index* [DB79] score of each execution. Finally, we chose the local best score, which in this case study was 10 clusters.

Procedure We first execute the probabilistic classifier, with the testing set (R) comprising of the last two weeks of water consumption for every user (approximately 336,000 records), in order to obtain the possibility of whether consumption will occur or not, during each hour. The rest of the dataset (approximately 8.36M records) is used as the training set (S). We perform binary classification, obtaining an intermediate dataset indicating whether or not consumption will occur for each training record. Using the hours during which we predicted that water will be consumed, we ran the regressor, obtaining a full predicted time-series result of water consumption for each user. Before each algorithm's execution, we determined the optimal scale vector using the genetic approach.

In order to choose the optimal k parameter for both algorithms, we employed a ten-fold *cross-validation* approach. We iteratively split the entire dataset into ten equal parts and executed each algorithm the same number of times, using a different subset as training set (S) while the rest of the sets, unified, comprised the testing set (R). As a metric, we used accuracy for classification and RMSE for regression. The k value that achieved the best balance between completion time and result quality was 15, for both the classifier and regressor.

Space filling curves accuracy evaluation FML-kNN supports three SFC-based solutions for reducing the dimensionality of the input data to just one dimension, namely the z-order, Grey code and Hilbert curves. We evaluated the completion time and approximation quality of each SFC, in order to choose the one that achieves the best balance between timing performance and approximation accuracy, regarding the water consumption dataset. Table 7.2

presents the metric and time performance related results of the probabilistic classifier and regressor for each SFC, which we obtained from running the algorithms using the ten-fold cross-validation.

Table 7.2: Space Filling Curves' performance.

Curve	Classification			Regression		
	Accuracy	F-Measure	Wall-Clock Time	RMSE	R^2	Wall-Clock Time
z-order Curve	70.24%	0.775	1m 20sec	18.86	0.64	0m 59sec
Hilbert Curve	70.54%	0.78	1m 32sec	18.69	0.66	1m 15sec
Gray-code Curve	70.4%	0.777	1m 25ec	18.81	0.64	1m 5sec

All three SFCs demonstrate similar performance in all aspects. Hilbert curve scores higher in all metrics as expected, however only slightly. Consequently, we chose the z-order curve for this case study, as it exhibits better time performance due to its decreased calculation complexity.

Results The classifier correctly predicted the 74.54% (F-Measure: 0.81) of the testing set's target variables, i.e., the hours during which some consumption occurred for the two-week period. For these specific hours the regressor achieved a RMSE score of 19.5 and a Coefficient of determination score of 0.69. The results were combined into a single file, forming the complete time-series of the dataset's last two weeks' (June 16-30 2014) water consumption prediction for all the users.

Figure 7.7 shows four users' consumption prediction versus the actual one, during four different days. The prediction for user #4 was close to reality. The results seem to follow the real values, but are not able to properly follow the observed ones. This indicates that it is rather difficult to accurately predict a single user's future water consumption, due to possible unforeseen or random events during a day, a fact which justifies the rather large RMSE score. For example, consumptions higher than 20 liters during an hour (e.g., user #3 around 6:00) could indicate a shower event, while larger consumptions (>50 liters) over more than one hour could suggest usage of the washing machine or dish washer (e.g., user #3 from 16:00 to 20:00), along with other activities. In order to assess the generalization of the results, we calculated the average RMSE of the hour and volume of the peak consumption during each day, as well as the average RMSE of the total water use per day, for all the predictions. The rather high errors, (8.89 hours, 28.9 liters and 132.23 liters respectively), confirm the previous observations regarding the difficulty in predicting random daily events. However, despite all the above, our algorithms' predictions are able to mostly follow the overall behavior during most days (e.g., users #3 and #1).

The results are particularly interesting if we aggregate the total predicted consumption of all users during each hour. The two upper diagrams in Figure 7.8 illustrate this case for two different days. It is apparent that our algorithms' predictions are able to properly follow the real aggregated consumption during each hour. This indicates that the negative effect of the sudden/unusual/unforeseen events is lost when we attempt to predict the total hourly water consumption of a larger number of users. The two lower diagrams present the same prediction performed by feeding our algorithms with already aggregated hourly consumption data. While the pre-aggregated dataset's predictions appear to less diverge from the actual values in certain points, the non-aggregated results are smoother and better assemble the

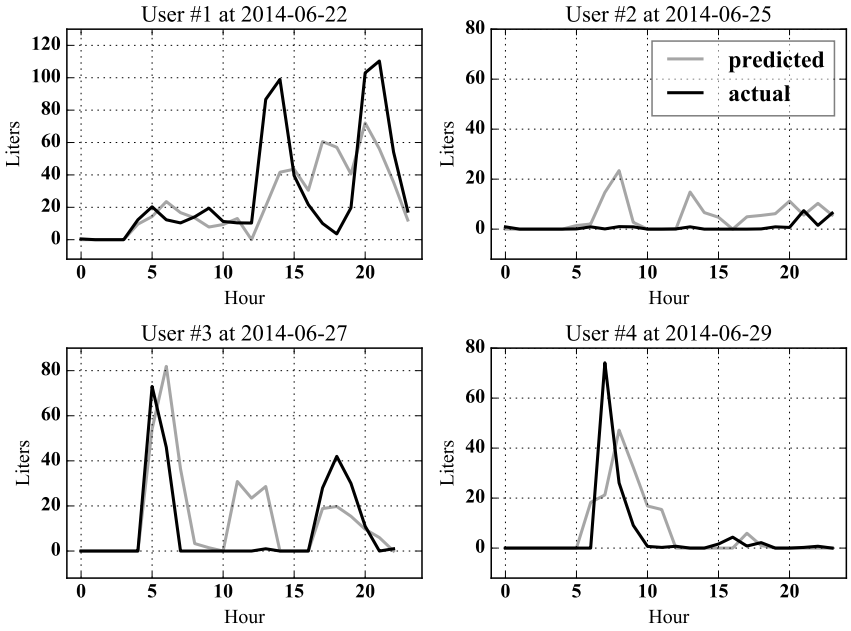


Figure 7.7: Personalised hourly water consumption prediction.

actual overall behavior. This is due to the fact that the non-aggregated dataset also contains user-specific features and is thus able to perform predictions based on user similarity, thus, yielding more accurate predictions.

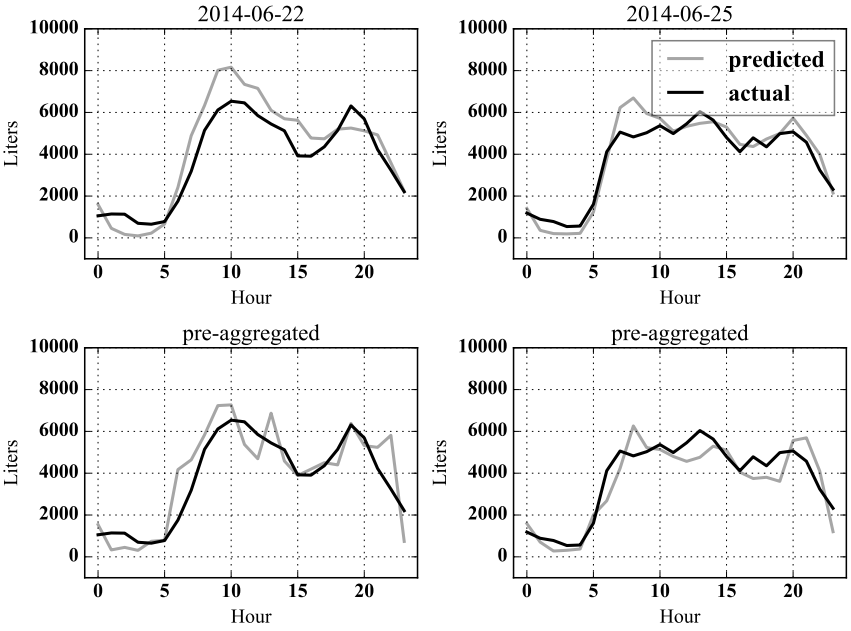


Figure 7.8: Aggregated hourly water consumption prediction.

7.3.4.2 Shower dataset

This case study exhibits the framework's potential in extracting useful knowledge from shower water consumption data, using the shower dataset. More specifically, the probabilistic classifier is used (the z-order curve is selected, similarly to the previous case study) in order to predict specific characteristics of a user, from shower water consumption events.

Feature extraction As mentioned, the dataset, apart from shower water consumption related data, also included demographic information. In this case study we focused on predicting the sex, age and income of the person that generates a shower event, using classification. For the case of sex, we used the shower events for which we knew whether the person was male or female (binary classification), i.e., households with only one, or only same sex inhabitants. Consequently, each record consisted of all the smart meter measurements (shower stops, break time, shower time, average temperature, average flow rate) as features and the sex as the target variable. Due to the dataset's rather small size, the age and income prediction was also performed by binary classification, i.e., determine whether the person that generates a shower event is of age less than 35 years or not, or has income less than 3000 CHF or not. The features of each record were the same in all three cases.

Procedure We ran the probabilistic classifier in order to perform binary classification of the shower events for each of the target variables. Ten-fold *cross-validation* was also used in this case. The latter, similarly to the previous case, helped us determine the optimal value of k parameter, which was set to 10.

Results The results obtained by the classification are illustrated in Figure 7.9. The classifier achieved cross-validation accuracy of 78.6%, 64.7% and 61.5% for sex, age and income prediction respectively. Despite the fact that a larger dataset would generate more confident results, it is safe to conclude to that predicting the age and income of a shower-taker based solely on her showers is a non-trivial task. On the other hand, sex is easier to predict as it directly affects the actions of a person during a shower.

7.4 Summary

In this chapter, we have presented a novel distributed processing framework, which supports a probabilistic classification and a regression approach. Its core algorithm is an extension of a distributed approximate kNN implementation, optimized to operate efficiently in a single distributed session. It was implemented with the Apache Flink scalable data processing engine.

We have conducted a detailed experimental evaluation in order to assess the framework in terms of wall-clock completion time and scalability, comparing it with similar approaches based on Apache Spark and Apache Hadoop. Our framework outperformed all competing implementations, managing to execute significantly faster on the same workloads and scale better for larger datasets, as a result of our optimizations and its ability to execute in a single distributed session. Furthermore, we performed two water consumption related real-world

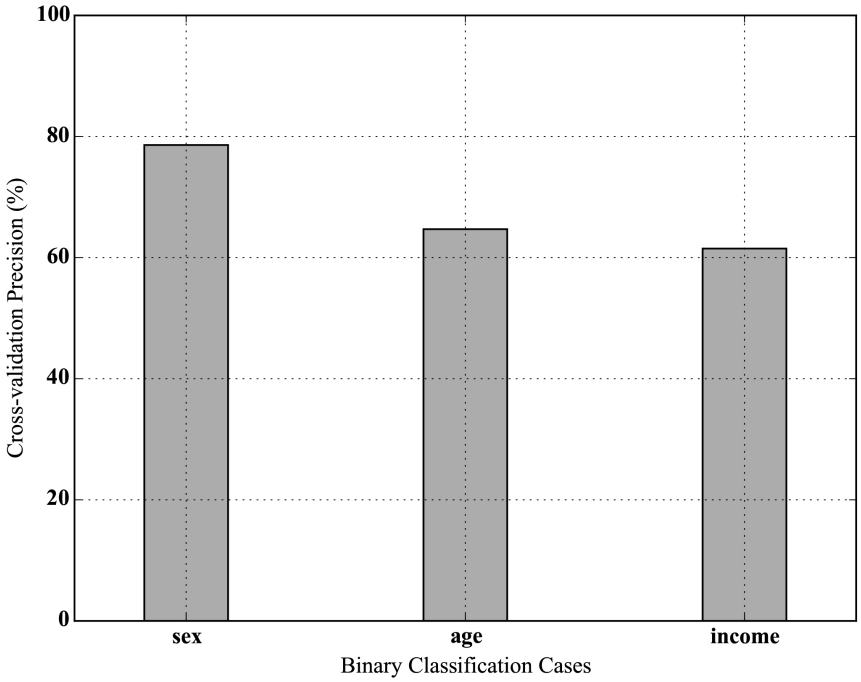


Figure 7.9: Cross-validation accuracy for sex, age and income prediction.

case studies, demonstrating our framework's potential in knowledge extraction tasks on data of very high volume.

The next chapter concludes this thesis and presents possible future directions.

Chapter 8

Conclusions and Future work

In this Thesis, we focused on scalable hybrid indexing, querying and exploration on big time series data, geolocated or not. We also developed a framework for efficient data analysis and knowledge extraction from Big Data. Section 8.1 concludes this work, while in Section 8.2 we present our main future directions.

8.1 Conclusions

Our main focus lies on *geolocated* time series, a novel data type that combines time series data with a spatial extend. First, we introduced the BTSR-tree index, a hybrid index for geolocated time series. Then, we proposed a variety of new hybrid queries that utilize BTSR-tree to apply certain thresholds both on the spatial and time series domain. To efficiently explore large datasets of geolocated time series, we developed two summarization approaches for efficient visual exploration, named bundle and tilemap summary. Next, we introduced the measure of local similarity, that considers two time series similar if the pairwise distance of their values per timestamp does not exceed a given threshold during a pre-defined time interval. Based on this new measure, we introduced two approaches for pair and bundle discovery on time series datasets. We also used local similarity for a new type of hybrid similarity search on large geolocated time series data, using the BTSR-tree index and a modified version of it, named $\mathcal{S}$BTSR-tree. Finally, we developed a novel distributed framework for analytics, named FML-kNN. The framework applies kNN joins on Big Data of various data types to allow efficient mining and knowledge extraction.

In more details, regarding hybrid indexing and querying:

- We introduced the *TSR-tree*, an extension of the R-tree spatial index. In the TSR-tree, each node is augmented with additional information corresponding to the bounds of the time series contained in its subtree, in addition to the standard *Minimum Bounding Rectangle* (MBR), denoting the spatial bound of its contents. Maintaining both kinds of bounds in each node allows to prune the search space simultaneously in the spatial and in the time series dimension while traversing the index. Thus, the number of required node accesses is significantly reduced, since we only retrieve the contents of nodes that may actually contain objects satisfying both types of predicates.
- We proposed the *BTSR-tree*, an optimized variant of *TSR-tree*, with its nodes having entries with more refined bounds by bundling together similar time series. This allows

to compute and maintain tighter bounds for each individual bundle, hence increasing the pruning effectiveness. To allow for a larger number of bundles in nodes at higher levels in the tree hierarchy, we exploit *Piecewise Aggregate Approximation* [KCPM01, YF00] to trade off between the number of bundles and the resolution of the bounding time series for each bundle.

- We utilized BTSR-tree to answer a variety of hybrid similarity queries on large geolocated time series datasets. To do so, we leveraged its hybrid indexing potential, allowing for more aggressive pruning in the spatial and time series domains simultaneously.

- We introduced the hybrid similarity join query that retrieves pairs of geolocated time series among two datasets such that both the distance between their locations and the distance between the time series themselves do not exceed certain given thresholds. We utilized the BTSR-tree index to speed up the computations and, since similarity join on time series is an inherently expensive procedure, we further proposed a space-driven data partitioning scheme that enables a parallel and distributed approach for hybrid similarity joins. Our method leverages hybrid indexing methods to efficiently handle similarity join queries locally within each partition. This is then combined with an optimization that minimizes the amount of data transferred between worker nodes at query time without false misses.

- We evaluated all the above on several real-world and synthetic datasets, assessing various metrics, such as node accesses, indexing size and build time, execution time and scalability.

Regarding our approaches on geolocated time series visual exploration:

- We introduced two geolocated time series summarization approaches for visual exploration, named *bundle* and *tile map summary*. These are supported and driven by two appropriate hybrid indices that speed up the result computation, providing efficient exploration of geolocated time series data. They consist of a spatial and a time series summary that jointly facilitate knowledge extraction and insight gaining. The spatial summary is similar for both and consists of MBRs of geolocated time series, according to a specific predicate (i.e., spatial proximity, or time series similarity). Each MBR is associated with a counter denoting the number of time series it contains

- Regarding the bundle summary, it consists of sets of MBTS, that is a band with upper and lower bounds that encloses all time series of a set, providing with a notion of a range of the time series values throughout the time axis. For providing prompt visualizations of summaries over geolocated time series data and minimizing latency when drawing the relevant graphic elements, we need early access to both spatial and time series information while traversing the index. For this purpose, we adapted our BTSR-tree index so as to also include *aggregates* per node, i.e., the number of time series pertaining to each bundle. Subsequently, we introduced a new traversal algorithm for efficient retrieval of a given number of bundles that are the most representative in the map area.

- Regarding the tile map summary it is driven by geo-iSAX, a hybrid index we introduced. It constitutes a hybrid variant of the iSAX index, augmented with spatial attributes of its nodes' children, to combine spatial and time series information. In each node, besides the SAX word that describes all its children time series, geo-iSAX keeps also the MBR that they form. To minimize the size and overlap of the MBRs, we proposed a spatial splitting policy, that instead of choosing the splitting dimension in a round-robin fashion (as in iSAX), it does so by selecting the dimension that produces the smallest overlap and overall size of the two generated MBRs. We introduced a traversal algorithm for applying timebox search on large (both vertically and horizontally) geolocated time series datasets. The traversal algorithm is applied on our geo-iSAX index and returns a tile map-like summary of the qualifying geolocated time series, by taking advantage of the SAX representation's properties.
- We evaluated our methods' efficiency, scalability, accuracy using real-world and synthetic datasets. We also assessed the quality of the information they provide, through mock-up visualization examples.

In the field of pair/bundle discovery and local similarity search:

- We introduced the measure of *local similarity*, that can be applied on co-evolving (i.e., time aligned) time series. Two co-evolving time series are locally similar if the pairwise distance of their values per timestamp does not exceed a given threshold during a time interval, that lasts at least a pre-defined number of consecutive timestamps.
- Based on local similarity, we introduced two methods for pair and bundle discovery on co-evolving time series datasets. Since discovering all possible pairs and bundles of locally similar time series within large sets is a computationally expensive process, we employed a value discretization approach that divides the value axis in ranges equal to the value difference threshold, in order to reduce the number of candidate pairs or bundles that need to be checked per timestamp. We also introduced a more aggressive filtering that only checks at selected *checkpoints* across time, but ensuring that no false negatives ever occur. To further reduce the number of examined candidates, we proposed a strategy that judiciously places these checkpoints across the time axis in a more efficient manner.
- We extended our previous approach on hybrid queries over geolocated time series to support local similarity, thus allowing more flexible and fine-grained queries and analyses. We introduced the *local similarity score* between two time series, which is defined as the maximum number of consecutive timestamps during which their respective values do not differ by more than a user-specified threshold. For evaluating such queries, we employed the BTSR-tree index. To further enhance the evaluation performance, we introduced an improvement to the BTSR-tree index, named $\mathcal{S}$BTSR-tree. It is based on temporally segmenting the time series bounds within each node and deriving tighter bounds per segment. Once the time series bounds in each node become more fine-grained, pruning the search space for local similarity queries proves much more effective.
- We evaluated the efficiency and scalability of our methods in terms of execution time, using real-world and synthetic datasets.

Finally, regarding scalable kNN joins:

- We introduced FML-kNN, a framework of methods for scalable management, analysis and mining on Big Data collections. The framework implements a probabilistic classifier and a regressor. Specifically, we introduced a MapReduce-based version of kNN joins, which reduces file operations for large amounts of data and is uniquely initialized upon launch. Our approach is unified in a single session to reduce space occupation and cluster overloading.
- We evaluated our framework on real-world and synthetic datasets against similar approaches, showing that the proposed method achieves high prediction precision and better scalability, while providing with useful knowledge extraction capabilities.

8.2 Future Work

In the following, we provide several possible future directions for our work, presented in this Thesis.

- We plan to expand the indexing capabilities of BTSR-tree on multi-dimensional feature spaces over distributed processing frameworks and also explore adaptivity to query workloads.
- Regarding visual exploration of geolocated time series, we will research and support more detailed visual analytics and identify more fine-grained patterns. An interesting direction would be to support drilling-down in a particular summarized result and discover whether there are differentiations in the distributions of its constituent, more detailed patterns, both in spatial and time series domains. Moreover, we will focus on supporting more complex time series distance measures that may boost the quality of our summaries.
- For pair and bundle discovery, we plan to further improve the scalability of our algorithms to extend their applicability over very large time series datasets, both in terms of cardinality, as well as in terms of length.
- Regarding local similarity search, we plan to study the applicability of $\mathcal{S}$BTSR-tree on various other hybrid query types, enlarging its potential in geolocated time series exploration.
- Finally, we will perform extended case studies using FML-kNN on more datasets from various sources, in order to establish our framework's ability in performing ad-hoc data mining tasks. Furthermore, we will explore its applicability on data stream mining applications, where the input is a continuous flow of data records. We will also enhance our framework's knowledge discovery capacity, by extending it with more distributed machine learning approaches, in an attempt to raise its potential on the continuously growing field of Big Data analytics.

Bibliography

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[ACC+14] Diego Raphael Amancio, Cesar Henrique Comin, Dalcimar Casanova, Gonzalo Travieso, Odemir Martinez Bruno, Francisco Aparecido Rodrigues, and Luciano da Fontoura Costa. A systematic comparison of supervised classifiers. *PloS one*, 9(4):e94137, 2014.

[Asc65] Jürgen Aschoff. Circadian rhythms in man. *Science*, 148(3676):1427–1432, 1965.

[ASP+05] Aleks Aris, Ben Shneiderman, Catherine Plaisant, Galit Shmueli, and Wolfgang Jank. Representing unevenly-spaced time series data for visualization and interactive exploration. In *IFIP Conference on Human-Computer Interaction*, pages 835–846. Springer, 2005.

[BCS16] Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *SIGMOD*, pages 1363–1375, 2016.

[bdb] Big data benchmark. http://prof.ict.ac.cn/bigdatabench/, accessed 15 november 2017.

[BGH+06] Ella Bingham, Aristides Gionis, Niina Haiminen, Heli Hiisilä, Heikki Mannila, and Evimaria Terzi. Segmentation and dimensionality reduction. In *SIAM*, pages 372–383, 2006.

[BGHW08] Marc Benkert, Joachim Gudmundsson, Florian Hübner, and Thomas Wolle. Reporting flock patterns. *Computational Geometry*, 41(3):111–125, 2008.

[BGM12] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.

[BK04] Christian Böhm and Florian Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems*, 6(6):728–749, 2004.

[BKD98] S. Berchtold and A. Keim Daniel. Indexing high-dimensional space: Database support for next decades's applications. In *Proc. of ACM SIGMOD*, 1998.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, pages 237–246, 1993.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[C+09] Fabrice Pierre Robert Colas et al. *Data mining scenarios for the discovery of subtypes and the comparison of algorithms*. Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, Leiden, The Netherlands, 2009.

[CCJW13a] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[CCJW13b] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. volume 6, pages 217–228, 2013.

[CDW+11] F. Chen, J. Dai, B. Wang, S. Sahu, M. Naphade, and C. Lu. Activity analysis based on low sample rate smart meters. In *Proc. of ACM SIGKDD*, pages 240–248, 2011.

[CF99] Kin-pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

[CGA16] Pantelis Chronis, Giorgos Giannopoulos, and Spiros Athanasiou. Open issues and challenges on time series forecasting for water consumption. In *EDBT/ICDT Workshops*, 2016.

[CHD+11] Maria Christoforaki, Jinru He, Constantinos Dimopoulos, Alexander Markowetz, and Torsten Suel. Text vs. space: efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[CJW09] Gao Cong, Christian S. Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[CKAS15] Georgios Chatzigeorgakidis, Sophia Karagiorgou, Spiros Athanasiou, and Spiros Skiadopoulos. A mapreduce based k-nn joins probabilistic classifier. In *Proc. of IEEE BigData*, pages 952–957, 2015.

[CKAS18] Georgios Chatzigeorgakidis, Sophia Karagiorgou, Spiros Athanasiou, and Spiros Skiadopoulos. Fml-knn: scalable machine learning on big data using k-nearest neighbor joins. *Journal of Big Data*, 5(1):4, 2018.

[CPS+18] Georgios Chatzigeorgakidis, Kostas Patroumpas, Dimitrios Skoutas, Spiros Athanasiou, and Spiros Skiadopoulos. Scalable hybrid similarity join over geolocated time series. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 119–128. ACM, 2018.

[CPS+19] Georgios Chatzigeorgakidis, Kostas Patroumpas, Dimitrios Skoutas, Spiros Athanasiou, and Spiros Skiadopoulos. Visual exploration of geolocated time series with hybrid indexing. *Big Data Research*, 15:12–28, 2019.

[CPSK10] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn J. Keogh. iSAX 2.0: Indexing and mining one billion time series. In *ICDM*, pages 58–67, 2010.

[CSM06] Yen-Yu Chen, Torsten Suel, and Alexander Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.

[CSP+14] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. *Knowl. Inf. Syst.*, 39(1):123–151, 2014.

[CSP+17] Georgios Chatzigeorgakidis, Dimitrios Skoutas, Kostas Patroumpas, Spiros Athanasiou, and Spiros Skiadopoulos. Indexing geolocated time series data. In *SIGSPATIAL*, pages 19:1–19:10, 2017.

[CSP+18] Georgios Chatzigeorgakidis, Dimitrios Skoutas, Kostas Patroumpas, Spiros Athanasiou, and Spiros Skiadopoulos. Map-based visual exploration of geolocated time series. In *Proceedings of the Workshops of the EDBT/ICDT 2018 Joint Conference (EDBT/ICDT 2018), Vienna, Austria, March 26, 2018.*, pages 92–99, 2018.

[CSP+19a] Georgios Chatzigeorgakidis, Dimitrios Skoutas, Kostas Patroumpas, Themis Palpanas, Spiros Athanasiou, and Spiros Skiadopoulos. Local pair and bundle discovery over co-evolving time series. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*, pages 160–169. ACM, 2019.

[CSP+19b] Georgios Chatzigeorgakidis, Dimitrios Skoutas, Kostas Patroumpas, Themis Palpanas, Spiros Athanasiou, and Spiros Skiadopoulos. Local similarity search on geolocated time series using hybrid indexing. In *SIGSPATIAL*, 2019.

[CSZ05] Richard Cole, Dennis Shasha, and Xiaojian Zhao. Fast window correlations over uncooperative time series. In *SIGKDD*, pages 743–749, 2005.

[CTFJ15] Yongjie Cai, Hanghang Tong, Wei Fan, and Ping Ji. Fast mining of a network of coevolving time series. In *SDM*, pages 298–306, 2015.

[CWR10] Ariel Cary, Ouri Wolfson, and Naphtali Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *SSDBM*, pages 87–95, 2010.

[CXGH08] Sye-Min Chan, Ling Xiao, John Gerth, and Pat Hanrahan. Maintaining interactivity while exploring massive time series. In *IEEE VAST*, pages 59–66, 2008.

[dai] Daiad repository. https://github.com/daiad/utility-flink-fml-knn, accessed 15 november 2017.

[DB79] D. L. Davies and D. W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, 1979.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DTS+08] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn J. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.

[DWD+15] Rui Ding, Qiang Wang, Yingnong Dang, Qiang Fu, Haidong Zhang, and Dongmei Zhang. Yading: Fast clustering of large-scale time series data. *Proc. VLDB Endow.*, 8(5):473–484, January 2015.

[EF13] Bahaeddin Eravci and Hakan Ferhatosmanoglu. Diversity based relevance feedback for time series search. *Proc. VLDB Endow.*, 7(2):109–120, 2013.

[EZPB18] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *PVLDB*, 12(2):112–127, 2018.

[Fal86] C. Faloutsos. Multiattribute hashing using gray codes. In *Proc. of ACM SIGMOD*, pages 227–238, 1986.

[FHR08] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[fli] Flink jobs and scheduling. https://ci.apache.org/projects/flink/flink-docs-master/internals/job_scheduling.html, accessed 21 december 2017.

[Gra95] Amara Graps. An introduction to wavelets. *IEEE Comput. Sci. Eng.*, 2(2):50–61, 1995.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[GvK06] Joachim Gudmundsson and Marc van Kreveld. Computing longest duration flocks in trajectory data. In *ACM GIS*, pages 35–42, 2006.

[GXK11] Jianping Gou, Taisong Xiong, and Yin Kuang. A novel weighted voting for k-nearest neighbor rule. *JCP*, 6(5):833–840, 2011.

[GZ14] Jack Galilee and Ying Zhou. A study on implementing iterative algorithms using big data frameworks. In *School of IT Research Conversazione Posters*, 2014.

[Haa10] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69(3):331–371, 1910.

[HHLM07] Ramaswamy Hariharan, Bijit Hore, Chen Li, and Sharad Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, page 16, 2007.

[HPC11] Lily A. House-Peters and Heejun Chang. Urban water demand modeling: Review of concepts, methods, and organizing principles. *Water Resources Research*, 47(5), 2011.

[HS99] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[HS03] Harry Hochheiser and Ben Shneiderman. Interactive exploration of time series data. In *The Craft of Information Visualization*, pages 313–315. Elsevier, 2003.

[HS04] Harry Hochheiser and Ben Shneiderman. Dynamic query tools for time series data sets: timebox widgets for interactive exploration. *Information Visualization*, 3(1):1–18, 2004.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of ACM STOC*, pages 604–613, 1998.

[JE10] Waqas Javed and Niklas Elmqvist. Stack zooming for multi-focus interaction in time-series data visualization. In *Visualization Symposium (PacificVis), 2010 IEEE Pacific*, pages 33–40. IEEE, 2010.

[JJHM14] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. M4: a visualization-oriented time series data aggregation. *Proceedings of the VLDB Endowment*, 7(10):797–808, 2014.

[JKH14] Yong Gyu Jung, Min Soo Kang, and Jun Heo. Clustering performance comparison using k-means and expectation maximization algorithms. *Biotechnology & Biotechnological Equipment*, 28(sup1):S44–S48, 2014.

[KAK95] Daniel A Keim, Mihael Ankerst, and Hans-Peter Kriegel. Recursive pattern: A technique for visualizing very large amounts of data. In *Proceedings of the 6th Conference on Visualization'95*, page 279. IEEE Computer Society, 1995.

[KCPM01] Eamonn J. Keogh, Kaushik Chakrabarti, Michael J. Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowl. Inf. Syst.*, 3(3):263–286, 2001.

[KDM13] Trupti Kodinariya and P.R. Dan Makwana. Review on determining of cluster in k-means clustering. 1:90–95, 01 2013.

[KDZP18] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. Coconut: A scalable bottom-up approach for building data series indexes. *PVLDB*, 11(6):677–690, 2018.

[KHS02] Eamonn Keogh, Harry Hochheiser, and Ben Shneiderman. An augmented visual query mechanism for finding patterns in time series data. In *International Conference on Flexible Query Answering Systems*, pages 240–250. Springer, 2002.

[KK11] Shrikant Kashyap and Panagiotis Karras. Scalable kNN search on vertically stored time series. In *SIGKDD*, pages 1334–1342, 2011.

[KMB+13] E. Kermany, H. Mazzawi, D. Baras, Y. Naveh, and H.gai Michaelis. Analysis of advanced meter infrastructure data of water consumption in apartment buildings. In *Proc. of ACM SIGKDD*, pages 1159–1167, 2013.

[KP99] Eamonn J Keogh and Michael J Pazzani. An indexing scheme for fast similarity search in large time series databases. In *SSDBM*, pages 56–67, 1999.

[LAB+09] Tim Lammarsch, Wolfgang Aigner, Alessio Bertone, Johannes Gärtner, Eva Mayr, Silvia Miksch, and Michael Smuc. Hierarchical temporal patterns and interactive aggregated views for pixel-based visualizations. In *Information Visualisation, 2009 13th International Conference*, pages 44–50. IEEE, 2009.

[Law00] J.K. Lawder. Calculation of mappings between one and n-dimensional values using the hilbert space-filling curve. Technical report, Birkbeck College, University of London, 2000.

[LK01] Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *ACM SIGMOD Record*, 30(1):19–24, 2001.

[LKWL07] Jessica Lin, Eamonn J. Keogh, Li Wei, and Stefano Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, 2007.

[LLL01] Swanwa Liao, Mario Lopez, and Scott T Leutenegger. High dimensional similarity search with space filling curves. In *Proc. of IEEE ICDE*, pages 615–622, 2001.

[LLZ+11] Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lun Lee, and Xufa Wang. IR-tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[LP18] Michele Linardi and Themis Palpanas. Scalable, variable-length similarity search in data series: The ulisse approach. *PVLDB*, 11(13):2236–2248, 2018.

[LZPK18] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. VALMOD: A suite for easy and exact detection of variable length motifs in data series. In *SIGMOD*, pages 1757–1760, 2018.

[MAK02] Mohamed F Mokbel, Walid G Aref, and Ibrahim Kamel. Performance of multi-dimensional space-filling curves. In *Proc. of ACM SIGSPATIAL*, pages 149–154, 2002.

[MFSW97] David Mintz, Terence Fitz-Simons, and Michelle Wayland. Tracking air quality trends with sas/graph. In *Proceedings of the 22nd Annual SAS User Group International Conference (SUGI'97)*, pages 807–812, 1997.

[MLVP17] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. New trends on exploratory methods for data analytics. *Proc. VLDB Endow.*, 10(12):1977–1980, 2017.

[MSF14] Yasuko Matsubara, Yasushi Sakurai, and Christos Faloutsos. Autoplait: Automatic mining of co-evolving time sequences. In *SIGMOD*, pages 193–204. ACM, 2014.

[NARS16] Rodica Neamtu, Ramoza Ahsan, Elke Rundensteiner, and Gabor Sarkozy. Interactive time series exploration powered by the marriage of similarity distances. *Proc. VLDB Endow.*, 10(3):169–180, 2016.

[NLKS11] M. Naphade, D. Lyons, C.A. Kohlmann, and C. Steinhauser. Smart water pilot study report. *IBM Research*, 2011.

[OSS01] R Keith Oswald, William T Scherer, and Brian L Smith. Traffic flow forecasting using approximate nearest neighbor nonparametric regression. *Center for Transportation Studies, University of Virginia*, 2001.

[Pal16] Themis Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In *SOFSEM*, pages 63–80, 2016.

[PFP18] Botao Peng, Panagiota Fatourou, and Themis Palpanas. Paris: The next destination for fast data series indexing and query answering. In *IEEE BigData*, 2018.

[PG15] John Paparrizos and Luis Gravano. k-shape: Efficient and accurate clustering of time series. In *SIGMOD*, pages 1855–1870, 2015.

[PG17] John Paparrizos and Luis Gravano. Fast and accurate time-series clustering. *ACM Trans. Database Syst.*, 42(2):8:1–8:49, 2017.

[PM02] Ivan Popivanov and Renée J. Miller. Similarity search over time-series data using wavelets. In *ICDE*, pages 212–221, 2002.

[PMT99] Dimitris Papadias, Nikos Mamoulis, and Yannis Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *SIGMOD*, pages 44–55, 1999.

[PSJC12] Sandeep Panda, Sanat Sahu, Pradeep Jena, and Subhagata Chattopadhyay. Comparing fuzzy-c means and k-means clustering techniques: a comprehensive study. *Advances in Computer Science, Engineering & Applications*, pages 451–460, 2012.

[PSP06] Spiros Papadimitriou, Jimeng Sun, and S Yu Philip. Local correlation tracking in time series. In *ICDM*, pages 456–465, 2006.

[QBM13] Shuyao Qi, Panagiotis Bouros, and Nikos Mamoulis. Efficient top-k spatial distance joins. In *SSTD*, pages 1–18, 2013.

[RB17] Kexin Rong and Peter Bailis. ASAP: Prioritizing attention via time series smoothing. *Proc. VLDB Endow.*, 10(11):1358–1369, 2017.

[RCM+12] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *SIGKDD*, pages 262–270, 2012.

[RGJN11] João B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[RSV02] Philippe Rigaux, Michel Scholl, and Agnés Voisard. *Spatial Databases with Application to GIS*. Morgan Kaufmann Publishers Inc., 2002.

[Sag12] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, New York City, 2012.

[SG14] Garima Sehgal and Dr Kanwal Garg. Comparison of various clustering algorithms. *International Journal of Computer Science and Information Technologies*, pages 3074–3076, 2014.

[SK08] Jin Shieh and Eamonn J. Keogh. *i*SAX: indexing and mining terabyte sized time series. In *SIGKDD*, pages 623–631, 2008.

[SLS+12] Christian Schwarz, Felix Leupold, Tobias Schubotz, Tim Januschowski, Hasso Plattner, and SAP Innovation Center. Rapid energy consumption pattern detection with in-memory technology. *Int'l Journal on Advances in Intelligent Systems*, 5(3 & 4), 2012.

[SMS10] Aleksandar Stupar, Sebastian Michel, and Ralf Schenkel. Rankreduce: processing k-nearest neighbor queries on top of mapreduce. In *Proc. of LSDS-IR*, pages 13–18, 2010.

[SRHM15] G. Song, J. Rochas, F. Huet, and F. Magoulès. Solutions for processing k nearest neighbor joins for massive data on mapreduce. In *Proc. of PDP*, pages 279–287, 2015.

[SS12] Nidhi Singh and Divakar Singh. Performance evaluation of k-means and heirarichal clustering in terms of accuracy and running time. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 4119–4121, 2012.

[SW13] R. Silipo and P. Winters. Big data, smart energy, and predictive analytics: Time series prediction of smart energy data. *KNIME.com*, 2013.

[TVK15] Pedro Sena Tanaka, Marcos R Vieira, and Daniel S Kaster. Efficient algorithms to discover flock patterns in trajectories. In *GeoInfo*, pages 56–67, 2015.

[VBT09] Marcos R Vieira, Petko Bakalov, and Vassilis J Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *SIGSPATIAL*, pages 286–295, 2009.

[WBM14] Eugene Wu, Leilani Battle, and Samuel R. Madden. The case for data visualization management systems: Vision paper. *Proc. VLDB Endow.*, 7(10):903–906, 2014.

[WK06] Li Wei and Eamonn Keogh. Semi-supervised time series classification. In *Proc. of ACM SIGKDD*, pages 748–753, 2006.

[Xu11] Jianhua Xu. Multi-label weighted k-nearest neighbor classifier with adaptive weight estimation. *Neural Information Processing*, pages 79–88, 2011.

[YAMP18] Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. Massively distributed time series indexing and querying. *TKDE (to appear)*, 2018.

[Yan99] Yiming Yang. An evaluation of statistical approaches to text categorization. *Information retrieval*, 1(1):69–90, 1999.

[YF00] Byoung-Kee Yi and Christos Faloutsos. Fast time sequence indexing for arbitrary Lp norms. In *VLDB*, pages 385–394, 2000.

[YJF98] Byoung-Kee Yi, HV Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 201–208. IEEE, 1998.

[YLK10] Bin Yao, Feifei Li, and Piyush Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *Proc. of IEEE ICDE*, pages 4–15, 2010.

[YZU+16] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Matrix profile i: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *ICDM*, 2016.

[YZU+18] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Zachary Zimmerman, Diego Furtado Silva, Abdullah Mueen, and Eamonn J. Keogh. Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile. *Data Min. Knowl. Discov.*, 32(1):83–123, 2018.

[ZCB11] Jian Zhao, Fanny Chevalier, and Ravin Balakrishnan. Kronominer: using multi-foci navigation for the visual exploration of time-series data. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1737–1746. ACM, 2011.

[ZCPB11] Jian Zhao, Fanny Chevalier, Emmanuel Pietriga, and Ravin Balakrishnan. Exploratory analysis of time-series with chronolenses. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2422–2431, 2011.

[ZIP14] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD*, pages 1555–1566, 2014.

[ZIP15] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. Rinse: Interactive data series exploration with ADS+. *Proc. VLDB Endow.*, 8(12):1912–1915, 2015.

[ZLJ12] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proc. of EDBT*, pages 38–49, 2012.

[ZMM14] Yu Zhang, Youzhong Ma, and Xiaofeng Meng. Efficient spatio-textual similarity join using mapreduce. In *WI*, pages 52–59, 2014.

[ZS02] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369, 2002.

[ZZ05] Min-Ling Zhang and Zhi-Hua Zhou. A k-nearest neighbor based algorithm for multi-label classification. In *2005 IEEE International Conference on Granular Computing*, volume 2, pages 718–721 Vol. 2, July 2005.