Indexing and analysis of very large masses of time series

Djamel-Edine Edine Yagoubi

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE MONTPELLIER

En Informatique

École doctorale I2S

Unité de recherche LIRMM, UMR 5506

INDEXATION ET ANALYSE DE TRES GRANDES MASSES DE SERIES TEMPORELLES

Présentée par Djamel Edine YAGOUBI
Le 19/03/2018

Sous la direction de Florent Masseglia, Reza Akbarinia et Themis Palpanas

Devant le jury composé de

Nadine Hilgert,	DR2, INRA, UMR MISTEA	Présidente du jury
Raja Chiky,	Professeure, ISEP, Laboratoire LISITE]	Rapportrice
Karine Zeitouni,	Professeure, Univ. Versailles St Quentin	Rapportrice
Reza Akbarinia,	CR1, Inria, Laboratoire Lirmm	Co-encadrant
Florent Masseglia,	CR1 HDR, Inria, Laboratoire Lirmm	Co-directeur
Themis Palpanas,	Professeur, Univ. Paris-Descartes & Institut Universitaire de France	Co-directeur
Dennis Shasha,	Professeur, Univ. New-York	Invité

UNIVERSITÉ DE MONTPELLIER

Résumé

Les séries temporelles sont présentes dans de nombreux domaines d'application tels que la finance, l'agronomie, la santé, la surveillance de la Terre ou la prévision météorologique, pour n'en nommer que quelques-uns. En raison des progrès de la technologie des capteurs, de telles applications peuvent produire des millions, voir des des milliards, de séries temporelles par jour, ce qui nécessite des techniques rapides d'analyse et de synthèse.

Le traitement de ces énormes volumes de données a ouvert de nouveaux défis dans l'analyse des séries temporelles. En particulier, les techniques d'indexation ont montré de faibles performances lors du traitement des grands volumes des données.

Dans cette thèse, nous abordons le problème de la recherche de similarité dans des centaines de millions de séries temporelles. Pour cela, nous devons d'abord développer des opérateurs de recherche efficaces, capables d'interroger une très grande base de données distribuée de séries temporelles avec de faibles temps de réponse. L'opérateur de recherche peut être implémenté en utilisant un index avant l'exécution des requêtes.

L'objectif des indices est d'améliorer la vitesse des requêtes de similitude. Dans les bases de données, l'index est une structure de données basées sur des critères de recherche comme la localisation efficace de données répondant aux exigences. Les index rendent souvent le temps de réponse de l'opération de recherche sous linéaire dans la taille de la base de données. Les systèmes relationnels ont été principalement supportés par des structures de hachage, B-tree et des structures multidimensionnelles telles que R-tree, avec des vecteurs binaires jouant un rôle de support. De telles structures fonctionnent bien pour les recherches, et de manière adéquate pour les requêtes de similarité. Nous proposons trois solutions différentes pour traiter le problème de l'indexation des séries temporelles dans des grandes bases de données. Nos algorithmes nous permettent

d'obtenir d'excellentes performances par rapport aux approches traditionnelles.

Nous étudions également le problème de la détection de corrélation parallèle de toutes paires sur des fenêtres glissantes de séries temporelles. Nous concevons et implémentons une stratégie de calcul incrémental des sketchs dans les fenêtres glissantes. Cette approche évite de recalculer les sketchs à partir de zéro. En outre, nous développons une approche de partitionnement qui projette des sketchs vecteurs de séries temporelles dans des sous-vecteurs et construit une structure de grille distribuée. Nous utilisons cette méthode pour détecter les séries temporelles corrélées dans un environnement distribué.

Titre en français

Indexation et analyse de très grandes masses de séries temporelles

Mots-clés

- Séries Temporelles
- Indexation
- Recherche de similarité

Abstract

Time series arise in many application domains such as finance, agronomy, health, earth monitoring, weather forecasting, to name a few. Because of advances in sensor technology, such applications may produce millions to trillions of time series per day, requiring fast analytical and summarization techniques.

The processing of these massive volumes of data has opened up new challenges in time series data mining. In particular, it is to improve indexing techniques that has shown poor performances when processing large databases.

In this thesis, we focus on the problem of parallel similarity search in such massive sets of time series. For this, we first need to develop efficient search operators that can query a very large distributed database of time series with low response times. The search operator can be implemented by using an index constructed before executing the queries. The objective of indices is to improve the speed of data retrieval operations. In databases, the index is a data structure, which based on search criteria, efficiently locates data entries satisfying the requirements. Indexes often make the response time of the lookup operation sublinear in the database size.

After reviewing the state of the art, we propose three novel approaches for parallel indexing and queryin large time series datasets. First, we propose DPiSAX, a novel and efficient parallel solution that includes a parallel index construction algorithm that takes advantage of distributed environments to build iSAX-based indices over vast volumes of time series efficiently. Our solution also involves a parallel query processing algorithm that, given a similarity query, exploits the available processors of the distributed system to efficiently answer the query in parallel by using the constructed parallel index.

Second, we propose RadiusSketch a random projection-based approach that scales nearly linearly in parallel environments, and provides high quality answers. RadiusS-

ketch includes a parallel index construction algorithm that takes advantage of distributed environments to efficiently build sketch-based indices over very large databases of time series, and then query the databases in parallel.

Third, we propose ParCorr, an efficient parallel solution for detecting similar time series across distributed data streams. ParCorr uses the sketch principle for representing the time series. Our solution includes a parallel approach for incremental computation of the sketches in sliding windows and a partitioning approach that projects sketch vectors of time series into subvectors and builds a distributed grid structure.

Our solutions have been evaluated using real and synthetics datasets and the results confirm their high efficiency compared to the state of the art.

Title in English

Massive distribution for indexing and mining time series

Keywords

- Time Series
- Indexing
- Similarity Search

Equipe de Recherche

Zenith Team, INRIA & LIRMM

Laboratoire

LIRMM - Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier

Adresse

Université Montpellier
Bâtiment 5
CC 05 018
Campus St Priest - 860 rue St Priest
34095 Montpellier cedex 5

Résumé Étendu

Introduction

Les séries temporelles sont présentes dans de nombreux domaines d'application (finance, agronomie, santé, surveillance de la Terre, prévisions météorologiques, etc.). Gérer et analyser des séries temporelles est crucial pour ces domaines, mais les exigences des ces applications sont très difficiles à satisfaire. La recherche de similarité entre séries temporelles est une clé pour effectuer de nombreuses tâches d'exploration de données telles que la découverte de shapelets, de motifs, la classification ou le clustering. Cette recherche doit donc être rapide et précise.

Afin d'améliorer les performances de ces requêtes de similitude, l'indexation est l'une des techniques les plus populaires, qui a été utilisée avec succès dans une variété d'applications. Malheureusement, créer un index sur des milliards de séries temporelles en utilisant des approches centralisées traditionnelles prend beaucoup de temps. De plus, une construction naïve de l'index sur l'environnement parallèle peut conduire à de mauvaises performances. La plupart des techniques traditionnelles ont mis l'accent sur : i) la représentation de données, avec des techniques de réduction de dimensionnalité telles que Discrete Wavelet Transform, Discrete Fourier Transform ou plus récemment Symbolic Aggregate Approximation ; et ii) des techniques de construction d'index. La malédiction de la dimensionnalité est le principal obstacle à l'analyse de ces données, et principalement à l'indexation des séries temporelles.

Bien que les méthodes d'indexation centralisées atteignent de bonnes performances par rapport à la recherche séquentielle, leurs performances se détériorent à mesure que la taille de la base de données augmente, ce qui pose des questions sur la fiabilité des méthodes centralisées. Pour traiter des données dont la taille peut aller jusqu'à des di-

zaines de téraoctets, une solution consiste à les distribuer sur plusieurs machines et les traiter en parallèle, on d'utiliser des frameworks parallèles, tels que MapReduce [19] ou Spark [78].

De la même manière, trouver les paires de séries temporelles hautement corrélées sur des fenêtres glissantes est utile pour de nombreuses applications dans les réseaux de capteurs, l'analyse financière ou la surveillance de réseaux de communication. Ce type de données nécessite des algorithmes avec des caractéristiques spécifiques. Ce problème a été étudié à travers les flux de données en utilisant des approches centralisées. La plupart des approches se concentre sur la réduction du temps de calcul des distance entre paires. Cependant, dans la littérature il n'y a pas de solution efficace pour l'indexation parallèle et l'interrogation de séries temporelles dans des environnements distribués.

État de L'art

Time Series

Une série temporelle peut être formellement définie comme suit :

Definition 1. *Une **série temporelle** est une suite d'observations d'une variable numérique v représentant l'évolution d'une quantité spécifique au cours du temps.*

$$T = \{(t_1, v_1), ..., (t_n, v_n)\}, v_i \in \mathbb{R} \tag{1}$$

Où $t_1 < t_2 < ... < t_n$.

La représentation générale *d'une base de données de séries temporelles* est une matrice D avec $m \times n$ éléments, où $t_i = (t_{i,1}, t_{i,2}, \cdots, t_{i,n})$ est un vecteur désignant la $i$ème série temporelle, et $t_{i,j}$ désigne le $j$ème valeur de la $i$ème série temporelle.

$$D = \begin{bmatrix} t_1 =< t_{1,1} & t_{1,2} \cdots & t_{1,j} & \cdots & t_{1,n} > \\ t_2 =< t_{2,1} & t_{2,2} \cdots & t_{2,j} & \cdots & t_{2,n} > \\ \vdots & \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ t_i =< t_{i,1} & t_{i,2} \cdots & t_{i,j} & \cdots & t_{i,n} > \\ \vdots & \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ t_m =< t_{m,1} & t_{m,2} \cdots & t_{m,j} & \cdots & t_{m,n} > \end{bmatrix}$$

Dans certaines applications, les séries temporelles peuvent être produites par un dispositif ou un processus qui génère des données en continu. Dans ce cas précis, il s'agit d'un flux de données. Ce concept peut être formalisé comme suit :

Definition 2. ***Un flux de série temporelle*** *est une série temporelle avec $n = \infty$, dont les points de données arrivent en continu à un taux arbitraire.*

Ce type de série temporelle est présent dans de nombreuses applications liées, par exemple, à la finance, au trafic internet, aux réseaux électriques, à des expériences scientifiques et aux capteurs.

Indexation des séries temporelles

Dans le contexte de l'exploration des séries temporelles, l'idée d'indexer des séries temporelles est pertinente pour toutes les techniques d'exploration de données, puisque l'indexation est une technique utilisée pour accélérer la recherche de similarité et l'accès aux données stockées. Même si plusieurs systèmes de gestion de bases de données ont été développés pour la gestion des séries temporelles (comme Informix Time Series [1], InfluxDB [2], OpenTSDB [3], et DalmatinerDB [4] basé sur RIAK), ils ne peuvent pas supporter efficacement les requêtes de recherche de similarité. Au fur et à mesure que les jeux de données se répandent dans une grande variété de contextes, nous devons relever le défi important de développer des méthodes d'indexation des séries temporelles plus efficaces. Généralement, les techniques d'indexation fonctionnent en deux étapes : d'abord, elles réduisent la dimensionnalité des séries temporelles en une représentation de faible dimension, et ensuite elles les indexent afin d'accélérer la recherche de similarité sur des séries chronologiques.

Réduction de dimensionnalité

Pour les très grandes bases de données de séries temporelles, il est important d'estimer très rapidement la distance entre deux séries temporelles. Ici, la forte dimensionnalité

[1] https ://www.ibm.com/developerworks/topics/timeseries
[2] https ://influxdata.com/
[3] http ://opentsdb. net/
[4] https ://dalmatiner.io/

des données de séries temporelles pose de véritables défis pour leur exploration et en particulier leur indexation. Dans la littérature, de nombreuses techniques ont été proposées qui représentent des séries temporelles avec une dimensionnalité réduite, puis appliquent une fonction de distance pour mesurer la similarité entre les séries temporelles transformées. Par exemple, Discrete Fourier Transformation (DFT) [23], Single Value Decomposition (SVD) [23], Discrete Wavelet Transformation (DWT) [41], Piecewise Aggregate Approximation (PAA) [36], Adaptive Piecewise Constant Approximation (APCA) [13], Chebyshev polynomials (CHEB) [10], Piecewise Linear Approximation (PLA) [16] et Symbolic Aggregate approXimation (SAX) [47]. Adaptive Piecewise Constant Approximation (APCA) et Piecewise Linear Approximation (PLA) [16] sont similaires à Piecewise Aggregate Approximation (PAA) [36], car elles divisent la série temporelle en segments. Les segments ont une taille fixe, et chaque segment est représenté par une ligne droite avec une certaine pente. La représentation APCA est une généralisation de PAA où chaque segment a une longueur arbitraire.

Dans [79], les auteurs propose une méthode de projection aléatoire basée sur des vecteurs aléatoires. L'idée de base est de multiplier chaque série temporelle par un ensemble de vecteurs aléatoires. Le résultat de cette opération est appelé un "sketch". Il est calculé pour chaque série temporelle. Ensuite, deux séries temporelle peuvent être comparées en comparant leurs sketches. Les sketches sont utilisés pour calculer une approximation de la distance entre chaque paire de séries temporelle. Les auteurs montrent que la projection aléatoire peut calculer l'approximation de différents types de distances comme la distance euclidienne et la distance L_p.

Indexation

Le problème de l'indexation de séries temporelles utilisant des solutions centralisées a été largement étudié dans la littérature, [6, 10, 23, 65, 12, 69]. Par exemple, dans [6], les auteurs propose TS-tree , une structure d'index pour une recherche efficace de similarité sur des séries temporelles. TS-tree fournit des résumés compacts des sous-arbres, réduisant ainsi l'espace de recherche très efficacement. Pour garantir une ventilation élevée, qui à son tour donne des arbres petits et efficaces, les entrées d'index sont quantifiées et leur nombre de dimensions est réduit. Dans [23], les auteurs utilisent des R*-trees pour localiser des séquences multidimensionnelles dans une collection de séries tempo-

relles. L'idée est de faire correspondre une séquence de grandes séries temporelles en un ensemble de rectangles multidimensionnels, puis d'indexer les rectangles à l'aide d'un R*-tree.

Dans [65], les auteurs proposent une représentation symbolique multirésolution appelée *indexable Symbolic Aggregate approXimation* (iSAX) qui est basée sur la représentation SAX [47]. L'avantage d'iSAX sur SAX est qu'il permet la comparaison de mots avec des cardinalités différentes, et même des cardinalités différentes dans un seul mot. iSAX peut être utilisé pour créer des indices efficaces sur de très grandes bases de données. La représentation SAX [47] est basée sur la représentation PAA [44] qui permet une réduction de la dimensionnalité tout en garantissant une propriété de limite inférieure. L'idée de PAA est d'avoir une taille de segment fixe, et minimiser la dimensionnalité en utilisant les valeurs moyennes sur chaque segment.

Dans [11], une version améliorée d'iSAX, appelée iSAX 2.0, a été utilisée pour indexer plus d'un milliard de séries temporelles. Il s'agit d'exploiter deux couches tampon différentes, à savoir First Buffer Layer (FBL) et Leaf Buffer Layer (LBL), pour stocker des parties de l'index et des séries temporelles en mémoire avant de les stocker dans le disque. Dans le FBL, toutes les séries temporelles qui se retrouveront dans le même sous-arbre iSAX 2.0 sont groupées et peuvent croître jusqu'à occuper toute la mémoire principale disponible, puisqu'elles n'ont pas de restriction de taille. Le LBL correspond aux nœuds feuilles de l'arbre iSAX et il est utilisé pour rassembler toutes les séries temporelles de nœuds feuilles et les vider sur le disque. les auteurs utilise également une technique efficace pour diviser un nœud feuille lorsque sa taille est supérieure à un seuil.

Dans [12], les auteurs proposent deux extensions d'iSAX 2.0, à savoir iSAX 2.0 Clustered et iSAX2 +. Ces extensions se concentrent sur la gestion efficace des données de séries temporelles brutes pendant le processus de chargement en masse, en utilisant une technique qui utilise des tampons de mémoire principale pour regrouper et router des séries temporelles similaires dans l'arborescence, en effectuant l'insertion de manière efficace.

Dans [80], au lieu de construire l'index complet iSAX2+ sur l'ensemble de données complet et d'interroger seulement plus tard, les auteurs proposent de construire de manière adaptative des parties de l'index, uniquement pour les parties des données concernées par les requêtes.

Dans [69], en se basant sur la représentation de PCA [13] les auteurs proposent

DSTree qui découpe des séries temporelles en segments de longueurs variables mais adaptatifs à la forme de la série. DSTree utilise l'écart-type pour définir les limites inférieures. Contrairement à iSAX qui ne prend en charge que le fractionnement horizontal, et où seules les valeurs moyennes peuvent être utilisées dans le fractionnement, DSTree utilise des stratégies de fractionnement multiples et fournit plus de manières possibles de diviser les fichiers de feuilles de séries temporelles.

Indexation de séries temporelles dans des systèmes distribués

Bien que les méthodes de recherche centralisées permettent des gains de temps de plusieurs ordres de grandeur par rapport au balayage séquentiel, leurs performances se détériorent à mesure que la taille de la base de données augmente, ce qui pose des questions sur la capacité de ces méthodes centralisées à passer à l'échelle. Pour faire face à l'augmentation du volume des données, une solution prometteuse est d'exploiter des frameworks parallèles, tels que MapReduce [19] ou Spark [78], pour créer de puissantes unités de calcul et de stockage à l'aide de machines ordinaires.

Dans [43], les auteurs proposent une structure d'index parallèle évolutive pour le traitement de données relationnelles. Cela fait partie d'un framework d'indexation pour les systèmes MapReduce [19] et est extensible en termes de types de données et de requêtes avec plusieurs types d'index. L'index est organisé en arborescence et stocké en tant que fichier séquentiel dans le système de fichiers HDFS [62]. Le fichier d'index est également partitionné en plusieurs blocs, chacun contenant les données d'un certain nombre de sous-index et certains blocs d'index sont chargés sélectivement dans la mémoire. Pour créer un index, il faut passer par un nouveau job MapReduce. Dans la phase "map", les mappeurs analysent les données et génèrent des fichiers intermédiaires, enregistrant la façon dont les données sont distribuées en différents sous-espaces. Dans la phase "reduce", chaque reducer collecte les fichiers intermédiaires à partir des mappeurs pour un sous-espace spécifique et construit un index local. Les résultats constituent un ensemble de sous-index, qui sont collectés par le nœud maître pour la construction de l'index global.

Cependant, aucune des solutions ci-dessus n'est appropriée pour l'indexation de séries temporelles et la recherche de similarité. Dans l'ensemble, dans la littérature, il n'existe pas d'index de séries chronologiques conçu/adapté pour fonctionner dans des

environnements distribués.

Contributions

L'objectif de cette thèse est de développer des nouvelles techniques d'indexation séries temporelles dans des environnements massivement distribués. Nos contributions sont les suivantes.

Indexation et interrogation de séries temporelles massivement distribuées avec DPiSAX. Dans ce travail, nous proposons DPiSAX, une nouvelle solution efficace et parallèle qui comprend un algorithme de construction d'un indice parallèle qui profite des environnements distribués pour construire efficacement des indices basés sur iSAX sur de grands volumes de séries temporelles. Notre solution implique également un algorithme parallèle de traitement de requêtes qui, étant donné une requête de similarité, exploite les processeurs disponibles du système distribué pour répondre efficacement à la requête en utilisant l'index parallèle construit. Notre proposition a été évaluée à l'aide de grands volumes de données réelles et de jeux de données synthétiques (jusqu'à 4 milliards de séries chronologiques, pour un volume total de 6 To). Les résultats expérimentaux illustrent l'excellente performance de DPiSAX, ce qui confirme l'efficacité de notre approche.

RadiusSketch : indexation massivement distribuée de séries temporelles. Dans ce travail, nous proposons une approche basée sur la projection aléatoire (sketch). Les temps de réponse de cette approche, en fonction de la taille des données, évoluent presque linéairement dans des environnements parallèles, et elle fournit des réponses de grande précision. RadiusSketch inclut un algorithme de construction d'index parallèle qui tire parti des environnements distribués pour construire efficacement des index basés sur des sketches à partir de très grands volumes de séries temporelles, et un algorithme de traitement de requête parallèle qui exploite ces index. Nous illustrons la performance de notre approche, sur des jeux de données réels et synthétiques de 1 Téraoctets et 500

millions de séries chronologiques. La méthode de sketch, telle que nous l'avons mise en œuvre, est supérieure tant en qualité qu'en temps de réponse par rapport à l'approche de référence de la littérature.

ParCorr : identification efficace de paires de séries temporelles similaires en parallèle. Dans ce travail, nous identifiions les paires de séries temporelles fortement corrélées dans les flux de données distribués. Nous proposons ParCorr, une solution parallèle efficace pour détecter des séries temporelles similaires sur des fenêtres glissantes. ParCorr utilise le principe des sketches pour représenter la série temporelles. Il comprend une approche parallèle pour le calcul incrémental des sketches dans les fenêtres glissantes et une approche de partitionnement qui projette les sketches dans des sous-vecteurs et construit une structure de grille distribuée. Notre proposition a été évaluée en utilisant des ensembles de données réels et synthétiques et les résultats confirment l'efficacité et le comportement presque linéaire de la solution proposée par rapport à l'état de l'art.

Publications

- Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Themis Palpanas. DPiSAX : Massively Distributed Partitioned iSAX. ICDM 2017 : IEEE International Conference on Data Mining, Nov 2017, New Orleans, United States. pp.1-6, 2017.
- Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Dennis Shasha. RadiusSketch : Massively Distributed Indexing of Time Series. DSAA 2017 : IEEE International Conference on Data Science and Advanced Analytics, Oct 2017, Tokyo, Japan. pp.1-10, 2017

Organisation de la Thèse

La suite de cette thèse est organisé ainsi :

Le chapitre 2, nous passons en revue l'état de l'art. Il est divisé en trois sections principales : Dans la section 2.1, nous donnons un aperçu général des principales techniques

d'exploration de données en environnement centralisé. En particulier, nous traitons trois problèmes : la classification supervisée, la classification non supervisée, et la découverte de motifs. Nous introduisons également le problème des mesures de similarité. Dans la section 2.2.1, nous présentons les techniques de représentation des séries temporelles et les solutions qui ont été proposées pour traiter le problème de la haute dimensionnalité des séries temporelles. Dans la section 2.2, nous traitons du problème de l'indexation des séries temporelles et discutons des méthodes et des techniques proposées dans la littérature. Plus précisément, nous nous intéressons à l'indexation des séries temporelles et au calcul distribué, qui feront l'objet des chapitres 3 et 4.

Dans le chapitre 3, nous traitons du problème de l'indexation et de l'interrogation de séries temporelles dans des bases de données très volumineuses. Dans les sections 3.2 et 3.3, nous proposons deux solutions parallèles pour l'indexation et l'interrogation des données de séries temporelles. Dans la section 3.4, nous évaluons l'efficacité de notre approche en effectuant des expériences approfondies avec de très grands ensembles de données réelles et synthétiques.

Dans le chapitre 4, nous proposons une solution pour résoudre le problème de l'indexation et de l'interrogation d'ensembles massifs de séries temporelles. Dans la section 4.1, nous proposons RadiusSketch qui est une approche basée sur la projection aléatoire pour l'indexation parallèle et l'interrogation des séries chronologiques. Dans la section 4.2, nous validons notre proposition à travers des expériences étendues et différentes en utilisant des ensembles de données très réels et synthétiques.

Dans le chapitre 5, nous traitons le problème de trouver les paires de séries temporelles fortement corrélées sur les fenêtres glissantes. Dans la section 5.2, nous proposons une solution parallèle pour détecter des séries temporelles similaires sur des fenêtres glissantes. Dans la section 5.4, nous validons notre approche en effectuant diverses expériences approfondies en utilisant des ensembles de données réelles et synthétiques massives avec de très grandes dimensions. Enfin, dans la section 5.5, nous résumons notre travail et nous discutons d'autres améliorations potentielles.

Contents

Chapter 1

Introduction

1.1 Context

Time series occur in many application domains like economy, medical surveillance, weather forecasting, biology, agronomy, earth monitoring, hydrology, genetics and musical querying. This results in the production of very large time series datasets [34, 63, 76, 56, 58, 27, 60, 50, 51]. Mining these datasets is crucial for the underlying applications, but the exhibit characteristics of time series data make their analysis particularly difficult. With such complex and massive sets of time series, fast and accurate similarity search is a key to perform many data mining tasks like Motifs Discovery, Classification or Clustering [56]. The problem of high dimensionality in time series data is the main obstacle for time series data mining, and mainly for defining a form of similarity measure based on human perception.

In order to improve the performance of similarity queries over time series, the indexing is one of the most popular techniques [22], which has been successfully used in a variety of settings and applications [23, 66, 6, 69, 12, 80]. Unfortunately, creating an index over billions of time series by using traditional centralized approaches is highly time-consuming. Moreover, a naive construction of the index in the parallel environments may lead to poor querying performances. Most state of the art indices have focused

on: i) data representation, with dimensionality reduction techniques such as Discrete Wavelet Transform (DWT), Discrete Fourier Transform (DFT), or more recently Symbolic Aggregate Approximation (SAX); and ii) index building techniques, considering the index as a tree that calls for optimal navigation among sub-trees and shorter traversals.

An appealing opportunity for improving the index construction time is to take advantage of the computing power of distributed systems and parallel frameworks such as MapReduce[19] or Spark [78]. With billions of time series distributed on a cluster computing nodes, querying and mining principles cannot rely on traditional techniques and call for dedicated algorithms and a deep combination of parallelism and indexing techniques.

Similarly, finding the highly correlated pairs of time series on sliding windows is useful for many applications such as sensor fusion, financial trading, or communications network monitoring, to name a few. This type of data requires algorithms with specific characteristics. The problem has been studied across data streams using centralized approaches [49, 48, 73, 61, 54, 53, 17, 52]. Most of them focus on reducing the computation time of the pairwise distance computation. However, in the litterature there is no efficient solution for parallel indexing and querying of time series in distributed environments.

1.1.1 Contributions

The objective of this thesis is to develop new indexing techniques allowing fast data access, and answering fundamental primitives such as KNN queries. Our main contributions are as follows.

Massively Distributed Time Series Indexing and Querying with DPiSAX. In this work [74], we propose DPiSAX, a novel and efficient parallel solution that includes a parallel index construction algorithm that takes advantage of distributed environments to build iSAX-based indices over vast volumes of time series efficiently. Our solution also involves a parallel query processing algorithm that, given a similarity query, ex-

ploits the available processors of the distributed system to efficiently answer the query in parallel by using the constructed parallel index. Our proposal has been evaluated using large volumes of the real world and synthetic datasets (up to 4 billion time series, for a total volume of 6TBs). The experimental results illustrate the excellent performance of DPiSAX, which confirms the effectiveness of our approach.

RadiusSketch: Massively Distributed Indexing of Time Series. In this work [75], we propose a random projection-based approach that scales nearly linearly in parallel environments, and provides high quality answers. RadiusSketch includes a parallel index construction algorithm that takes advantage of distributed environments to efficiently build sketch-based indices over very large volumes of time series, and a parallel query processing algorithm, which given a query, exploits the available processors of the distributed system to answer the query in parallel by using the constructed index. We illustrate the performance of our approach, on real and synthetic datasets of up to 1 Terabytes and 500 million time series. The sketch method, as we have implemented, is superior in both quality and response time compared with the state of the art approach.

ParCorr: Efficient Parallel Methods to Identify Similar Time Series Pairs Across Sliding Windows. In this work, we study the problem of finding the highly correlated pairs of time series in distributed data streams. We propose ParCorr, an efficient parallel solution for detecting similar time series across sliding windows. ParCorr uses the sketch principle for representing the time series. It includes a parallel approach for incremental computation of the sketches in sliding windows and a partitioning approach that projects sketch vectors of time series into subvectors and builds a distributed grid structure. Our proposal has been evaluated using real and synthetics datasets and the results confirm the efficiency and almost linear scalability of the proposed solution compared to the state of the art.

1.1.2 Publications

- Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Themis Palpanas. DPiSAX: Massively Distributed Partitioned iSAX. IEEE International Conference on Data Mining (ICDM), New Orleans, United States. pp.1-6, 2017.
- Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Dennis Shasha. RadiusSketch: Massively Distributed Indexing of Time Series. IEEE International Conference on Data Science and Advanced Analytics (DSAA), Tokyo, Japan. pp.1-10, 2017

1.1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

In Chapter 2, we review the state of the art. It is divided into three main sections: In Section 2.1, we give a general overview of the main data mining techniques in the centralized environment. In particular, we deal with three methods: classification, clustering, Motif Discovery. Also, we introduce the problem of similarity measures. In Section 2.2.1, we introduce the time series representation techniques and solutions that have been proposed to deal with the problem of high dimensionality in time series data. In Section 2.2, we deal with the problem of time series indexing, and discuss the methods and techniques that have been proposed in the literature. Specifically, we focus on time series indexing and distributed computing, which will be the subject of Chapters 3 and 4.

In Chapter 3, we deal with the problem of time series indexing and querying in very large databases. In Sections 3.2 and 3.3, we propose two parallel solutions for indexing and querying time series data. In Section 3.4, we assess the efficiency of our proposed approach by carrying out extensive experiments with very large real-world and synthetic datasets.

In Chapter 4, we propose an our solution to deal with the problem of indexing and querying massive sets of time series. In Section 4.1, we propose RadiusSketch that is

random projection-based approach for parallel indexing and querying time series. In Section 4.2, we validate our proposal through extensive, different experiments using very real-world and synthetic datasets.

In Chapter 5, we deal with the problem of finding the highly correlated pairs of time series over sliding windows. In Section 5.2, we propose a parallel solution for detecting similar time series across sliding windows. In Section 5.4, we validate our approach by carrying out various extensive experiments using very massive with very large real-world and synthetic datasets. Finally, in Section 5.5, we summarize our work and we discuss potential further improvements.

Chapter 2

State of the Art

In this chapter, we introduce the basics and the necessary background of this thesis. The chapter is organized as follows. First, we formally define time series. Then, we introduce the problem of similarity search in time series datasets, and discuss the main existing techniques and methods that have been proposed for this problem. Afterwards, we focus on time series representation and indexing, and discuss the most efficient approaches proposed in the literature.

2.1 Time Series Data Mining

There has been an increasing amount of research over the last two decades on time series data mining, knowledge discovery and applications. Consequently, many data mining algorithms have been developed like : classification, clustering, associations, forecasting, anormality detection, frequent pattern discovery, etc. Time series can be produced in many applications, such as economy, medical surveillance, climate forecasting, biology, hydrology, genetics, and musical querying. This results in the production of large volumes of time series that challenge knowledge discovery [56, 51, 50]. With such complex and massive sets of time series, fast and accurate similarity search is a key to perform many data mining tasks.

In this section, we define the time series, and discuss briefly the main techniques that have been proposed for mining time series datasets.

2.1.1 Time Series

A time series can be formally defined as follows:

Definition 3. *A time series T is a sequence of n real-values obtained from some process over time. All values in the sequence are ordered in time and stored together with a timestamp.*

$$T = \{(t_1, v_1), ..., (t_n, v_n)\}, v_i \in \mathbb{R} \tag{2.1}$$

Where $t_1 < t_2 < ... < t_n$.

For the ease of presentation, a time series can also be represented by the values v_i that it takes.

Example 1. *Figure 2.1 shows an example of a time series from an earthquake occurred few kilometers from Greve in Chianti, Italy [1]. Figure 2.2 shows an example of conversion from the RGB image color histograms, to a time series. This time series can be used to apply image matching, to object the characterization of shapes [70].*

The general representation of a *time series database* is a matrix D with $m \times n$ elements, where $t_i = (t_{i,1}, t_{i,2}, \cdots, t_{i,n})$ is a vector denoting the ith time series, and $t_{i,j}$ denotes the jth value of ith time series.

$$D = \begin{bmatrix} t_1 =< t_{1,1} & t_{1,2} \cdots & t_{1,j} & \cdots & t_{1,n} > \\ t_2 =< t_{2,1} & t_{2,2} \cdots & t_{2,j} & \cdots & t_{2,n} > \\ \vdots & \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ t_i =< t_{i,1} & t_{i,2} \cdots & t_{i,j} & \cdots & t_{i,n} > \\ \vdots & \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ t_m =< t_{m,1} & t_{m,2} \cdots & t_{m,j} & \cdots & t_{m,n} > \end{bmatrix}$$

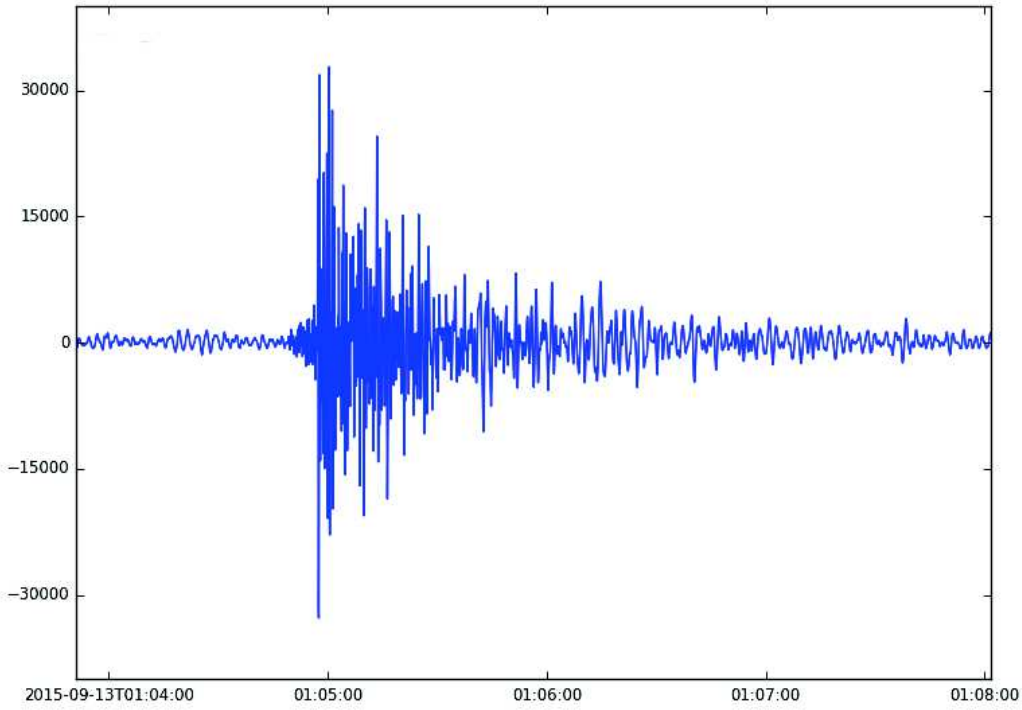


Figure 2.1 – Examples of time series data relative to seismic signal from an earthquake occurred few kilometers from Greve in Chianti, Italy [1]

In some applications, the time series can be produced through a device or process that is continuously generating data. In this specific case, it is called *streaming time series*, or a *data stream*. This concept can be formalized as flows.

Definition 4. *A time series stream X is a time series with $n = \infty$, whose data points arrive continuously at an arbitrary rate.*

This type of time series is present in many application, e.g., finance, communication networks, internet traffic, online transactions in the financial market or retail industry, electric power grids, industry and production processes, scientific and engineering experiments and remote sensors. We can transform *timeseriesstreams* into static time series by defining an endpoint (or sliding window) to the time series. Then, a traditional or classical data mining algorithm can be applied to the truncated time series.

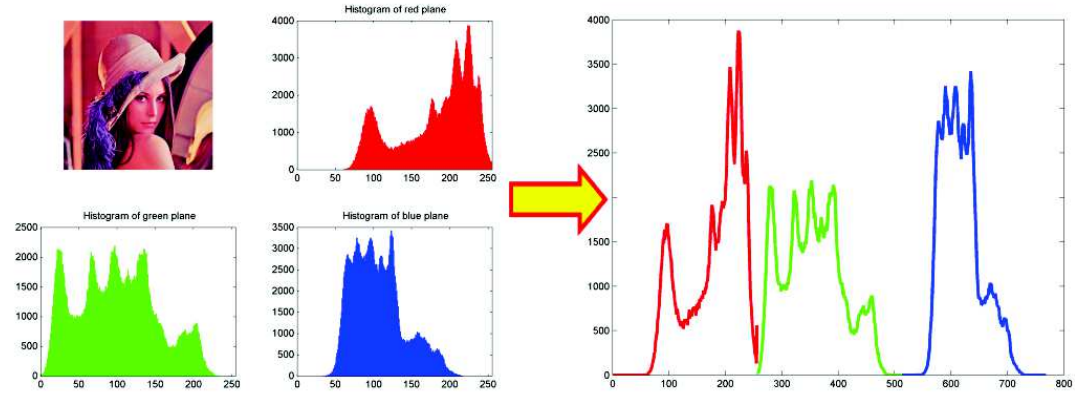


Figure 2.2 – An image data converted to time series from image color histograms.

2.1.2 Data Mining Techniques And Similarity Measurements

Data Mining [72, 26] presents a core step of a knowledge discovery process. Therefore, mining time series has attracted a lot of attention during the last decade because of the increasing production of time series data. Nowadays, several techniques have been developed and applied to time series data [22], e.g. , clustering [38], classification [32], indexing [23, 66, 6, 69, 12, 80] , motif discovery [45], etc. In most of these tasks, the similarity measurement is a central sub task.

In this section, we provide a brief overview of the main tasks that have attracted extensive research interest in time series data mining, including *clustering*, *classification* and *motif discovery*. Then, we discuss time series similarity measuring approaches.

2.1.2.1 Classification

The classification has become one of the most important data mining tasks. Over the last few decades there have been a large number of data classification algorithms proposed in the literature. However, Time series classification problem is different from the traditional classification problem. In [22] the authors define two types of classification for time series datasets. The most frequent type is the classification of time series defined as follows. Given sets of time series with a label for each set, the task consists in training a

classifier and labeling new time series. The second type is time point classification, the task is to classify each point in one of the classes. The classifier learns from the time points of the training set and given a new time series.

2.1.2.2 Clustering

Clustering is a traditional data mining task useful in many different domains like seismology studies, finance, economics, communication, automatic control, and online services [7, 20, 40]. The clustering techniques are used to group different time series to produce similar clusters. In [37, 57] the authors defined the clustering as finding natural groupings of the time series in a database under some similarity or dissimilarity measure. Therefore, other definition must be taken into account. In [57] the clustering is defined as the unsupervised version of classification since the instances are not previously labeled with class. The community of data mining is still working on this topic since each application has its own requirements, and no general approach can efficiently solve the problems of all applications. Time series clustering results are often illustrated by *dendrograms*. These tree diagrams graphically represent the result of a hierarchical clustering algorithm. Figure 2.3 shows the results of hierarchical clustering over 12 time series in a dendrogram.

2.1.2.3 Motif Discovery

Motif discovery is another typical task in the field of time series data mining, and consists in finding the subsequences, frequent patterns, or motifs that appear recurrently in a longer time series. This problem was first proposed in [46]. The idea of motifs was transferred from symbolic gene time series in bioinformatics to numerical time series [22]. Generally, there are two types of motif discovery: univariate and multivariate. The goal of univariate motif discovery is to find repeated subsequences in one single and longer time series or a database of time series. Multivariate motif discovery aims to find motifs that span across different time series for the same time range.

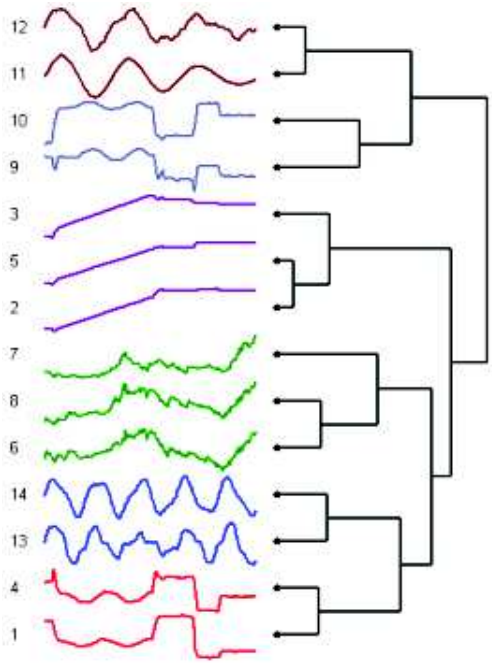


Figure 2.3 – Example of a time series dendrogram [57]

2.1.2.4 Similarity Measurements

The measurement of similarity between two time series is an essential subroutine in time series analysis and data mining tasks. In fact, There are over a dozen distance measures like Euclidean distance [23], Dynamic Time Warping [21], Edit Distance with Real Penalty [15], distance based on Longest Common Subsequence [68], etc. In [22], four categories of similarity measures are defined in order to calculate the similarity of the time series: 1) *Feature-based* distances that extract the features that usually describing time independent aspects of the series that are compared with static distance functions; 2) *Shape-based* distances that compare the overall appearance of the time series; 3) *Model-based* distances that fit a model to the data and measure the similarity by comparing the models; 4) *Compression-based* distances that analyze how well time series can be compressed alone and together. Figure Figure 2.4 shows the categories of similarity measures with some examples.

The Euclidean distance (ED) is one of the most straightforward similarity measure-

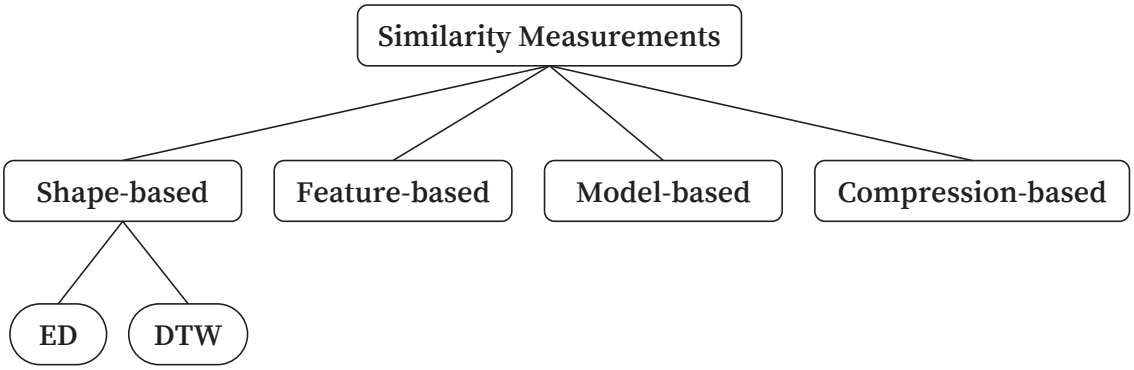


Figure 2.4 – Categorization of similarity measures.

ment methods used in time series analysis, and has been widely used [36, 47, 65, 66]. Given two time series $X = \{x_1, ..., x_n\}$ and $Y = \{y_1, ..., y_m\}$ such that $n = m$, the Euclidean distance between X and Y is defined as [23]:

$$ED(X, Y) = \sqrt{\sum_{n}^{i=1}(x_i - y_i)^2} \tag{2.2}$$

However, the Euclidean distance is an effective measurement of similarity between two time series [37, 59]. The weaknesses of the Euclidean distance is that it cannot be applied to time series of different lengths and it doesn't handle outliers or noise [23]. Figure 2.5 shows an example of the Euclidean distance between two time series X and Y.

Another popular similarity measure is the Dynamic Time Warping algorithm (DTW) [5]. DTW compares time series of different length, as it uses both many-to-one point and one-to-many point comparisons. Given two time series $X = \{x_1, ..., x_n\}$ and $Y = \{y_1, ..., y_m\}$ of length n and m, respectively, DTW starts by building the distance matrix $D_m \in \mathbb{R}^{n \times m}$ where :

$$D_m \in \mathbb{R}^{n \times m} : d_{i,j} = (x_i - y_j)^2, 1 \leq i \leq n, 1 \leq j \leq m \tag{2.3}$$

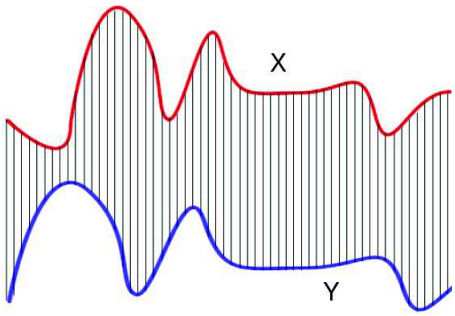


Figure 2.5 – X and Y are two time series of a particular variable v. The Euclidean distance results in the sum of the point-to-point distances (gray lines), along all the time series.

The alignment path is found by calculating the shortest warping path in the matrix of distances between all pairs of time points. However, the optimal path $W = \{w_1, w_2, ..., w_l\}$ is the path that minimizes the warping cost :

$$DTW(X, Y) = min\{ \sqrt{\sum_{k=1}^{l} w_k} \tag{2.4}$$

where w_k is the matrix element (i, j). This warping path can be calculated using a dynamic programming approach [8] that evaluates the following recurrence.

$$f(i, j) = \| x_i - y_j \| + min(f(i, j-1), f(i-1, j), f(i-1, j-1)) \tag{2.5}$$

where $\| x_i - y_j \|$ is the distance found in the current cell, and $f(i, j)$ is the cumulative distance of $\| x_i - y_j \|$ and the minimum cumulative distances from the three adjacent cells.

In [66], the authors show that DTW is significantly more accurate than the Euclidean distance for small datasets, and the difference diminishes as the datasets get larger until there is no measurable difference.

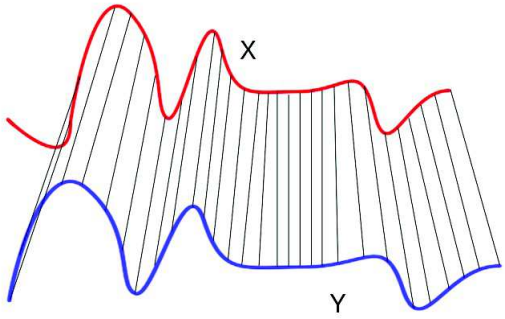


Figure 2.6 – X and Y are two time series of a particular variable v and different length. The Dynamic Time Warping distance takes into account the warping of the time axis of both series in order to better align the two time series.

2.2 Time Series Indexing

In the context of time series data mining, the idea of indexing time series is relevant to all data mining techniques, since indexing is a technique used to speed up similarity search and access to stored data. Even though several database management systems have been developed for the management of time series (such as Informix Time Series[1], InfluxDB[2], OpenTSDB[3], and DalmatinerDB[4] based on RIAK), they do not include similarity search indexes, focusing on (temporal) SQL-like query workloads. Thus, they cannot efficiently support similarity search queries. As the datasets grow more prevalent in a wide variety of settings, we face the significant challenge of developing more efficient time series indexing methods. Generally, the indexing techniques consist of two steps: at first, they reduce the dimensionality of time series into some low-dimensional representation and secondly index them in order to speedup the similarity search over time series.

[1]https://www.ibm.com/developerworks/topics/timeseries
[2]https://influxdata.com/
[3]http://opentsdb.net/
[4]https://dalmatiner.io/

2.2.1 Dimensionality reduction

For very large time series databases, it is important to estimate the distance between two time series very quickly. The high dimensionality of time series data poses big challenges to time series data mining and especially indexing. In the literature, many techniques have been proposed that represent time series with reduced dimensionality, and then apply a distance function to measure the similarity between transformed time series. For example, Discrete Fourier Transformation (DFT) [23], Single Value Decomposition (SVD) [23], Discrete Wavelet Transformation (DWT) [41], Piecewise Aggregate Approximation (PAA) [36], Adaptive Piecewise Constant Approximation (APCA) [13], Chebyshev polynomials (CHEB) [10], Piecewise Linear Approximation (PLA) [16] and Symbolic Aggregate approXimation (SAX) [47]. This latter takes the PAA representation as an input and discretizes it into a small alphabet of symbols as we will show later. Figure 2.7 shows an example of such techniques that can significantly reduce the time and space.

In [23], Faloutsos et al. use DFT-based method to reduce dimensionality. The basic idea is that any complex time series or signal can be expressed in terms of cosine waves. Each time series can be represented using complex numbers called the Fourier Coefficients. DFT representation is probably one of the first dimensionality reduction techniques known in the literature.

Based on the Discrete Fourier Transform [23], the authors in [41] have developed the Discrete Wavelet Transform DWT. The DWT can uniquely represent the time series by a wavelet transform.

Adaptive Piecewise Constant Approximation (APCA) [13] and Piecewise Linear Approximation (PLA) [16] are similar to Piecewise Aggregate Approximation (PAA) [36], since they divide the time series into frames. In the former, the frames have fixed size, and each frame is represented by a straight line with a certain slope. The APCA representation is a generalization of PAA where each frame has arbitrary length.

In [79], Zhu et al. propose random projection method based on random vectors. The basic idea is to multiply each time series with a set of random vectors. The result of that

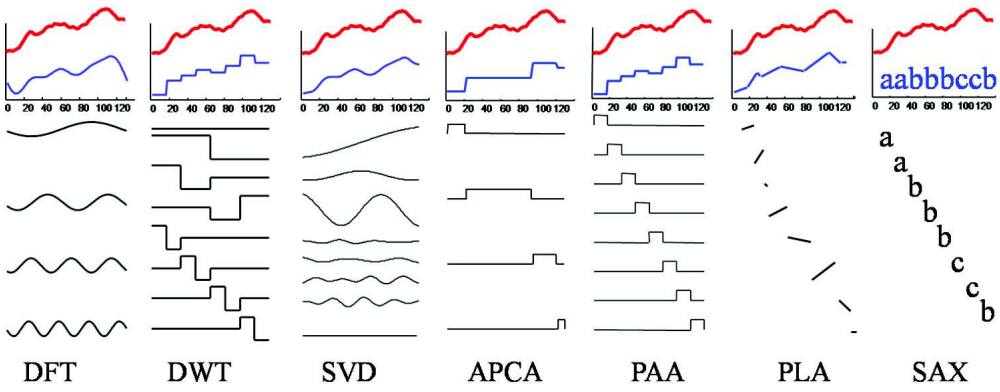


Figure 2.7 – Example of techniques that can significantly reduce the dimensionality of time series [57]

operation is a "sketch" for each time series consisting of the distance (or similarity) of the time series to each random vector. Then two time series can be compared by comparing sketches. The sketches are used to approximate the distance between each pair of time series. The authors show that the random projection can approximate different types of distances like Euclidean Distance and L_p Distance. The sketch approach is a kind of Locality Sensitive Hashing [25], by which similar items are hashed to the same buckets with high probability. In particular, the sketch approach is similar in spirit to SimHash [14], in which the vectors of data items are hashed based on their angles with random vectors.

2.2.2 Indexing

The problem of indexing time series using centralized solutions has been widely studied in the literature, *e.g.*, [6, 10, 23, 65, 12, 69]. For instance, in [6], Assent et al. propose the TS-tree (time series tree), an index structure for efficient retrieval and similarity search over time series. The TS-tree provides compact summaries of subtrees, thus reducing the search space very effectively. To ensure high fanout, which in turn results in small and efficient trees, index entries are quantized and dimensionally reduced.

In [10], Cai et al. use Chebyshev polynomials as a basis for dealing with the problem

of approximating and indexing d-dimensional trajectories and time series. They show that the Euclidean distance between two d-dimensional trajectories is lower bounded by the weighted Euclidean distance between the two vectors of Chebyshev coefficients, and use this fact to create their index.

In [3], Agrawal et al. use R-Tree to index the first few DFT coefficients of each time series. In [55], the authors propose an optimization of this later by considering the symmetric property of Fourier coefficients.

In [23], Faloutsos et al. use R*-trees to locate multi dimensional sequences in a collection of time series. The idea is to map a large time series sequence into a set of multi-dimensional rectangles, and then index the rectangles using an R*-tree.

In [79], Zhu et al. give a review of a simple and easy to maintain multidimensional index structure, called grid structure. This index structure can be used for indexing higher-dimensional data. The grid structure can be stored in d-dimensional array in the main memory. To use grid structure as an index, we need to apply dimensionality reduction techniques, such as random projection. In [4], the authors propose a new grid-based indexing approach called grid-based Datawise Dimensionality Reduction (DDR). They use DDR dimensionality reduction approach and apply quantization to construct a grid-based index structure.

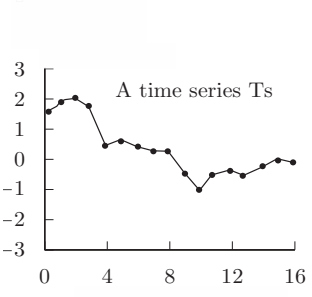
In [65], Shieh et Keogh propose a multiresolution symbolic representation called indexable Symbolic Aggregate approXimation (iSAX) which is based on the SAX representation [47]. The advantage of iSAX over SAX is that it allows the comparison of words with different cardinalities, and even different cardinalities within a single word. iSAX can be used to create efficient indices over very large databases. The SAX representation [47] is based on the PAA representation [44] which allows for dimensionality reduction while providing the important lower bounding property as we will show later. The idea of PAA is to have a fixed segment size, and minimize dimensionality by using the mean values on each segment. Example 2 gives an illustration of PAA.

Example 2. *Figure 2.8b shows the PAA representation of X, the time series of Figure 2.8a. The representation is composed of $w = |X|/l$ values, where l is the segment size. For each segment, the set of values is replaced with their mean. The length of the final representation*

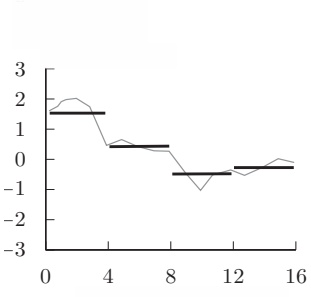
w is the number of segments (and, usually, w << |X|).

The SAX representation takes as input the reduced time series obtained using PAA. It discretizes this representation into a predefined set of symbols, with a given cardinality, where a symbol is a binary number. Example 3 gives an illustration of the SAX representation.

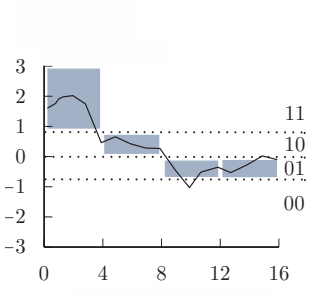
Example 3. *In Figure 2.8c, we have converted the time series X to SAX representation with size 4, and cardinality 4 using the PAA representation shown in Figure 2.8b. We denote SAX(X) = [11, 10, 01, 01].*



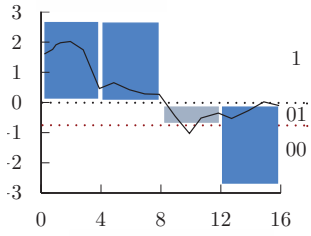
(a) A time series X of length 16



(b) A PAA representation of X, with 4 segments



(c) A SAX representation of X, with 4 segments and cardinality 4, $[11, 10, 01, 01]$.



(d) An iSAX representation of X, with 4 segments and different cardinalities $[1_2, 1_2, 01_4, 0_2]$.

Figure 2.8 – A time series X is discretized by obtaining a PAA representation and then using predetermined break-points to map the PAA coefficients into SAX symbols

The iSAX representation uses a variable cardinality for each symbol of SAX representation, *i.e.*, each symbol is accompanied by a number that denotes its cardinality. We

denote the iSAX representation of time series X by $iSAX(X)$ and we call it the iSAX word of the time series X. For example, the iSAX word shown in Figure 2.8d can be written as $iSAX(X) = [1_2, 1_2, 01_4, 0_2]$.

The lower bounding approximation of the Euclidean distance for iSAX representation $iSAX(X) = \{x'_1, ..., x'_w\}$ and $iSAX(Y) = \{y'_1, ..., y'_w\}$ of two time series X and Y is defined as:

$$MINDIST(iSAX(X), iSAX(Y)) = \sqrt{\frac{n}{w}} \sqrt{\sum_{w}^{i=1} (dist(x'_i, y'_i))^2}$$

where the function $dist(x'_i, y'_i)$ is the distance between two iSAX symbols x'_i and y'_i. The lower bounding condition is formulated as:

$$MINDIST(iSAX(X), iSAX(Y)) \leq ED(X, Y)$$

Using a variable cardinality allows the iSAX representation to be indexable. We can build a tree index as follows. Given a cardinality b, an iSAX word length w and leaf capacity th, we produce a set of b^w children for the root node, insert the time series to their corresponding leaf, and gradually split the leaves by increasing the cardinality by one character if the number of time series in a leaf node rises above the given threshold th.

Example 4. *Figure 2.9 illustrates an example of iSAX index where each iSAX word has 2 symbols and a maximum cardinality of 4. The root node has 2^2 children while each child node forms a binary sub-tree. There are three types of nodes: root node, internal nodes (N2, N5, N6, N7) and terminal nodes or leaf nodes (N3, N4, N8, N9, N10, N11, N12, N13). Each leaf node links to a disk file that contains the corresponding time series. The maximum number of time series in each leaf file is defined by a threshold th.*

The previous studies have shown that the iSAX index is robust with respect to the choice of parameters (word length, cardinality, leaf threshold) [66, 12, 81]. Moreover, it can also be used to answer queries with the Dynamic Time Warping (DTW) distance, through the use of the corresponding lower bounding envelope [39].

In [11], an improved version of iSAX, called iSAX 2.0, has been used to index more

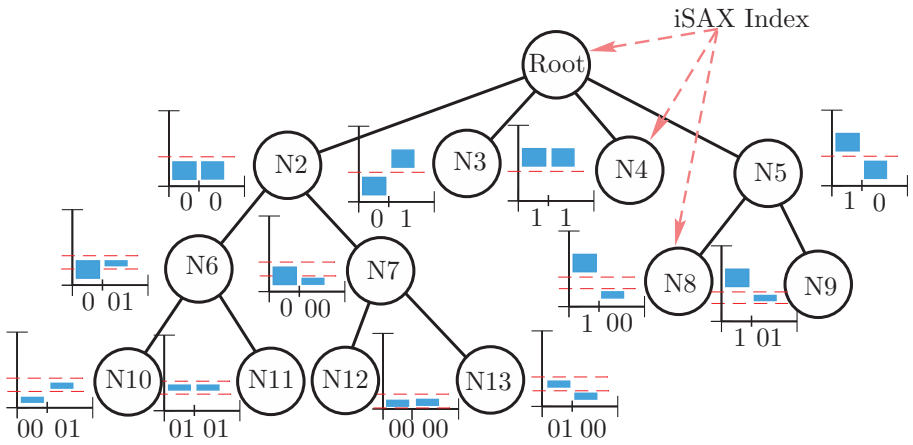


Figure 2.9 – Example of iSAX Index

than one billion time series. It uses two different buffer layers, namely First Buffer Layer (FBL) and Leaf Buffer Layer (LBL), for storing parts of the index and time series in memory before flushing them into the disk. In the FBL, all the time series time series that will end up in the same iSAX 2.0 subtree are cluster together and they can grow till they occupy all the available main memory, since they don't have a restriction in their size. The LBL corresponds to leaf nodes and they have the same size as the size of the leaf nodes. These buffers are used to gather all the time series of leaf nodes and flush them to disk. It also uses an efficient technique for splitting a leaf node when its size is higher than a threshold.

In [12], the authors propose two extensions of iSAX 2.0, namely iSAX 2.0 Clustered and iSAX2+. These extensions focus on the efficient handling of the raw time series data during the bulk loading process, by using a technique that uses main memory buffers to group and route similar time series together down the tree, performing the insertion in a lazy manner. We view iSAX2+ as the current state of the art in time series indexing.

In [80], instead of building the complete iSAX2+ index over the complete dataset and querying only later, Zoumpatianos et al. propose to adaptively build parts of the index, only for the parts of the data on which the user's issue queries.

In [69], based on PCA representation [13] Wang et al. propose DSTree that segments time series into variable length segment but adaptive to the shape of the series. DSTree

uses standard deviation to define the lower bounds. Unlike iSAX which only supports horizontal splitting, and only the mean values can be used in splitting, the DSTree uses multiple splitting strategies and provides more possible ways to divide time series leaf files.

2.2.3 Similarity Queries

Indexing is a technique used to speed up similarity search. After building the index, three search procedures can be applied in order to find similar time series in the datasets : (i) *Range search*; (ii) *Exact k* nearest neighbors; and (iii) *Approximate k* nearest neighbors. In information retrieval, finding the k nearest neighbors (k-NN) of a query is frequently used where the k closest results to the query are returned to the user. In this section, we define Range search and two kinds of k nearest neighbors based queries.

Definition 5. *(Range search)*

Given a query time series Q and a set of time series D, $\alpha - range\ query$ will return a subset $R \subseteq D : |ED(Q, R_i)| < \alpha$, where $ED(X, Y)$ is the Euclidean distance between the points X and Y.

Range search is not typically used for real-world applications because the parameter α cannot be easily determined by the user.

Definition 6. *(Exact k nearest neighbors)*

Given a query time series Q and a set of time series D, let $R = ExactkNN(Q, D)$ be the set of k nearest neighbors of Q from D. Let $ED(X, Y)$ be the Euclidean distance between the points X and Y, then the set R is defined as follows:

$$(R \subseteq D) \wedge (|R| = k) \wedge (\forall a \in R, \forall b \in (D - R), ED(a, Q) \leq ED(b, Q))$$

Definition 7. *(Approximate k nearest neighbors)*

Given a set of time series D, a query time series Q, and $\epsilon > 0$. We say that $R = AppkNN(Q, D)$ is the approximate k nearest neighbors of Q from D, if $ED(a, Q) \leq (1 +$

$\epsilon)ED(b, Q)$, *where* a *is the* k^{th} *nearest neighbor from* R *and* b *is the true* k^{th} *nearest neighbor.*

2.3 Time Series Indexing in Distributed Systems

Although centralized search methods achieve speed-ups of orders of magnitude compared with sequential scanning, their performances deteriorate as the size of the database increases, which poses questions concerning the scalability of centralized methods. To deal with the massive scale of time series data, a promising solution is to take advantage of parallel frameworks, such as MapReduce [19] or Spark [78], to make powerful computing and storage units on top of ordinary machines.

In this section, we first introduce MapReduce and Spark frameworks, and then present the parallel indexing solutions.

2.3.1 Parallel Frameworks

In this section, we introduce MapReduce and Spark, the two most popular frameworks that use ordinary machines of distributed systems for high performance parallel data processing.

2.3.1.1 MapReduce

MapReduce is one of the most popular solutions for big data processing [9], in particular due to its automatic management of parallel execution in clusters of machines. Initially proposed in [19], it was popularized by Hadoop [71], an open-source implementation. MapReduce divides the computation in two phases, namely map and reduce, which in turn are carried out by several tasks that process the data in parallel.

The idea behind MapReduce is simple and elegant. Given an input file, and two functions map and reduce, each MapReduce job is executed in two main phases: map

and reduce. In the first phase, called map, the input data is divided into a set of splits, and each split is processed by a map task in a given worker node. These tasks apply the map function on every key-value pair of their split and generate a set of intermediate pairs. In the second phase, called reduce, all the values of each intermediate key are grouped and assigned to a reduce task. Reduce tasks are also assigned to worker machines and apply the reduce function on the created groups to produce the final results.

Each MapReduce job includes two functions: map and reduce. For executing the job, we need a master node for coordinating the job execution, and some worker nodes for executing the map and reduce tasks. When a MapReduce job is submitted by a user to the cluster, after checking the input parameters, e.g., input and output directories, the input splits (blocks) are computed. The number of input splits can be personalized, but typically there is one split for each 64MB of data. The location of these splits and some information about the job are submitted to the master. The master creates a job object with all the necessary information, including the map and reduce tasks to be executed. One map task is created per input split. When a worker node, say w, becomes idle, the master tries to assign a task to it. The map tasks are scheduled using a locality-aware strategy. Thus, if there is a map task whose input data is kept on w, then the scheduler assigns that task to w. If there is no such task, the scheduler tries to assign a task whose data is in the same rack as w (if any). Otherwise, it chooses any task.

Each map task reads its corresponding input split, applies the map function on each input pair and generates intermediate key-value pairs. , which are firstly maintained in a buffer in main memory. When the content of the buffer reaches a threshold (by default 80% of its size), the buffered data is stored on the disk in a file called spill.

Once the map task is completed, the master is notified about the location of the generated intermediate key-values. In the reduce phase, each intermediate key is assigned to one of the reduce workers. Each reduce worker retrieves the values corresponding to its assigned keys from all the map workers, and merges them using an external merge-sort. Then, it groups pairs with the same key and calls the reduce function on the corresponding values. This function will generate the final output results. When, all tasks of a job are completed successfully, the client is notified by the master.

2.3.1.2 Spark

Spark [78] is an open-source parallel data processing framework that aims at efficiently processing large datasets. This programming model can perform data analytics with in-memory techniques to overcome disk bottlenecks. Spark extends the MapReduce model to efficiently support more types of computations, including interactive queries and stream processing. Spark can be deployed on the Hadoop Distributed File System (HDFS) [62] as well as standalone. Unlike traditional in-memory systems, the main feature of Spark is its distributed memory abstraction, called *resilient distributed datasets (RDD)*. RDD is an efficient and fault-tolerant abstraction for distributing data in a cluster. With RDD, the data can be easily persisted in main memory as well as on the hard drive. Spark is basically designed for being used in iterative algorithms.

To execute a Spark job, we need a master node to coordinate job execution, and some worker nodes to execute a parallel operation. These parallel operations are summarized to two types:

- *Transformations:* that are operations to create a new RDD from an existing RDD. As examples of transformations, we can mention Map, MapToPair, MapPartition, FlatMap.
- *Actions:* operations that return a final value to driving program or output to external storage, but do not change the original data. Examples of action operations are: Reduce, Aggregate and Count.

The transformations are not executed until a subsequent action has a need for the result. Where possible, these RDDs can persist in memory if persist function is called. In most cases persist increase the performance of the cluster.

2.3.2 Parallel Indexing

In [43], the authors propose a scalable parallel index structure for relational data processing. It is part of an indexing framework for MapReduce systems [19] and is extensible

in terms of both data and query types with multiple types of indexes. The index is organized as a tree structure and stored as a sequential file in the HDFS [62] file system. The index file is also partitioned into multiple blocks, each containing the data of a number of sub-indexes and some index blocks are loaded selectively into memory. To build an index using the proposed indexing framework, a new MapReduce job is submitted for index construction. In the map phase, the mappers scan the data and generate intermediate files, recording how data are distributed to different sub-spaces. In the reduce phase, each reducer collects the intermediate files from the mappers for a specific subspace and constructs a local index. The results constitute a set of sub-indexes, which are collected by the master node for the construction of the global index.

ScalaGiST [43] framework develops a distributed query processing service with one master and multiple workers. Each index worker handles one index block. Upon reception of a query, the master node forwards it to the worker hosting the root node, which progressively forwards the request to the other workers. The search results of ScalaGiST framework are offsets of the HDFS that refer to the data that satisfy the predicates.

However, none of the above solutions is appropriate for time series indexing and similarity search. On the whole, in the literature there is no time series index designed/adapted for operation in distributed environments.

2.4 Conclusion

In this chapter, we have discussed the state of the art about time series data mining. We gave an overview of time series data mining and a brief description of the main tasks in time series data mining that have attracted extensive research interest.

To the best of our knowledge, in the literature, there is no efficient solution for parallel indexing of time series in distributed environments. In this thesis, we carry out extensive theoretical and practical studies and propose various parallel indexing and querying solutions for time series, validated with real-world very large datasets.

Chapter 3

Massively Distributed Time Series Indexing and Querying with DPiSAX

Indexing is crucial for many data mining tasks that rely on efficient and effective similarity query processing. Consequently, indexing large volumes of time series, along with high performance similarity query processing, have became topics of high interest. For many applications across diverse domains though, the amount of data to be processed might be intractable for a single machine, making existing centralized indexing solutions inefficient.

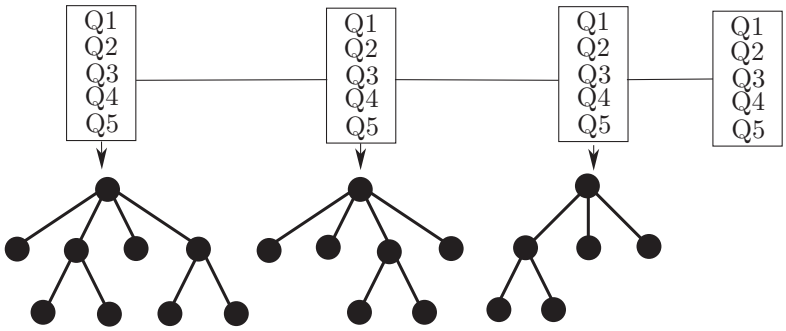
In this chapter, we propose two parallel indexing solutions that gracefully scale to billions of time series, and a parallel query processing strategy that, given a batch of queries, efficiently exploits the index.

The rest of the chapter is organized as follows. In Section 3.1, we present the context and give an overview of our work. We describe our algorithms in Sections 3.2 and 3.3. Then, in Section 3.4, we evaluate the proposed algorithms by carrying out extensive various experiments very large real-world data sets. Finally, we conclude in Section 3.5.

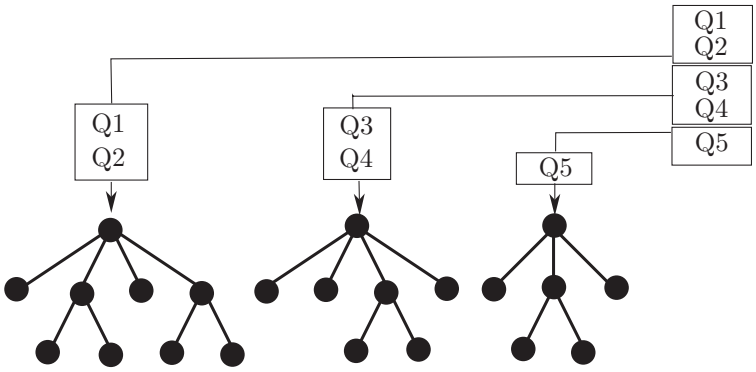
3.1 Motivation and Overview of the Proposal

Nowadays, individuals are able to monitor various indicators for their personal activities (*e.g.*, through smart-meters or smart-plugs for electricity or water consumption), or professional activities (*e.g.*, through the sensors installed on plants by farmers). Sensors technology is also improving over time and the number of sensors is increasing, *e.g.*, in finance and seismic studies. With such complex and massive sets of time series, we need to improve the performance of similarity queries. To do this, indexing is one of the most popular techniques [22], which has been successfully used in a variety of settings and applications [23, 66, 6, 69, 12, 80]. Although recent studies have shown that in certain cases sequential scans can be very efficient [56, 77], such techniques are only advantageous when the database consists of a single, long time series, and query answers are small subsequences of this long time series. Such approaches, however, are not beneficial in the general case of querying a mixed database of many small time series [81] (*e.g.*, in neuroscience, or manufacturing applications [50]), which is the focus of this study. Therefore, indexing is required in order to efficiently support data exploration tasks, which involve ad-hoc queries.

Interestingly and to the best of our knowledge, there has been no focus on the problem of similarity search in such massive sets of time series using scalable index construction. However, making an index over billions of time series by using traditional centralized approached is highly time-consuming. Moreover, a naive construction of the index on the parallel environment may lead to poor querying performances. This is illustrated in Figure 3.1 where the time series dataset is naively split on the W distributed nodes (Figure 3.1a). In this case, a batch of queries B has to be duplicated and sequentially processed on each node. By means of a dedicated strategy where each query in B could be oriented to the right partition (*i.e.*, the partition that must correspond to the query) the querying work load can be significantly reduced (Figure 3.1b shows an ideal case where B is split in W subsets and really processed in parallel). Our goal is to reach such an ideal distribution of index construction and query processing in massively distributed environments. In this work, we propose a parallel solution to construct the state of the art iSAX-based index [12] over billions of time series by making the most of the parallel environment by carefully distributing the work load. Our solution takes advantage of



(a) Straightforward implementation: the batch of queries is duplicated on all the computing nodes, and locally processed in sequential order.



(b) Ideal distribution of time series in the index nodes: each query is sent only to the relevant partition.

Figure 3.1 – Straightforward Vs. partitioned strategies for TS indexing and querying. Load balancing is a major lever.

the computing power of distributed systems by using parallel frameworks Spark [78]. We provide dedicated strategies and algorithms for a deep combination of parallelism and indexing techniques, for better query performances. We compare our approach to the current state of the art index building algorithm, and illustrate its important gains both in index construction and query processing time.

3.2 Distributed iSAX (DiSAX)

DiSAX, our first parallel index construction, sequentially splits the dataset for distribution into partitions. Then each worker builds an independent iSAX index on its partition, with the iSAX representations having the highest possible cardinalities. Representing each time series with iSAX words of high cardinalities allows us to decide later what cardinality is really needed, by navigating "on the fly" between cardinalities. The word of lower cardinality being obtained by removing the trailing bits of each symbol in the word of higher cardinality. The output of this phase, with a cluster of W nodes, is a set of W iSAX indexes built on each split.

The pseudo-code of this index construction can be seen in Algorithm 1. The input is a data partitions that contains time series in ASCII form. First, the algorithm obtain the iSAX representation of all time series using the highest possible cardinalities (lines 2-4). Then each worker builds an independent iSAX index on its partition (lines 5-9) using the iSAX index insertion function (lines 10-26).

3.2.1 Query Processing

Given a collection of queries Q, in the form of time series, and the index constructed in the previous section for a database D, we consider the problem of finding time series that are similar to Q in D, as presented in definitions 6 and 7. We perform such queries with two search methods: approximate and exact.

Algorithm 1: DiSAX Index construction

Input: Data partitions $P = \{P_1, P_2, \ldots, P_n\}$ of a database D, w the length of the iSAX word
Output: Index structures

```
1  D.cache(); //cache all the database in the cluster, where each time series has a
     unique ID
2  MapToPair( ID of Time series: ID ,Time Series: X )
3      Convert time series X to iSAX_word with high cardinalities and size w
4      emit (ID ,iSAX_word)
5  MapPartition( Set of <ID,iSAX_word>: iSAX_words )
6      rootNode = new RootNode
7      foreach <ID,iSAX_word> in iSAX_words do
8          rootNode.insert(ID ,iSAX_word)
9      emit (rootNode)
10 Function insert(ID ,iSAX_word)
11     if the subtree corresponding to iSAX_word exists then
12         node = the node corresponding to iSAX_word
13         if node is leaf node then
14             if node is not full then
15                 node.insert(ID ,iSAX_word)
16             else
17                 newNode = new InternalNode
18                 newNode.insert(ID ,iSAX_word)
19                 foreach iSAX_word in node do
20                     newNode.insert(ID ,iSAX_word)
21                 remove(node)
22         else
23             node.insert(ID ,iSAX_word)
24     else
25         newNode = new TerminalNode
26         newNode.insert(ID ,iSAX_word)
```

3.2.1.1 Approximate Search

Given a batch B of queries, the master node duplicates B on each worker (node) keeping an index for a subset of the data (i.e, a data split). Each worker uses its local index to retrieve time series that correspond to each query $Q \in B$, according to the approximate k-NN criteria. On each local index, the approximate search is done by traversing the local index to the terminal node that has the same iSAX representation as the query. The target terminal node contains at least one and at most th iSAX words, where th is the leaf threshold. A main memory sequential scan over these iSAX words is performed in order to obtain the k nearest neighbors using the Euclidean distance. Each worker w sends all the found time series to the master. Let $|W|$ be the number of workers, the master thus receives $k \times |W|$ nearest neighbors for each query Q, sorts them by decreasing order of their distance to Q, and selects the k top ones.

The algorithm, described in Algorithm 2, starts by obtain the iSAX representation of all queries time series using the highest possible cardinalities (lines 1-3). Then the master node duplicates the queries on each partition (worker) (line 4), and each worker uses its local index to retrieve time series that correspond to each query (lines 5-9), using the approximate search function (lines 10-15).

3.2.1.2 Exact Search

The exact search proceeds in two steps. In Step 1, the algorithm firstly uses the approximate search described in Section 3.2.1.1 to obtain *AKNN*, an approximate k nearest neighbours set. Then each worker creates a priority queue to examine the index nodes that may contain the time series that are probably more similar to Q than those of *AKNN*. Such nodes are identified as in the original iSAX [65, 66], where the lower bound distance used for priority queue ordering is computed using *MINDIST_PAA_iSAX* according to *AKNN*. The difference is that, instead of a sequential scan of the series found in the identified leaf nodes, we emit the IDs of the series. In step 2, the algorithm retrieves all the time series that match the IDs emitted by the workers, and then finds the k nearest neighbors using the Euclidean distance.

Algorithm 2: DiSAX Approximate Search

Input: iSAX Indexes where each partition has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of queries time series
Output: k nearest neighbors

```
MapToPair( ID of Time series: ID ,Time Series: q )
    Convert time series X to iSAX_word with high cardinalities and size w.
    emit (ID ,iSAX_word)
Duplicate Q on each partition
MapPartition( iSAX index, Set of <ID,iSAX_word>: iSAX_words )
    get the rootNode from iSAX index
    foreach <ID,iSAX_word> in iSAX_words do
        rootNode.ApprSearch(ID ,iSAX_word)
    emit (ApprSearch results)
Function ApprSearch(ID ,iSAX_word)
    node = the node corresponding to iSAX_word
    if node is a terminal node then
        Find the k nearest neighbors using Euclidean distance
    else
        node.ApprSearch(ID ,iSAX_word);
```

The algorithm, described in Algorithm 3. The master node duplicates the queries on each partition (worker) (line 1), and each worker uses its local index and starts by putting all the children of the root in priority queue using their lower distance bound towards the query (line 8), Then the one with the best minimum distance is explored (line 9), if the best lower bound is bigger than the BSF distance (line 12) the algorithm stops. If node is an internal node (line 15) then all children are added into the priority queue.

Algorithm 3: DiSAX Exact Search

Input: iSAX Indexes where each partition has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of queries time series
Output: k nearest neighbors

```
1  Duplicate Q on each partition
2  MapPartition( iSAX index, Q )
3      get the rootNode from iSAX index
4      foreach q in Q do
5          bsf = rootNode.ApprSearch(ID ,iSAX_word of q) rootNode.ExactSearch(ID
             ,q,bsf)
6      emit (ExactSearch results)
7  Function ExactSearch(ID ,q)
8      bsfDist = Infinite; queue = Initialize a priority queue with all the children of
         the root;
9      while node = pop next node from queue do
10         if node is terminal node and MinDist(q,node) < bsfDist then
11             bsf = Find the k nearest neighbors
12         else if MinDist(q, node) ≥ bsfDist then
13             break;
14         else
15             Add the children of the node to priority queue ;
```

3.2.2 Limitations of DiSAX

The parallel index constructed by DiSAX in a distributed environment is effective but calls for improvements. Actually, it leads to query response times that sometimes are

high, because the query processing work is not well distributed among the computing nodes. The reason is that each node should examine all queries in the index, even if the index contains no similar result for the query.

Furthermore the index obtained by iSAX2+ would be very different from the union of the local distributed iSAX indexes. This also has an impact on the size of the index. Since merging all the local indexes would call for specific algorithms (if it is even possible) the size of the global index of distributed iSAX is higher than the index of centralized iSAX2+.

3.3 Distributed Partitioned iSAX

In this section, we present a novel parallel partitioned index construction algorithms, along with very fast parallel query processing techniques.

Our approach is based on a sampling phase that allows anticipating the distribution of time series among the computing nodes. Such anticipation is mandatory for an efficient query processing, since it will allow, later on, to decide what partition contains the time series that actually correspond to the query. To do so, we first extract a sample from the time series dataset, and analyze it in order to decide how to distribute the time series in the splits, according to their iSAX representation. However, deciding the good split criteria calls for careful attention since bad choices may lead to highly imbalanced partitions, as illustrated in this section with i) DbasicPiSAX, a first version of our partitioned indexing technique and ii) DPiSAX, the final version with, to the best of our knowledge, the best load balance and the best querying performances obtained for time series indexing in distributed environments.

3.3.1 Sampling

In Distributed Partitioned iSAX, our index construction combines two main phases which are executed one after the other. First, the algorithm starts by sampling the time series

Table 3.1 – A sample S of 8 time series converted to iSAX representations with iSAX words of length 2

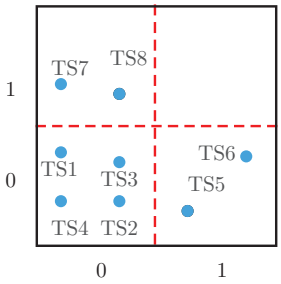
Time series	iSAX words	Time series	iSAX words
TS_1	$\{01, 00\}$	TS_5	$\{00, 10\}$
TS_2	$\{00, 01\}$	TS_6	$\{01, 11\}$
TS_3	$\{01, 01\}$	TS_7	$\{10, 00\}$
TS_4	$\{00, 00\}$	TS_8	$\{10, 01\}$

dataset and creates a partitioning table. Then, the time series are partitioned into groups using the partitioning table. Finally, each group is processed to create an iSAX index for each partition.

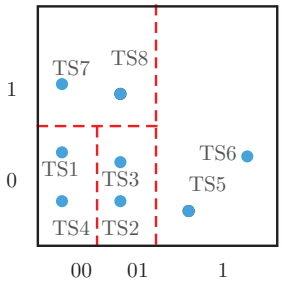
More formally, our sampling is done as follows. Given a number of partitions P and a time series dataset D, the algorithm takes S sample time series of size L from D using stratified sampling, and distributes them among the W available workers. Each worker takes S/W time series and emits its iSAX words $SWs = \{iSAX(ts_i), i = 1, ..., L\}$. The master collects all the workers' iSAX words and performs the partitioning algorithm accordingly. In the following, we describe two partitioning methods that enable separating the dataset into non-overlapping subsets based on iSAX representations, namely "the basic approach" (or DbasicPiSAX) and "the statistical approach" (or DPiSAX). Both methods proceed with a common simple strategy: successively divide the sample by splitting the biggest partition into two sub-partitions, until the number of partitions is equal to the number of workers. However, at each step, once the biggest partition is identified, the main difference is in the assignment strategy (*i.e.*, how is each time series in the sample assigned to one or the other of the new partitions?).

3.3.2 Basic Approach: DbasicPiSAX

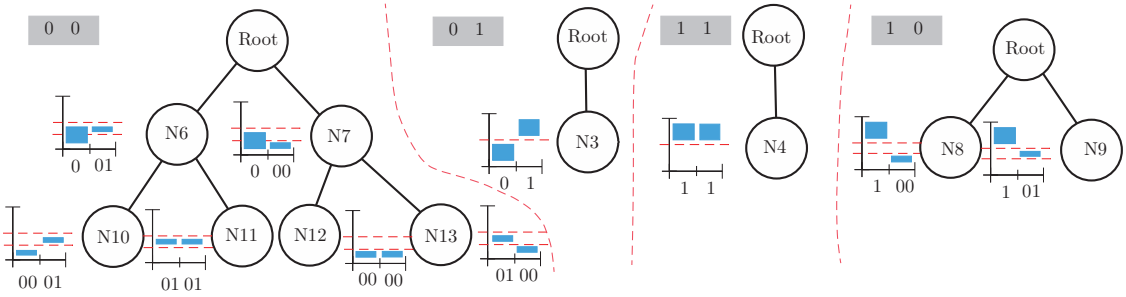
In the basic approach, splitting the biggest partition is done according to the first bit of each symbol in the iSAX words, as we can see in Algorithm 4 (line 1-4). Let us consider the n^{th} splitting step, each time series is assigned to a new partition depending on the first bit of its n^{th} symbol. Of course, when the number of symbols has been reached for a partition (*i.e.*, it cannot be divided anymore because the last symbol has been reached)



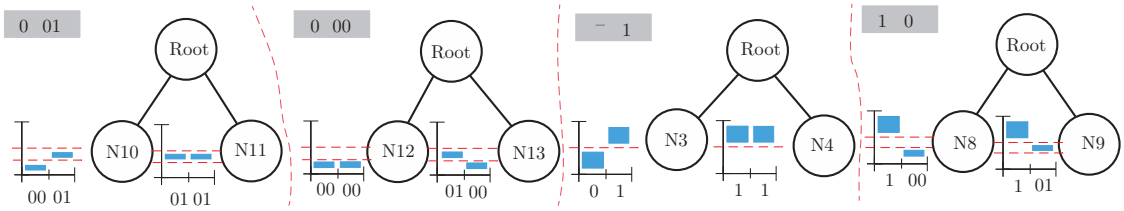
(a) Partitioning according to DbasicPiSAX.



(b) Partitioning according to DPiSAX.



(c) DbasicPiSAX indexes after partitioning and indexing. Partition "11" is empty for the sampling step, but may contain data after indexing.



(d) DPiSAX indexes after partitioning and indexing. The partitioning principle of DPiSAX allows better balance.

Figure 3.2 – The result of the partitioning algorithms (DPiSAX and DbasicPiSAX) on sample S (from Table 3.1) into four partitions.

then we need to consider the remaining partitions for new splits.

Example 5. *Let's consider Table 3.1, where we use iSAX words of length two to represent the time series of a sample S. Suppose that we need to generate four partitions. First, we use the first bit of the first segment to define two partitions. The first partition contains all the time series having their first iSAX word starting with 1, and the second partition contains the time series having their first iSAX word starting with 0. We obtain two partitions: "0" and "1". The biggest partition is "0" (i.e., containing the time series TS1 to TS6). This partition is split again, according to the first bit of the second symbol. We now have the following partitions: from the first step, partition "1", and from the second step, partitions "00", and "01". Now, partition "00" is the biggest one. However, it cannot be split anymore since the maximum number of symbols has been reached. We choose the next biggest partition, i.e., "1". After splitting this partition using the first bit of the second segment, we obtain two new partitions:"11" and "10". Partition "10" contains all the time series of the old partition (i.e., partition "1"). Consequently, we have four partitions where partition "11" is empty. Figure 3.2a shows the obtained partitions and Figure 3.2c shows the indexes obtained with these partitions.*

The partitioning Algorithm achieves two goals: 1) generating P partitions; and 2) preserving vertical division of the iSAX tree. Notice that the second goal is achieved because our partitioning algorithm uses the first bit of each symbol. Therefore, iSAX words having cardinality 2 are used to produce a set of, at most, 2^w partitions. In the original iSAX index, when the construction starts with a cardinality of 2, a set of 2^w children is produced at the root node. Intuitively, in our running example, when we compare the centralize index (the original iSAX index) in Figure 2.9, and the parallel indexes in the Figure 3.2c obtained with the basic partitioning approach, we observe the vertical division of the original iSAX index.

3.3.3 Limitations of the Basic Approach

Obviously, the partitions obtained with the basic partitioning approach are not balanced. This is due to two main reasons. First, the partitioning algorithm preserves vertical division of the original iSAX index and the iSAX index is not balanced. The second reason

is that, the partitioning algorithm does not take into account the data distribution in the partitions. Because of the limits in the number of symbols, it is possible to end up with highly imbalanced partitions, as illustrated by Figure 3.2c and also by our experiments. Because of this imbalanced distribution of the data, the basic approach is limited in the size of datasets it can process. If the capacity of a computing node is reached (*i.e.*, the node in charge of the biggest partition cannot handle the data that corresponds to it), then the index building process cannot progress.

Moreover, the maximum number of partitions that can be generated is 2^w (where w is the SAX word length). Since each partition is managed by a computing node for the local index construction, if the number of partitions is lower than the number of available computing nodes, then there will be idle nodes. This is a threat for the speed-up of the approach and calls for better solutions, as presented in the next subsection.

Algorithm 4: DbasicPiSAX Partitioning Function

Input: Sample S of *iSAX_words*, p number of partitions
Output: Partition Table *BT*
1 **while** *the number of partitions is less than p* **do**
2 *BigPartition* = the biggest partition
3 *//In the first iteration BigPartition = S*
4 Divide *BigPartition* into two partitions

3.3.4 Statistical Approach: DPiSAX

Here, our partitioning paradigm considers the splitting power of each bit in the iSAX symbols, before actually splitting the partition. As in the basic approach, the biggest partition is considered for splitting at each step of the partitioning process. The main difference is that we don't use the first bit of the n^{th} symbol for splitting the partition. Instead, we look for all bits (whatever the symbol) (Algorithm 5 lines 7-11) with the highest probability to equally distribute the time series of the partition among the two new sub-partitions that will be created. To this effect, we compute for each segment the $\mu \pm \sigma$ interval (lines 4-5), where μ is the mean and σ is the standard deviation, and we examine for each segment if the break-point of the additional bit (*i.e.*, the bit used to

generate the two new partitions) lies within the interval $\mu \pm \sigma$ (line 9). From the segments for which this is true, we choose the one having μ closer to the break-point (line 10).

In order to illustrate this, let us consider the blue boxes of the diagrams in Figure 3.2c. We choose the biggest blue box that ensures the best splitting by considering the next break-point.

Example 6. *Let's consider the same case as described in Example 5. Figure 3.2b shows the obtained partitions and Figure 3.2d shows the indexes obtained with these partitions. To generate four partitions, we compute the $\mu \pm \sigma$ interval for the first segment and the second segment, and choose the first bit of the second segment to define two partitions. The first partition contains all the time series having their second segment in iSAX word starting with 0, and the second partition contains the time series having their second segment in iSAX word starting with 1. We obtain two partitions: "0" and "1". The biggest partition is "0" (i.e., the one containing time series $TS1$ to $TS4$, $TS7$ and $TS8$). We compute the $\mu \pm \sigma$ interval for all segment over all the time series in this partition. Then, the partition is split again, according to the first bit of the first symbol. We now have the following partitions: from the first step, partition "1", and from the second step, partitions "00", and "10". Now, partition "00" is the biggest one. This partition is split for the third time, according to the second bit of the first symbol and we obtain four partitions.*

We also illustrate, in Figure 3.2c, the variability of the distribution of time series for each symbol. For instance, in partition "00", for node $N6$, there is a much higher variability in the first symbol (marked "0" in the diagram, and represented by the blue box) than the second symbol (marked "01", blue box).

Optimization. Because many time series have the same iSAX representation, we may end up with groups of iSAX words that are the same, even when using the maximum cardinality (as it is our case). Therefore, we turn this data duplication into an advantage. Actually, the index construction is done as in Section 3.2, but the difference is that in the insertion function, we provide the algorithm with a bulk insertion function. The goal of this function is to better consider iSAX words with the same representation and to improve the index construction cost. This is done by linking all the IDs of time series having the same representation to only one corresponding iSAX word.

The pseudo-code of the parallel index construction by DPiSAX is shown in Algorithm 6. Given a time series dataset, the algorithm firstly creates the iSAX representation of each time series in parallel (lines 2-4). Then, it inserts the representations in parallel to the index by using the bulkInsertion function (lines 5-9). Each time series t is inserted to the index by the worker (*i.e.*, the processor) that is responsible for the partition to which t belongs. If the subtree of the partition does not exist, it will be created (lines 23- 25). Then, the time series t is inserted to its corresponding leaf node in the subtree (lines 14-15). If the node gets full (*i.e.*, its size gets higher than the threshold), then it will be split (lines 16-20).

Algorithm 5: DPiSAX Partitioning Function

```
   Input: Sample S of iSAX_words, p number of partitions
   Output: Partition Table BT
1  while the number of partitions is less than p do
2      BigPartition = the biggest partition
3      //In the first iteration BigPartition = S
4      mean[] = ComputeSymbolsMean(BigPartition)
5      stdev[] = ComputeSymbolsStDev(BigPartition)
6      segmentToSplit = null
7      foreach segment s in BigPartition do
8          b = getbreak-point(s)
9          if b within mean[s] ± stdev[s] then
10             if mean[s] close to b then segmentToSplit then
11                 segmentToSplit = s
12     Divide BigPartition into two partitions in segmentToSplit
```

3.3.5 Query Processing

Given a collection of queries Q, in the form of time series, and the index constructed in the previous section for a database D, we consider the problem of finding time series that are similar to Q in D, according to the definitions of approximate k-NN and exact k-NN search as presented in definitions 6 and 7. Approximate and exact search are performed as follows:

Algorithm 6: DPiSAX Index construction

Input: Data partitions $P = \{P_1, P_2, \ldots, P_n\}$ of a database D, w the length of the iSAX word, p number of partitions
Output: Index structures

```
1  D.cache(); //cache all the database in the cluster, where each time series has a
     unique ID
2  MapToPair( ID of Time series: ID ,Time Series: X )
3      Convert the time series X to iSAX_word with high cardinalities and size w
4      emit (ID ,iSAX_word)
5  MapPartition( Set of Set<ID,iSAX_word>: iSAX_words )
6      rootNode = new RootNode
7      foreach Set <ID,iSAX_word> in iSAX_words do
8          rootNode.bulkInsertion(Set<ID,iSAX_word>)
9      emit (rootNode)
10 Function bulkInsertion(Set <ID,iSAX_word>: iSAX_words)
11     if the subtree corresponding to iSAX_words exists then
12         node = the node corresponding to iSAX_words
13         if node is leaf node then
14             if node is not full then
15                 node.bulkInsertion(iSAX_words)
16             else
17                 newNode = new InternalNode
18                 newNode.bulkInsertion(iSAX_words)
19                 newNode.bulkInsertion(all iSAX words of node)
20                 remove(node)
21         else
22             node.bulkInsertion(iSAX_words)
23     else
24         newNode = new TerminalNode
25         newNode.bulkInsertion(iSAX_words)
```

- **Approximate search**: searching for the approximate k nearest neighbors of the time series Q is done as in Section 3.2.1.1. The difference is that just one iSAX index is queried instead of all the parallel indexes. Actually, we are able to identify the right partition where the index is stored and send the corresponding query by using its iSAX words. Then, we send each query to the partition that has the same iSAX word as the query. The pseudo-code of DiSAX is shown in Algorithm 7. It starts by obtaining the iSAX representation of all queries time series using the highest possible cardinalities (lines 1-3). The master sends each query to the partition (worker) that has the same iSAX word as the query (line 4), and each worker uses its local index to retrieve time series that correspond to each query (lines 5-9), using the approximate search function (lines 10-15).

Algorithm 7: DPiSAX Approximate Search

Input: iSAX Indexes where each partition has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of Query time series
Output: k nearest neighbors

```
1  MapToPair( ID of Time series: ID ,Time Series: q )
2      Convert time series X to iSAX_word with high cardinalities and size w.
3      emit (ID ,iSAX_word)
4  Send each query to the partition that has the same iSAX word as the query
5  MapPartition( iSAX index, Set of <ID,iSAX_word>: iSAX_words )
6      get the the rootNode from iSAX index
7      foreach <ID,iSAX_word> in iSAX_words do
8          rootNode.ApprSearch(ID ,iSAX_word)
9      emit (ApprSearch results)
10 Function ApprSearch(ID ,iSAX_word)
11     node = the node corresponding to iSAX_word
12     if node is terminal node then
13         Find the k nearest neighbors using Euclidean distance
14     else
15         node.ApprSearch(ID ,iSAX_word);
```

- **Exact search**: for retrieving the exact k nearest neighbors of a given query time series q, we first use the approximate search, described above, in order to obtain

an approximate best-so-far k nearest neighbors. Then, each worker performs the exact search algorithm as described in Section 3.2.1.2. This is described in Algorithm 8. The master sends each query to the partition (worker) that has the same iSAX word as the query (line 1), and each worker uses its local index and starts by putting all the children of the root in priority queue using their lower distance bound towards the query (line 8). Then, the one with the best minimum distance is explored (line 9). If the best lower bound is bigger than the BSF distance (line 12) then the algorithm stops. If the node is an internal node (line 15) then all its children are added to the priority queue.

Algorithm 8: DPiSAX Exact Search

Input: iSAX Indexes where each partitions has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of queries time series
Output: k nearest neighbors

```
1  Send each query to the partition that has the same iSAX word as the query
2  MapPartition( iSAX index, Q )
3      get the rootNode from iSAX index
4      foreach q in Q do
5          bsf = rootNode.ApprSearch(ID ,iSAX_word of q) rootNode.ExactSearch(ID
             ,q,bsf)
6      emit (ExactSearch results)
7  Function ExactSearch(ID ,q)
8      bsfDist = Infinite; queue = Initialize a priority queue with all the children of
         the root;
9      while node = pop next node from queue do
10         if node is terminal node and MinDist(q,node) < bsfDist then
11             bsf = Finds the k nearest neighbors
12         else if MinDist(q, node) ≥ bsfDist then
13             break;
14         else
15             Add the children of the node to priority queue ;
```

Table 3.2 – Default parameters

Parameters	Value	Parameters	Value
iSAX word length	8	Leaf capacity	1,000
Basic cardinality	2	Number of machines	32
Maximum cardinality	512	Sampling fraction	10%

3.4 Performance Evaluation

In this section, we report experimental results that show the quality and the performance of DPiSAX for indexing time series.

The parallel experimental evaluation was conducted on a cluster of 32 machines, each operated by Linux, with 64 Gigabytes of main memory, Intel Xeon CPU with 8 cores and 250 Gigabytes hard disk. The iSAX2+ approach was executed on a single machine with the same characteristics.

We evaluate the performance of three versions of our solution: 1) DiSAX is the parallel implementation of iSax as described in Section 3.2 ; 2) DbasicPiSAX is the sampling-based indexing algorithm with basic partitioning as described in Section 3.3.2; 3) DPiSAX is our complete solution, with the statistical partitioning described in Section 3.3.4. Furthermore, we compare our solutions to two state of the art baselines: the most efficient centralized version of iSAX index (*i.e.*, iSAX2+ [12]), and Parallel Linear Search (PLS), which is a parallel version of the UCR Suite fast sequential search (with all applicable optimizations in our context: no computation of square root, and early abandoning) [56].

Our experiments are divided into two sections. In Section 3.4.2, we measure the index construction times with different parameters. In Section 3.4.3, we focus on the query performance of our approach.

we implemented our approaches on top of Apache-Spark [78] as a distributed environment, using the Java programming language. The iSAX2+ index is also implemented with Java.

3.4.1 Datasets and Settings

We carried out our experiments on two real world and synthetic datasets, up to 6 Terabytes and 4 billion series. The first real world data represents seismic time series collected from the IRIS Seismic Data Access repository [30]. After preprocessing, it contains 40 millions time series of 256 values, for a total size of 150Gb. The second real world data is the TexMex corpus [31]. It contains 1 Billion time series (SIFT feature vectors) of 128 points each (derived from 1 Billion images). Our synthetic datasets are generated using a Random Walk principle, each data series consisting of 256 points. At each time point the generator draws a random number from a Gaussian distribution N(0,1), then adds the value of the last number to the new number. This type of generator has been widely used in the past. [3, 23, 6, 65, 11, 12, 80]. Table 3.2 shows the default parameters (unless otherwise specified in the text) used for each approach. The iSAX word length, PAA size, leaf capacity, basic cardinality, and maximum cardinality were chosen to be optimal for iSAX, which previous works [65, 66, 11, 12, 80] have shown to work well across data with very different characteristics.

3.4.2 Index Construction Time

In this section, we measure the index construction time in DPiSAX, DbasicPiSAX and DiSAX, and compare it to the construction time of the iSAX2+ index.

Figure 3.3 reports the index construction times for all approaches on our Random Walk dataset. The index construction time increases with the number of time series for all approaches. This time is much lower in the case of all parallel approaches, than that of the centralized iSAX2+. On 32 machines, and for a dataset of one billion time series, DPiSAX builds the index in 65 minutes, DbasicPiSAX in 76 minutes and DiSAX in 64 minutes, while the iSAX2+ index is built in more than 5 days on a single node.

Figure 3.4 shows the same evaluation on the TexMex dataset. We can observe very similar behavior of our parallel approaches. As for the previous experiment, reported in Figure 3.3, the centralized version of iSAX2+ builds the index on a single machine in up to 4 days. We only report the response time of scalable approaches in Figure 3.4, for a

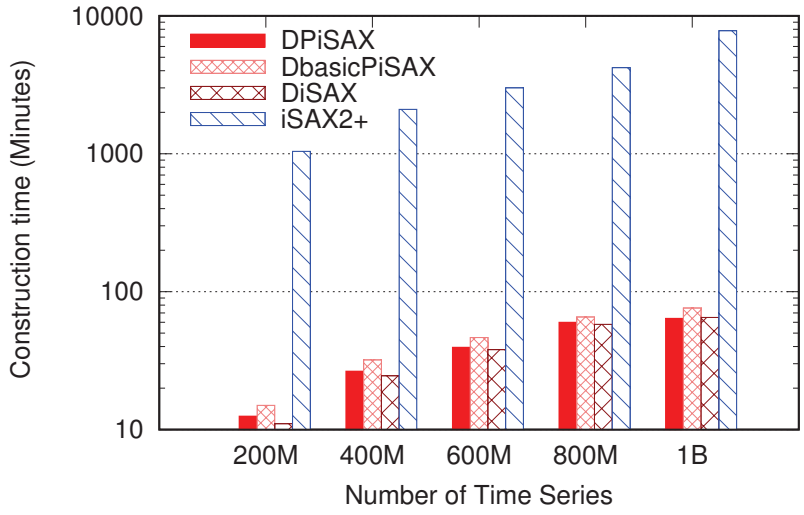


Figure 3.3 – Logarithmic scale. Construction time as a function of dataset size. Parallel algorithms (DiSAX and DPiSAX) are run on a cluster of 32 nodes. iSAX2+ is run on a single node. With 1 billion Random Walk TS, iSAX2+ needs 5 days and our distributed algorithms need less than 2 hours.

better visual comparison of their performances.

Figure 3.5 reports an extended view on the index construction times, only for parallel approaches, and with datasets having size up to 4 billion time series (6.2TB). The running time increases with the number of time series for DPiSAX and DiSAX. DbasicPiSAX does not scale and cannot execute on datasets having size above 1Tb. This is due to its imbalanced partitions, where one of the computing node receives so much data that it cannot build the index. This will be better discussed with Figure 3.8.

Figures 3.6 and 3.7 illustrate the parallel speed-up of our approach on the Random Walk (Figure 3.6) and the TexMex (Figure 3.7) datasets. The results show a near optimal gain for DPiSAX and DiSAX on our dataset. From the figure 3.6, we observe that the construction time for DbasicPiSAX is the same with 32 nodes and 40 nodes, this is because DbasicPiSAX does not use all the available processors. Actually, the basic partitioning algorithm (as described in Section 3.3.2) is limited in the number of partitions it can generate. By construction, it is able to generate up to $2^8 = 256$ partitions (more

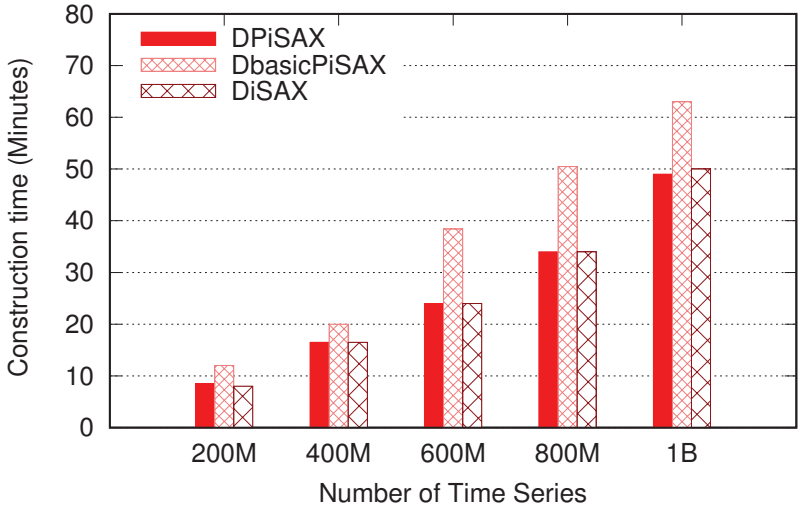


Figure 3.4 – Construction time as a function of dataset size. Parallel algorithms (DiSAX and DPiSAX) are run on a cluster of 32 nodes. With 1 billion TS from TexMex dataset.

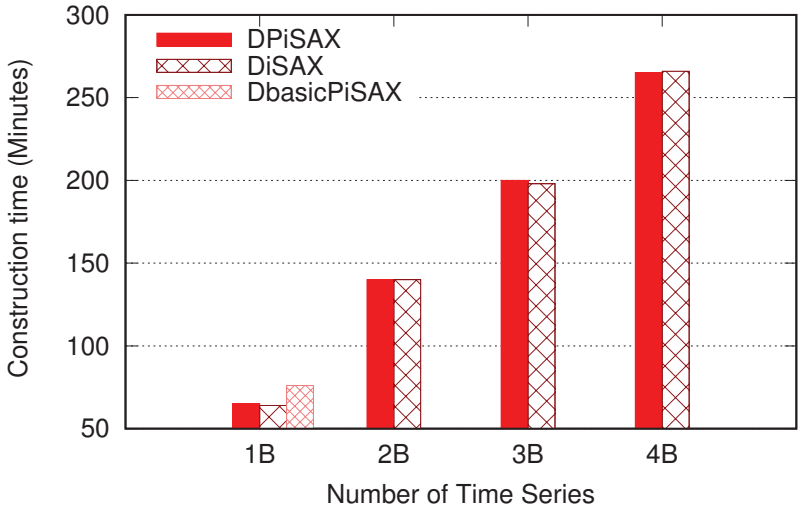


Figure 3.5 – Construction time as a function of dataset size, for parallel algorithms on a cluster of 32 nodes, and with datasets up to 4 billion Random Walk time series.

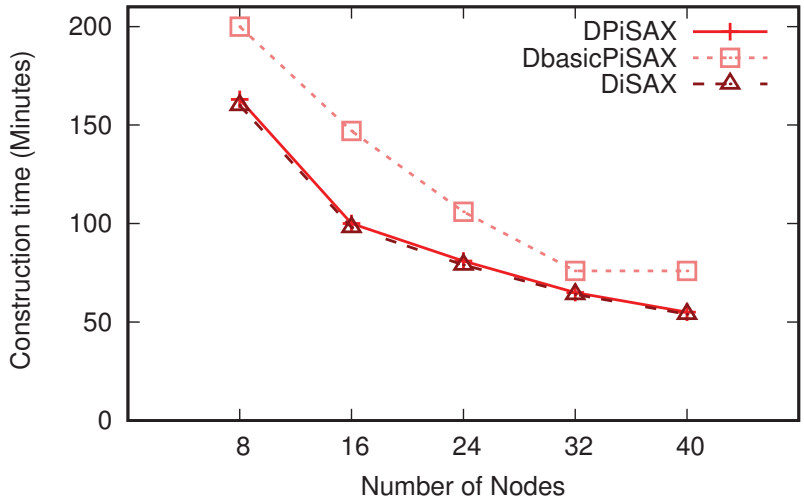


Figure 3.6 – Construction time as a function of cluster size. DPiSAX and DiSAX have has a near optimal parallel speed-up. With 1 billion TS from the Random Walk dataset.

generally, 2^w partitions where w is the SAX word length). In order to fully exploit the computing of all 320 cores, we need to build 320 partitions. This is over the maximum number of partitions that DbasicPiSAX is able to manage (*i.e.*, in this case, 256).

Figure 3.8 reports our measures of load balance, on 32 nodes and one billion time series, where partitions are sorted by decreasing order of the measured criteria: number of nodes in the local trees (Fig. 3.8a), number of time series in the partitions (Fig. 3.8b) and index depth (Fig. 3.8c). Our results illustrate the near ideal balance of our DPiSAX approach, while DbasicPiSAX is totally unbalanced. The number of time series, for instance, in the case of DbasicPiSAX, ranges from 0 (which means an empty partition) to 100 millions (*i.e.*, 10% of the data is indexed on one partition out of 320). DiSAX is perfectly balanced in the index construction phase owing to its sequential split of the data in the partitioning phase, but totally imbalanced in querying because it has to send the whole batch of queries to all partitions, leading to poor performances as illustrated in the remaining of our experiments.

Figure 3.9 reports the performance gains of our parallel approaches on the centralized version of iSAX2+ on our synthetic and real datasets. The results show that DPiSAX is between 40 and 120 times faster than iSAX2+. We observe that the performance gain

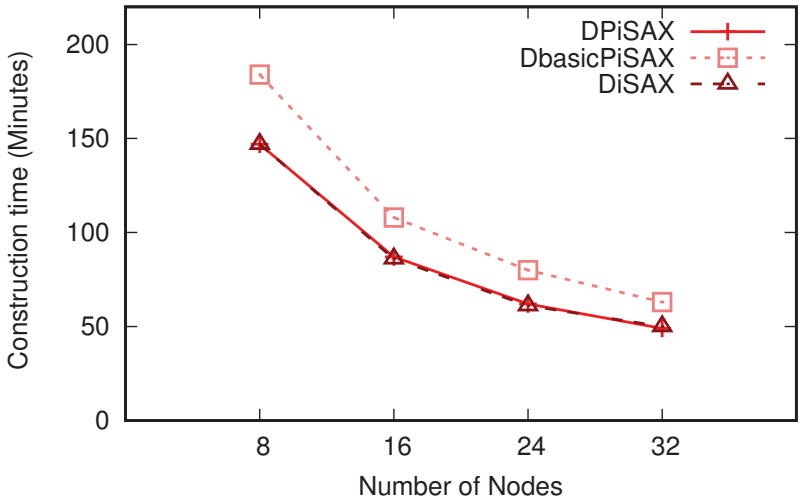


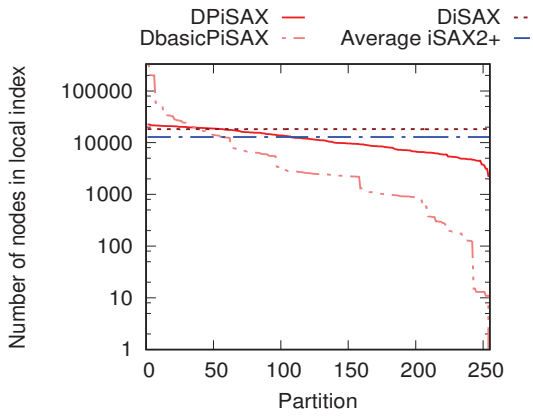
Figure 3.7 – Construction time as a function of cluster size. DPiSAX and DiSAX have has a near optimal parallel speed-up. With 1 billion TS from the TexMex dataset.

depends on the dataset size in relation to the number of Spark nodes used in the deployment. Note that the time Spark needs to deploy on 32 nodes is accounted for in our measurements. Thus, given the very short time needed to construct the DPiSAX index on the seismic dataset (420 seconds), the proportion of time taken by the Spark deployment when compared to index construction, is higher than the much larger Random Walk dataset.

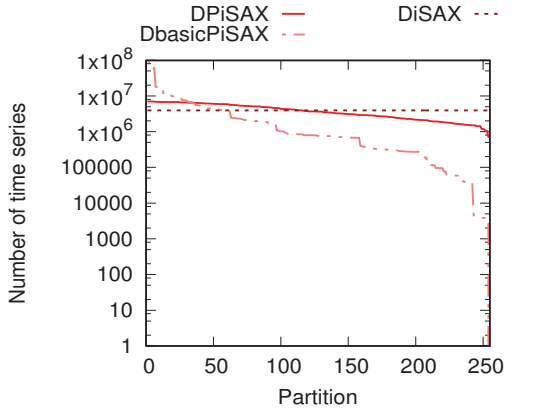
Our experiments with varying leaf capacity show that this parameter has a negligible effect on performance (results omitted for brevity). This is because the RDD implementation used by Spark avoids the performance penalty related to disk I/O, which is heavily affected by the choice of the leaf capacity [12].

3.4.3 Query Performance

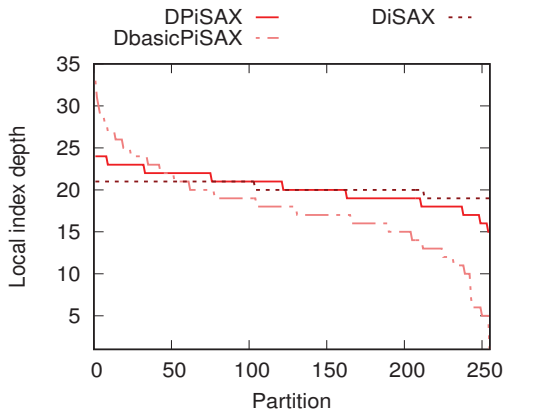
In the following experiments, we evaluate the querying performance of our algorithms, and compare them to the state of the art. In the case of our synthetic data, we generate Random Walk queries with the same distribution as described in Section 3.4.1. For the



(a) number of nodes



(b) number of time series



(c) Local index depth

Figure 3.8 – Load balance in partitions: distribution of the number of nodes (a), of the number of time series (b), and of index depth (c), sorted by decreasing order in the partitions. The strong imbalance of DbasicPiSAX is the main reason of failure on massive datasets (i.e., above 1 billion TS).

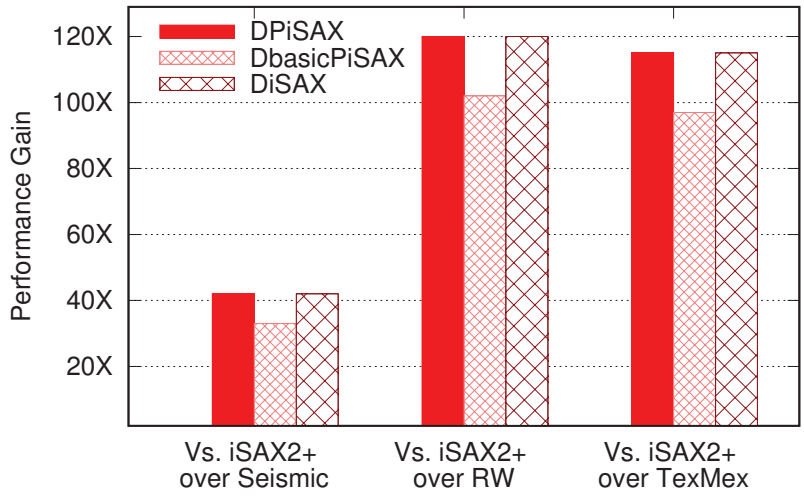


Figure 3.9 – Performance gain on iSAX2+ in construction time, over seismic (40 millions TS), Random Walk (RW, 1 billion TS) and TexMex (1 billion TS), with a cluster of 32 nodes.

seismic data, we obtained seismic time series from the same IRIS Seismic Data Access repository [30] to be used as queries. In the case of the TexMex corpus, similar series correspond to similar images. The corpus contains 10^4 example queries together with information about which image in the corpus is the nearest neighbor. In any dataset, for each time series t in the query batch, the goal is to check if the approach is able to find the k time series that are considered to be the most similar to t in this dataset, both with exact and approximate K-NN search.

Figure 3.10 compares the search time of approximate k nearest neighbors queries for the parallel approaches proposed in this work. We can observe that the response time increases with the number of queries for all approaches. However, for DPiSAX the search time is lower than DbasicPiSAX (owing to the better partition balancing) and much better than DiSAX (owing to DPiSAX's cability of splitting the query batch and redirect the queries to the adequate partitions). In our experiments, we also compared the search time of parallel approaches to that of iSAX2+ for answering approximate k nearest neighbors queries with a varying size of query batch. We observed that the approximate search time of DPiSAX is better than that of the iSAX2+ by a factor of up

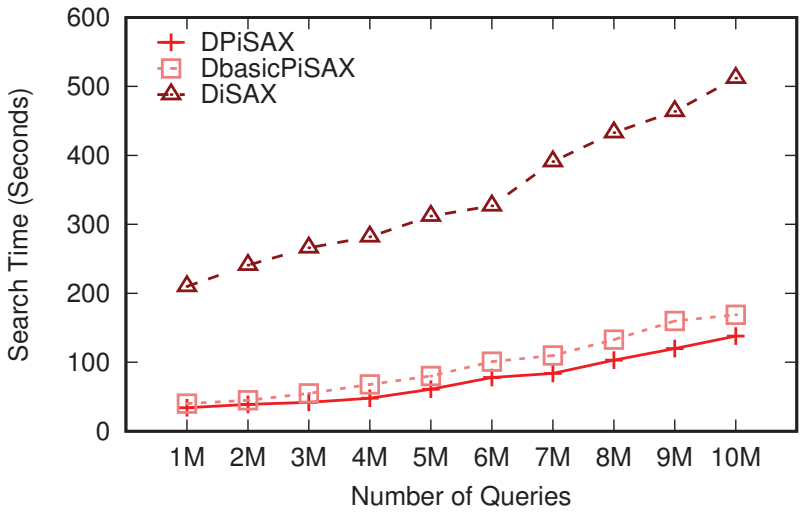


Figure 3.10 – Run time of approximate 10-NN queries over Random Walk dataset (limited to 1 billion TS because DbasicPiSAX does not scale on bigger datasets), cluster of 32 nodes,.

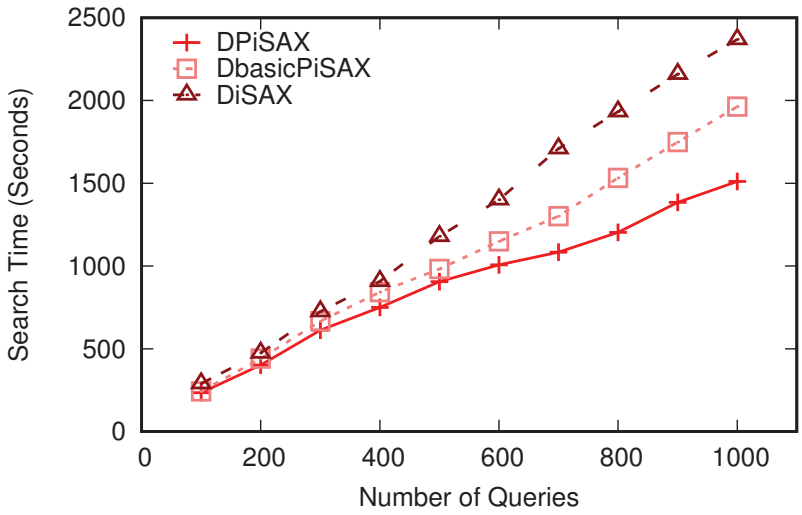


Figure 3.11 – Run time of exact 10-NN queries over Random Walk dataset (limited to 1 billion TS because DbasicPiSAX does not scale on bigger datasets), cluster of 32 nodes.

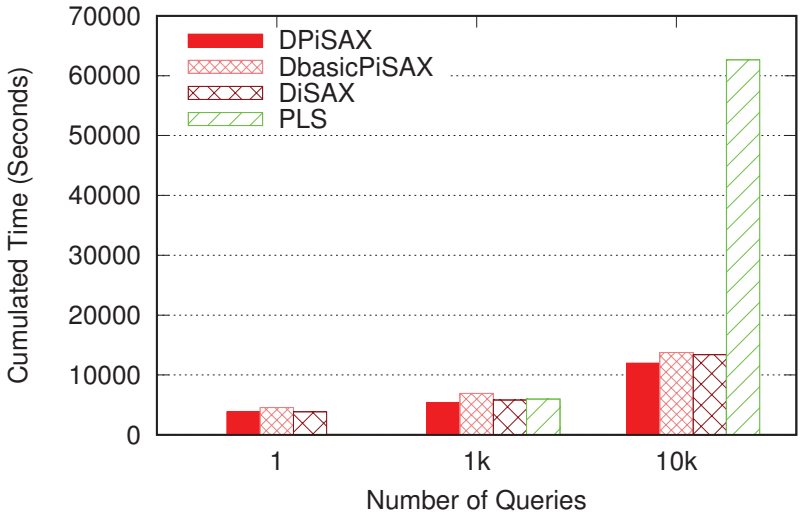


Figure 3.12 – Cumulative time (Indexing + Exact 10-NN) over Random Walk dataset (limited to 1 billion TS because DbasicPiSAX does not scale on bigger datasets), cluster of 32 nodes.

to 16 (*e.g.*, the search time for 10 millions queries is 2270 sec for iSAX2+ and 138 sec for DPiSAX).

Figure 3.11 gives the exact search run time of our parallel approaches on the index constructed over 1 billion time series. We observe that DPiSAX is always faster than DbasicPiSAX and DiSAX, owing to its near ideal load balance.

Figure 3.12 compares cumulative time (Indexing + Exact 10-NN) of DPiSAX, DbasicPiSAX and DiSAX to PLS. A direct use of PLS is justified under 1K queries. Above that limit, the cumulative time of building the index and querying is much lower for our approaches, which are the clear winners.

Figure 3.13 illustrates the performance gains of our approaches on the centralized version of iSAX2+ and on PLS on synthetic and real world datasets, with batches of 10K queries (indexing time not included). We observe that DPiSAX and DbasicPiSAX have the best performance, owing to their query redirection mechanisms. However, DbasicPiSAX is not always as efficient as DPiSAX because of a less balanced partitioning. DPiSAX is generally between 19 and 43 times faster than iSAX2+ and PLS.

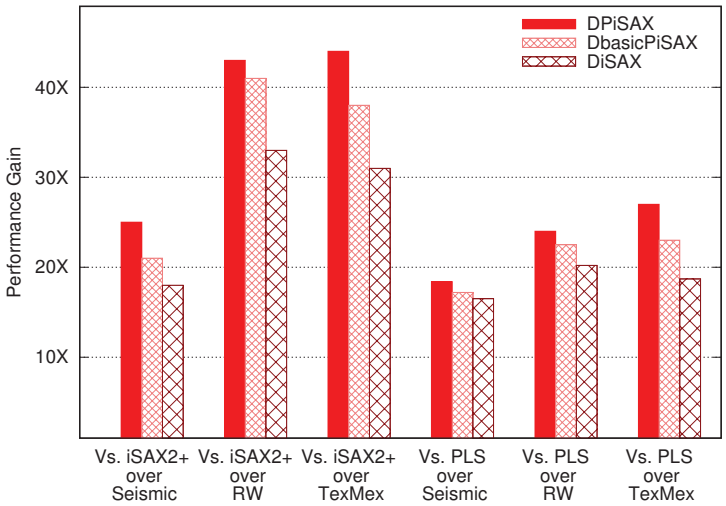


Figure 3.13 – Performance gain (query only) of our parallel approaches on iSAX2+ and PLS, for exact 10-NN search time, batches of 10k queries, over seismic, Random Walk (RW) and TexMex datasets.

3.5 Conclusions

We proposed DPiSAX, a novel and efficient parallel solution to index and query billions of time series. We evaluated the performance of our solution over large volumes of real world and synthetic datasets (up to 4 billion time series, for a total volume of 6TBs). The experimental results illustrate the excellent performance of DPiSAX (*e.g.*, an indexing time of less than 2 hours for more than one billion time series, while the state of the art centralized algorithm needs several days). The results also show that the distributed querying algorithm of DPiSAX is able to process millions of similarity queries over collections of billions of time series with very fast execution times (*e.g.*, 140s for 10M queries), thanks to our load balancing mechanism. Overall, the experimental results show that by using our parallel techniques, the indexing and mining of very large volumes of time series can now be done in very small execution times, which are impossible to achieve using traditional centralized approaches.

Chapter 4

RadiusSketch: Massively Distributed Indexing of Time Series

In centralized systems, one of the efficient ways to index time series for the purpose of similarity search is to combine a sketch approach with grid structures [17]. Random projection is based on the idea of taking the inner product of each time series, considered as a vector, with a set of random vectors whose entries are +1 or -1 [17]. The resulting sequence of inner products is called a *sketch vector* (or *sketch* for short). The goal is to reduce the problem of comparing pairs of time series to the problem of comparing their sketches, which are normally much shorter.

To avoid comparing the sketch of each time series of the database with that of the searched time series, [17] uses grid structures on pairs of sketch entries (*e.g.*, the first and second entry in one grid, the third and fourth in the second grid, and so on) to reduce the complexity of search. Given the sketches s and s' of two time series t and t', the more grids in which s and s' coincide, the greater the likelihood that t and t' are similar. In time series data mining, sketch-based approaches have also been used to identify representative trends [18, 29], maintain histograms [67], and to compute approximate wavelet coefficients [24], etc. All aspects of the sketch-based approach are parallelizable: the computation of sketches, the creation of multiple grid structures, and the computation of pairwise similarity. However, a straight parallel implementation of

existing techniques would under-exploit the available computing power.

In this work, we propose a parallel solution to construct a sketch-based index over billions of time series. Our solution makes the most of the parallel environment by exploiting each available core. Our contributions are as follows:

- We propose a parallel index construction algorithm that takes advantage of distributed environments to efficiently build sketch-based indices over very large volumes of time series. In our approach, we provide a greedy technique that uses idle processors of the system to increase query precision.
- We propose a parallel query processing algorithm, which given a query, exploits the available processors of the distributed system to answer the query in parallel by using the constructed index which has already been distributed among the nodes of the system at construction time.

The rest of this chapter is organized as follows. In Section 4.1, we describe the details of our parallel index construction and query processing algorithms. In Section 4.2, we present a detailed experimental evaluation to verify the effectiveness of Sketch Approach compared to iSAX2+. Finally, we conclude in Section 4.3.

4.1 Parallel Sketch Approach

This section reviews our algorithm for sketches, discusses the index structure required, and then shows how to parallelize the construction both to increase speed and improve quality.

4.1.1 The Sketch Approach

The sketch approach, as developed by Kushilevitz et al. [42], Indyk et al. [28], and Achlioptas [2], provides a very nice guarantee: with high probability a random map-

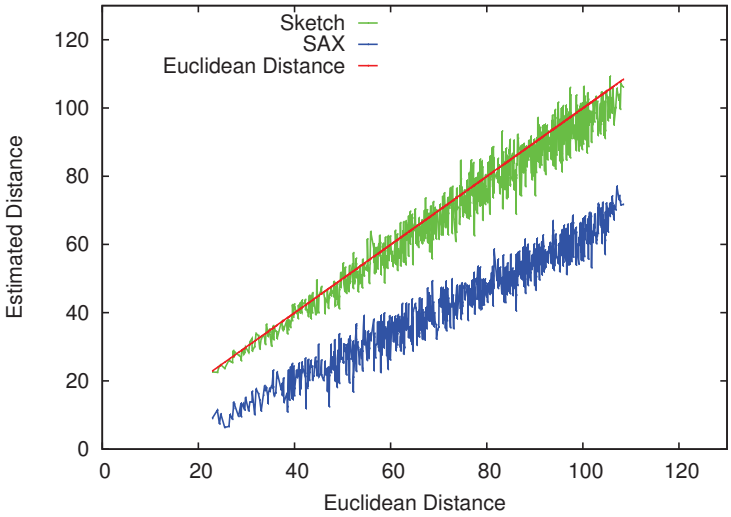


Figure 4.1 – Sketches allow very accurate distance computation compared to the Symbolic Aggregate approXimation (SAX) distance. Here, a comparison on 1,000 couples of time series from our seismic dataset.

ping taking b points in R^m to points in $(R^d)^{2b+1}$ (the (2b+1)-fold cross-product of R^d with itself) approximately preserves distances (with higher fidelity the larger b is).

In our version of this idea, given a point (a time series or a windo of a time series) $\mathbf{t} \in R^m$, we compute its dot product with N random vectors $\mathbf{r}_i \in \{1, -1\}^m$. This results in N inner products called the *sketch* (or random projection) of t_i. Specifically, $sketch(t_i) = (\mathbf{t_i} \bullet \mathbf{r_1}, \mathbf{t_i} \bullet \mathbf{r_2}, ..., \mathbf{t_i} \bullet \mathbf{r_N})$. We compute sketches for $t_1, ..., t_b$ using the same random vectors $r_1, ..., r_N$. By the Johnson-Lindenstrauss lemma [33], the distance $\|\mathbf{sketch(t_i)} - \mathbf{sketch(t_j)}\|$ is a good appproximation of $\|\mathbf{t_i} - \mathbf{t_j}\|$. Specifically, if $\|\mathbf{sketch(t_i)} - \mathbf{sketch(t_j)}\| < \|\mathbf{sketch(t_k)} - \mathbf{sketch(t_m)}\|$, then it's very likely that $\|\mathbf{t_i} - \mathbf{t_j}\| < \|\mathbf{t_k} - \mathbf{t_m}\|$.

Figure 4.1 gives an illustration of the Symbolic Aggregate approXimation distance (SAX distance) and sketch distance, compared to the actual Euclidean distance. This is done for 1,000 couples of random time series from a seismic dataset (detailed in Section 3.2). We report in Figure 4.1 the distance between i) the corresponding sketches of size 120, and ii) the SAX distance, where the cardinality y is 128, the number of segments w is 120, and the time series are not normalized. We did the same experiment with the maximum possible values of parameters for SAX (*i.e.*, the cardinality y is 512 and the

number of segments w is 120) and, even in this case, the SAX distance between time series is still much lower, *i.e.*, approximately half of the actual one in average. The sketch distance turns out to be a better approximation.

In our approach, we use a set of grid structures to hold the time series sketches. Each grid maintains the sketch values corresponding to a specific set of random vectors over all time series. Let $|g|$ be the number of random vectors assigned to each grid, and N the total number of random vectors, then the total number of grids is $b = N/|g|$. The distance of time series in different grids may be different. We consider two time series similar if they are similar in a given (large) fraction of grids.

Example 7. *Let's consider two time series t=(2, 2, 5, 2, 6, 5) and t′=(2, 1, 6, 5, 5, 6). Suppose that we have generated four random vectors as follows : r_1=(1, -1, 1, -1, 1, 1), r_2=(1, 1, 1, -1, -1, 1), r_3=(-1, 1, 1, 1, -1, 1) and r_4=(1, 1, 1, -1, 1, 1). Then the sketches of t and t′, i.e. the inner products computed as described above, are respectively s=(14, 6, 6, 18) and s′=(13, 5, 11, 15). In this example, we create two grids, $Grid_1$ and $Grid_2$, as depicted in figure 4.2. $Grid_1$ is built according to the sketches calculated with respect to vectors r_1 and r_2 (where t has sketch values 14 and 6 and t′ has sketch values 13 and 5). In other words, $Grid_1$ captures the values of the sketches of t and t′ on the first two dimensions (vectors). $Grid_2$ is built according to vectors r_3 and r_4 (where t has sketch values 5 and 18 and t′ has sketch values 11 and 15). Thus, $Grid_2$ captures the values of the sketches on the last two dimensions. We observe that t and t′ are close to one another in $Grid_1$. On the other hand, t and t′ are far apart in $Grid_2$.*

4.1.2 Partitioning Sketch Vectors

In the following, we use correlation and distance more or less interchangeably because one can be computed from the other once the data is normalized. Specifically, the Pearson correlation is related to the Euclidean distance as follows: Here $\hat{x}$ and $\hat{y}$ are obtained from the raw time series by computing

$$\hat{x} = \frac{x - avg(x)}{\sigma_x} \tag{4.1}$$

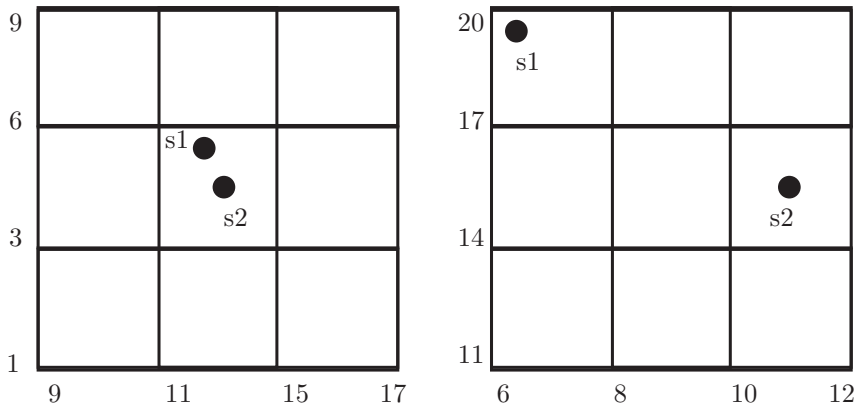


Figure 4.2 – Two series (s_1 and s_2) may be similar in some dimensions (here, illustrated by $Grid_1$) and dissimilar in other dimensions ($Grid_2$). The higher their similarity, the larger the fraction of grids in which the series are close.

$$where\ \sigma_x = \sqrt{\sum_{i=1}^{n}(x_i - avg(x))^2}. \tag{4.2}$$

Multi-dimensional search structures don't work well for more than four dimensions in practice [64]. For this reason, as indicated in Example 7, we adopt a first algorithmic framework that partitions each sketch vector into subvectors and builds grid structures for the subvectors as follows:

- Partition each sketch vector s of size N into groups of some size $|g|$.
- The ith group of each sketch vector s is placed in the ith grid structure (of dimension $|g|$).
- If two sketch vectors s and s' are within distance $c \times d$ in more than a given fraction f of the groups, then the corresponding time series are candidate highly correlated time series and should be checked exactly.

For example, if each sketch vector is of length $N = 40$, we might partition each one into ten groups of size $|g| = 4$. This would yield 10 grid structures. Suppose that the

fraction f is 90%, then a time series t is considered as similar to a searched time series t', if they are similar in at least ninw grids.

4.1.3 Massively Distributed Index construction

Our approach for sketch construction in massively distributed environments proceeds in two steps: 1) Local construction of sketch vectors and groups, on the distributed computing nodes; 2) Global construction of grids, with one computing node per grid.

4.1.3.1 Local construction of sketch vectors and groups

Before distributing the construction of sketch vectors, the master node creates a set of N random vectors of size n, such that each vector $r_i =< r_{i,1}, r_{i,2}, \cdots, r_{i,n} >$, contains n elements. Each element $r_{i,j} \in r_i$ is a random variable in $\{1, -1\}$ with probability 1/2 for each value. Let R be the set of random vectors. R is duplicated on all workers (*i.e.*, processors of the distributed system), so they all share the same random vectors.

Let D be the input dataset involving l times series. Each time series $t \in D$ is of length n: $t =< t_1, t_2, \cdots, t_n >$. D can be represented as a matrix as follows:

$$D = \begin{bmatrix} t_1 =< t_{1,1} & \cdots & t_{1,n} > \\ \vdots & \ddots & \vdots \\ t_l =< t_{l,1} & \cdots & t_{l,n} > \end{bmatrix} \tag{4.3}$$

During the first step of sketch construction, each mapper takes a set of time series $P \subseteq D$, and projects them to the random vectors of R, in order to create their sketches. Let $s_j =< s_{j,1} \;\; s_{j,2} \;\; \cdots \;\; s_{j,N} >$, be the sketch of a time series t_j, then $s_{j,i} \in s$ is the inner product between t_j and r_i. The sketch of t_j can be written as $s_j = t_j \bullet R$.

Let p be the number of time series involved in P, then the result of random projection in a mapper is a collection X of p sketches, each corresponding to one times series of P:

$$X_{N\times l} = \begin{bmatrix} s_1 =< s_{1,1} & \cdots & s_{1,N} > \\ \vdots & \ddots & \vdots \\ s_p =< s_{p,1} & \cdots & s_{p,k} > \end{bmatrix} \tag{4.4}$$

The sketches are partitioned into equal subvectors, or groups, according to the size of sketches and vectors. If, for instance, sketch vectors have length 40, and groups have size four, we partition each vector into ten groups (of size four). For distribution needs, the mapper assigns to each group an ID in $[1..NumberOfGroups]$.

With a sequential construction of groups, where groups are contiguous, the mappers simply have to emit $< key, value >$ pairs where key is the unique ID of a group and $value$ is a tuple made of the data values of the sketch for these dimensions, and the time series ID.

Example 8. *Let us consider s_j the sketch of series t_j, such that $s_j =< 2, 4, 5, 9 >$, and $\{g_1, g_2\}$ the set of two contiguous groups of size two that can be built on s_j (i.e., $g_1 = (s_{j,1}, s_{j,2})$, $g_2 = (s_{j,3}, s_{j,4})$). In the simple version of our approach, this information is communicated to reducers (in charge of building the corresponding grids) by emitting two $< key, value >$ pairs: $< key = g_1, value = ((2, 4), 1) >$ for the information about g_1 and $< key = g_2, value = ((5, 9), 1) >$ for the information about g_2.*

4.1.3.2 Optimized shuffling for massive distribution

In cases when a dimension may be involved in multiple groups, a mapper emits each dimension ID (rather than the group ID) as the key, while the value embeds a couple, made of the data value of the sketch for this dimension and the series ID. The goal is to avoid sending redundant information that is repeated from one group to another. This is even more important when the number of random groups is large, because redundancy increases with the number of overlapping groups. Example 9 illustrates this principle.

Example 9. *Let us consider the sketch and series of Example 8 ($s_j =< 2, 4, 5, 9 >$). Let us now consider $\{g_1, ..., g_5\}$ a set of 5 groups of size two built on $\{s_{j,1}, ..., s_{j,4}\}$, the four dimensions of s_j. Here, $g_1 = (s_{j,1}, s_{j,2})$, $g_2 = (s_{j,1}, s_{j,3})$, $g_3 = (s_{j,1}, s_{j,4})$, $g_4 = (s_{j,2}, s_{j,3})$, $g_5 =$*

$(s_{j,2}, s_{j,4})$ *and there are overlapping groups. The basic approach described in Section 4.1.3.1 aims to emit a* $< key, value >$ *pair, for each group, embedding dimension IDs, data values and time series IDs. However, that implies communicating much redundant information. That would be, for instance,* $key = (s_{j,1}, s_{j,2})$ *and* $value = ((2, 4), 1)$ *for the information about* g_1 *in* s_j*. However,* $s_{j,1}$ *is involved in three different groups and would therefore be emitted three times as part of different keys, resulting in unnecessary communication in the shuffling phase. This is why we choose to separate data transfer and grid construction. Grid construction is partly realized by mappers, and also by reducers. Each mapper will send a single dimension, the corresponding data value and the series ID, so the reducer builds the grid upon receipt. For group* g_1*, for instance, we would emit two pairs. In the first one, we have* $key = (s_{j,1})$ *and* $value = (2, 1)$*. And in the second one, we have* $key = (s_{j,2})$ *and* $value = (4, 1)$*. Then, for group* g_2*, there will be only one pair to emit, where* $key = (s_{j,3})$ *and* $value = (5, 1)$*. This is the same for* g_3 *where only one pair, embedding compact information, has to be emitted.*

4.1.3.3 Global construction of grids

Algorithm 9: Index construction

Input: Data partitions $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ of a database $\mathcal{D}$ and a collection $\mathcal{R}$ of random vectors
Output: Grid structures

```
// Map Task
flatMapToPair( Time Series: T )
    - Project T to R
    - Partition sketch into equal groups
    forall groups do
        emit (key: ID of group, value:(T ID and group data) )

// Reduce Task
reduceByKey( key: ID of group, list(values) )
    - Use the list of values to build a d-dimensional grid structure
    emit (key:ID of group, values: grid structure)
```

Reducers receive local information from mappers, from which they construct grids.

Algorithm 10: Query processing

Input: Grids Structures, a collection $\mathcal{R}$ of random vectors and a collection $\mathcal{Q}$ of Query time series
Output: List of time series

```
// Map Task 1
flatMapToPair( Time Series: T_q )
    - Project T_q to R
    - Partition sketch into equal groups
    forall groups do
        emit (key: ID of group, value: T_q ID and group data)
- Combine the values of the previous job result with the result of previous task,
  where key: ID of group and value:(Grid Structure , list(values))
// Map Task 2
flatMapToPair( key: ID of group, value:(Grid Structure , list(values)) )
    foreach group in values do
        if group ∃ Grid Structure then
            emit (key: T_q ID, value: IDs of the found time series)
// Reduce Task
reduceByKey( key: T_q ID, value: list(values) )
    foreach found time series do
        - Computes the number of occurrence
        if the found time series has the greatest count value then
            emit (key: T_q ID, value: ID of the found time series)
```

More precisely, in the reduce phase, each reducer receives a group ID, and the list of all generated values (group data and sketch ID). It uses the list to build a d-dimensional grid structure. Each grid is stored in a d-dimensional array in the mappers main memory or in HDFS (Hadoop Distributed File System), where each group is mapped to a cell according to its values. The pseudo-code of our index construction in Spark is shown in algorithm 9.

4.1.4 F-RadiusSketch

The above framework circumvents the curse of dimensionality by making the groups small enough that grid structures can be used. Our goal is to achieve extremely high recall (above 0.8) and reasonable precision (above 0.58). Increasing the size of the sketch vector improves the accuracy of the distance estimate but increases the search time. In our experiments, accuracy improved noticeably as the sizes increased to about 256. Beyond that, accuracy does not improve much and performance suffers. Because the dimensions used for comparison in the grids do not have to be disjoint, we build grids based on random and possibly overlapping combinations of dimensions. Our strategy is to choose the same number of combinations as the available processors.

Let the similarity $sim(t, t')$ of two series be the fraction of grids where t and t' fall in the same cell, for all possible grids. We show that by increasing the number of grids, the standard error in the computed similarity of t and q decreases. Let G be the set of all grids which can be generated for the sketches. Suppose p is the percentage of the grids of G, in which q and t are similar. Let G_k be the set of n grids randomly selected from G, and $m(G_k)$ be the fraction of the grids of G_k in which t and q are similar. Let Δ_m be the standard error in $m(G_k)$, *i.e.*, $\Delta_m = |p - m(G_k)|$. The selection of G_k grids from G can be considered as a sampling process. We know from statistics [35] that by increasing the number of samples, the standard error of the mean of samples decreases. Thus, increasing the number of random grids decreases the standard error of $m(G_k)$. When the samples are independent, the standard error of the samples mean is computed as: $\Delta_m = \frac{\delta}{\sqrt{k}}$ where δ is the standard deviation of samples' distribution. The samples aren't independent in our case because the grids may overlap, but this is a suggestive approximation. Our goal in random combinations is to get as close as possible to the best possible results of sketches, in order to lower the error. This goal can be achieved by adjusting the number of random groups where, according to the discussion above, the error decreases with the number of random groups.

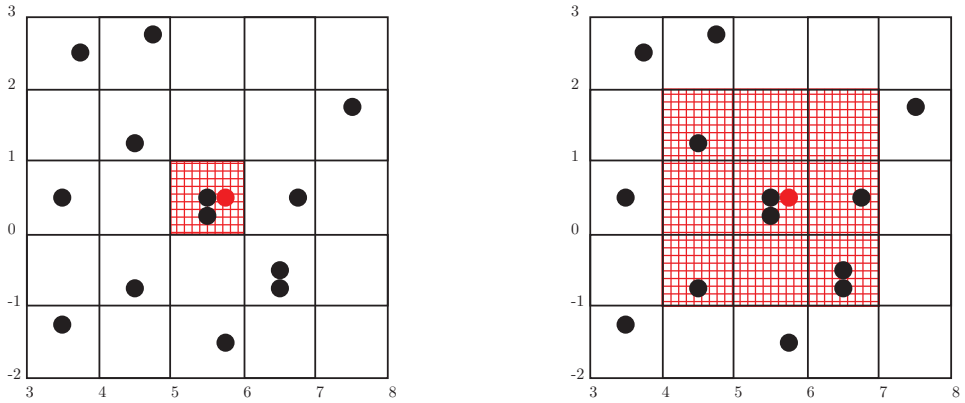


Figure 4.3 – Query processing in RadiusSketch. Each mapper identifies the cell that correspond to a query and emits the IDs of the corresponding time series (a). If there is not enough information the mapper does a broader search in the adjacent cells (b).

4.1.5 Query processing

Given a collection of queries Q, in the form of time series, and the index constructed in the previous section for a database D, we consider the problem of finding time series that are similar to Q in D. We perform such a search in three steps, as follows.

Step 1: map. Each mapper receives a subset of the searched time series $Q' \subset Q$ and the same collection R of random vectors that was used for constructing the index (see Section 4.1.3.1). The mappers generate in parallel the sketch vector for each given time series t in their subset of queries, and partition the sketch vectors into groups (the same dividing principle into groups used for constructing the grid structures is applied). Each mapper emits the ID of groups as the key, and the sketch ID (*i.e.*, query ID) coupled with group's sketch data as value.

Step 2: map. Each mapper takes one or several grid structures of the index and the emitted groups of step 1 that correspond to the chosen grids. For each sketch of a searched time series t in a group, the mapper checks in the corresponding grid, the cell that contains data points similar to t, where each data point contains the ID of the corresponding times series in the grid structure as depicted in Figure 4.3a. For each time series $t \in Q$, and for each times series t' that belongs to the same cell as t in the grid

structure, the mapper emits a key-value pair, where the key is the ID of t, and the value is the ID of t'.

Step 3: reduce. In the reduce phase, each reducer computes for each given key (*i.e.*, the ID of the searched time series) the count of each emitted value, *i.e.*, the IDs of the found time series in different grids. Then, for the searched time series, the reducer emits to HDFS the ID of the time series that has the greatest count value.

The pseudo-code of query processing for Spark is given by Algorithm 10.

Searching for the k-nearest neighbours (k-NN) of the time series Q is done as in the previous section 4.1.5 in three steps steps. The difference is that in Step 2, each mapper returns for each searched time series, k candidate times series from the grid. In addition, in Step 3, for each query t, k candidates that have the highest counts are returned as the answer to the query t. Sometimes, in Step 2, the mapper does not find enough data points in cell c, leading to a lack of information for time series retrieval on the third step. In such cases, all the neighbors of c will be visited until k points are found, as depicted in figure 4.3b.

4.2 Experiments

In this section, we report experimental results that show the quality and the performance of our parallel solution for indexing time series.

We evaluate the performance of two versions of our solution: 1) RadiusSketch that is the basic version of our parallel indexing approach with partitioning; 2) F-RadiusSketch (Fully Parallel Sketch) that includes RadiusSketch and a random overlapping combination technique that improves quality by using idle machines to create new groups whose sketch indexes overlap with the partitioned groups. We compare our solutions with the most efficient version of the iSAX index (*i.e.*, iSAX2+2) proposed in [12]. We implemented two versions of our approach, one for centralized environments and the other version on top of Apache-Spark [78] for a distributed environment, using the Java programming language. The iSAX2+ index [12] is also implemented with Java, in a central-

ized version only.

The parallel experimental evaluation was conducted on a cluster of 32 machines, each operated by Linux, with 64 Gigabytes of main memory, and Intel Xeon X5670 CPU and 250 Gigabytes hard disk. The centralized versions of sketches and iSAX2+ were executed on a single machine with the same characteristics.

Our experiments are divided into two sections. In Section 4.2.2, we measure the grid construction times with different parameters. In Section 4.2.3, we focus on the query performance of the sketch approach, both in terms of response time and accuracy.

4.2.1 Datasets and Setting

4.2.1.1 Datasets

We carried out our experiments on both real-word and synthetic datasets. The first one is a seismic dataset of 40 million time series, where each time series has a length of 256. It has a total size of 491 Gigabytes. For the second one, we generated a dataset of 500 million time series using a random walk data series generator, each data series consisting of 256 points. At each time point the generator draws a random number from a Gaussian distribution N(0,1), then adds the value (which may be negative) of the last number to the new number. The total size of our synthetic dataset is 1 Terabytes.

4.2.1.2 Parameters

Table 4.1 shows the default parameters (unless otherwise specified in the text) used for each approach. For Sketch and RadiusSketch, the number of groups is given by $SketchSize/\ GroupSize$. For F-RadiusSketch, the number of groups may be up to 256, depending on the number of exploited cores. When necessary, parameters are specified in the name of the approach reported in our experiments. For instance, $Sketch(4, 120)$ stands for the sketch approach with group size = 4 and sketch size = 120 (and the number of groups is $120/4 = 30$, since this is the default number of groups) while

$F - RadiusSketch(2, 60, 256)$ stands for F-RadiusSketch with groups of size 2, sketches of size 60 and the number of groups is 256

Method	Parameters	Method	Parameters
F-RadiusSketch	Group size = 2 Sketch size = 60 Number of groups = 256	iSAX2+	Threshold = 8,000 Word length w = 8
RadiusSketch	Group size = 2 Sketch size = 120	Sketch	Group size = 2 Sketch size = 60

Table 4.1 – Default parameters

4.2.2 Grid Construction Time

In this section, we measure the index construction time in RadiusSketch and F-RadiusSketch, and compare it to the construction time of the iSAX2+ index.

Figures 4.4 and 4.5 report the index construction times for both of the tested datasets. The index construction time increases with the number of time series for all approaches. In our distributed testbed, the index construction time is lower than it is in a centralized environment, with time reduced almost linearly. Figure 4.4 reports the construction time of centralized approaches (iSAX2+ and sketches) in days, while the scale unit is in minutes for RadiusSketch (60 groups of size 2) and F-RadiusSketch (256 groups of size 2). For 500 million time series, on the random walk dataset, the RadiusSketch index is built in 35 minutes on 32 machines, while the iSAX2+ index is built in more than 3 days on a single node.

To illustrate the parallel speed-up of our approach, Figures 4.6a and 4.6b show the relationship between the execution time and the number of nodes. For both of our approaches, we report the total construction time with and without I/O cost (*e.g.*, RadiusSketch-I/O is without I/O cost). The results illustrate a near optimal gain for F-RadiusSketch on the random walk dataset. For instance, the construction time is almost 60 minutes without I/O cost with 8 nodes, and drops down to 30 minutes without I/O cost for 16 nodes

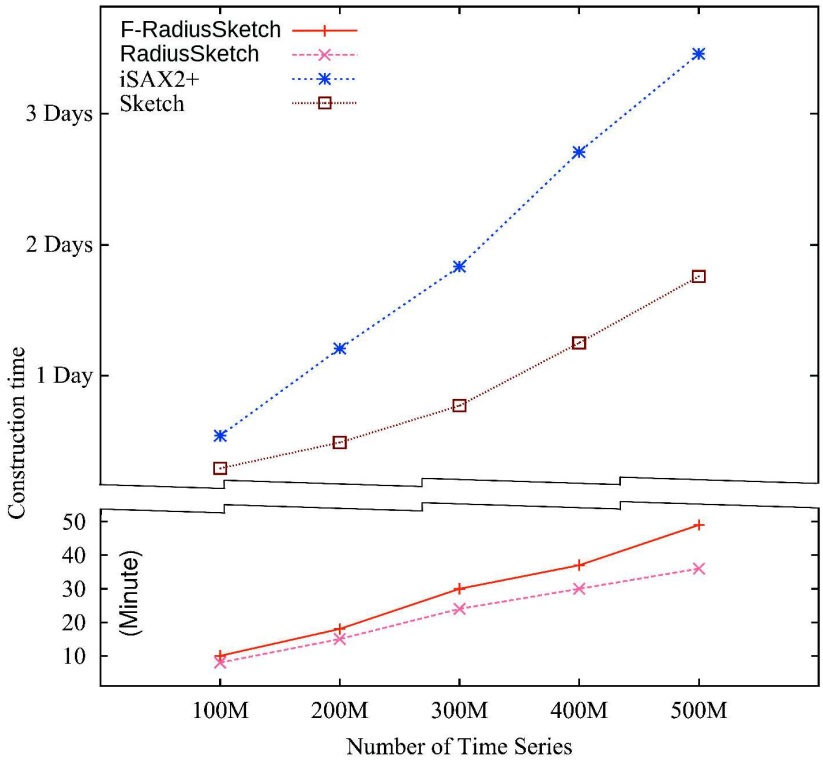


Figure 4.4 – Construction time as a function of dataset size (random walk dataset). Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.

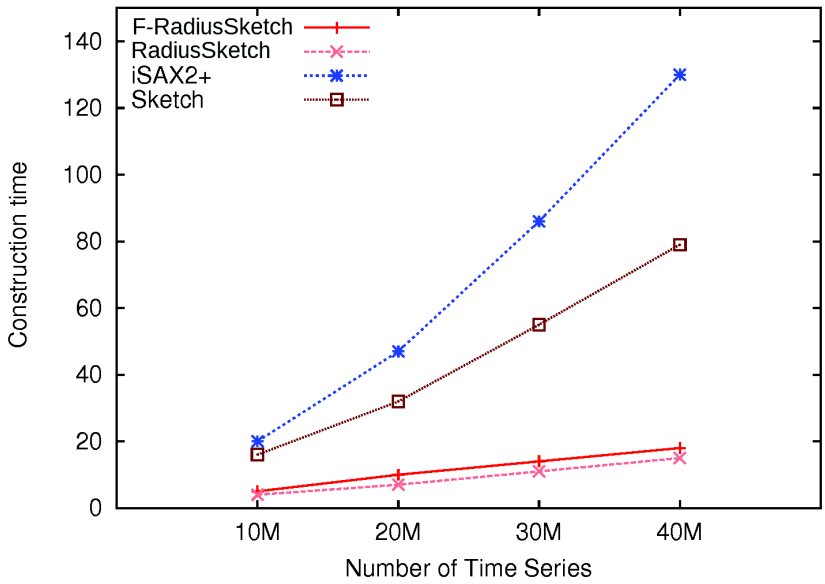
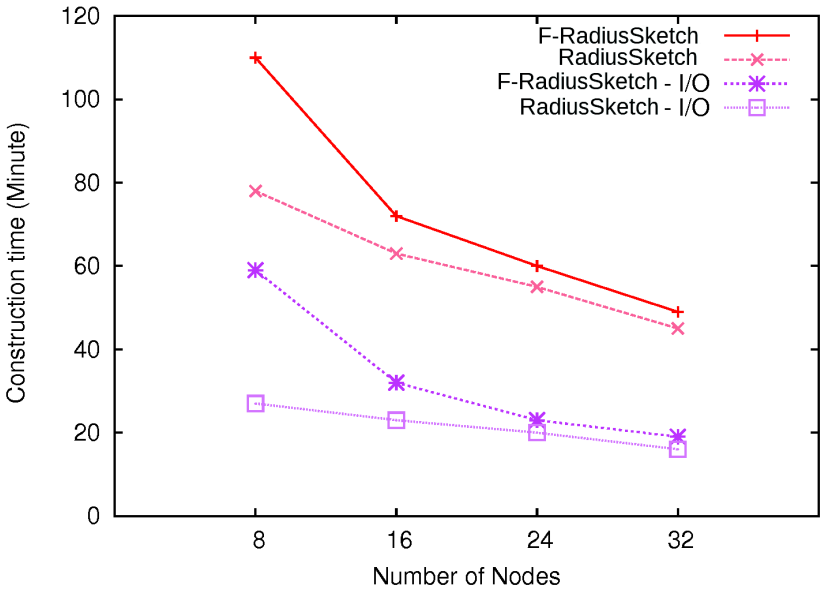
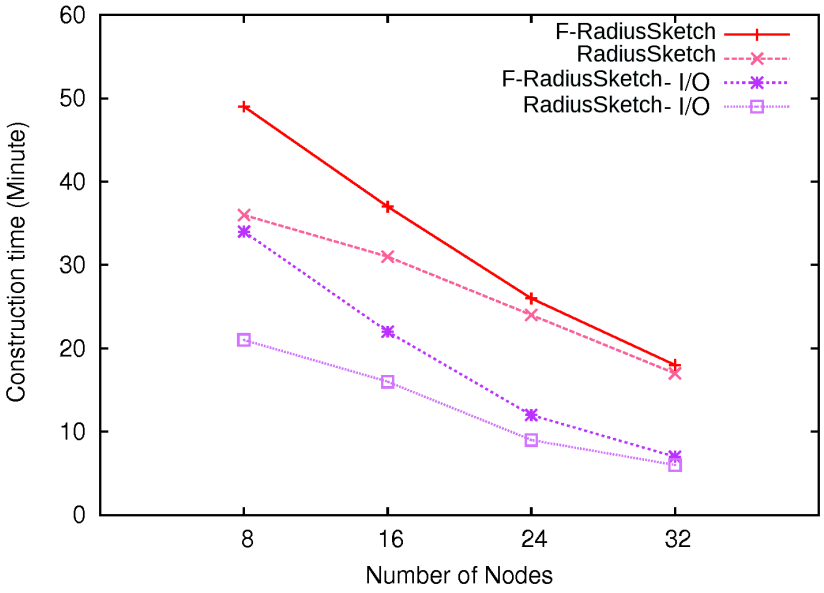


Figure 4.5 – Construction time as a function of dataset size (seismic dataset). Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.

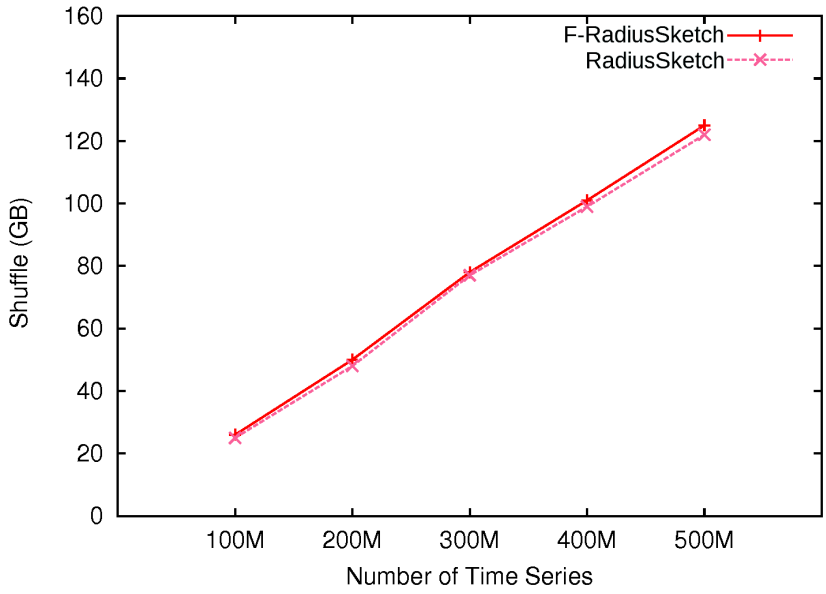


(a) random walk dataset

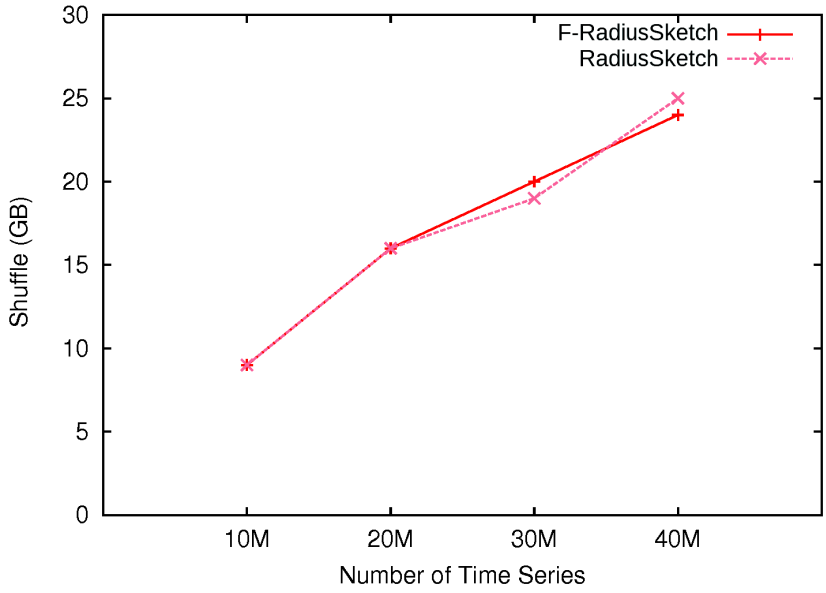


(b) seismic dataset

Figure 4.6 – Construction time as a function of cluster size. F-RadiusSketch has a near optimal parallel speed-up on the random walk dataset.

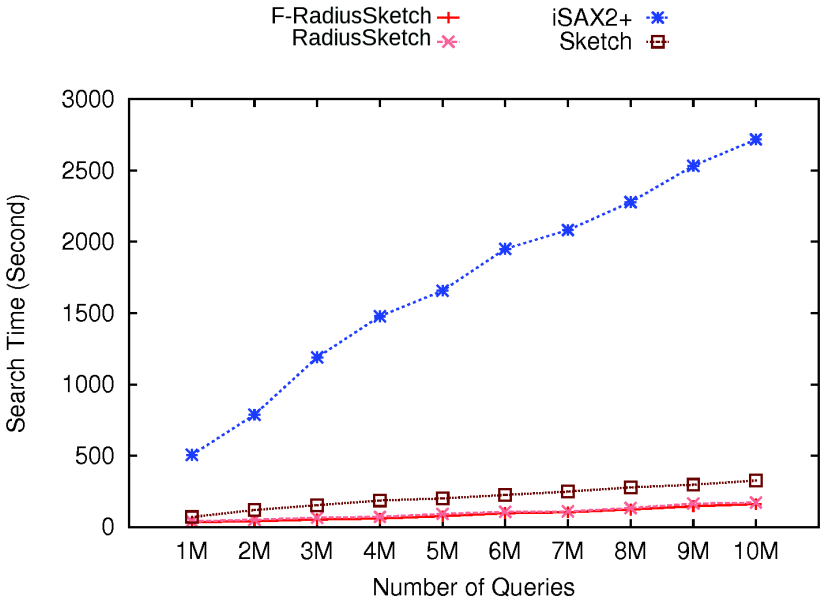


(a) random walk dataset

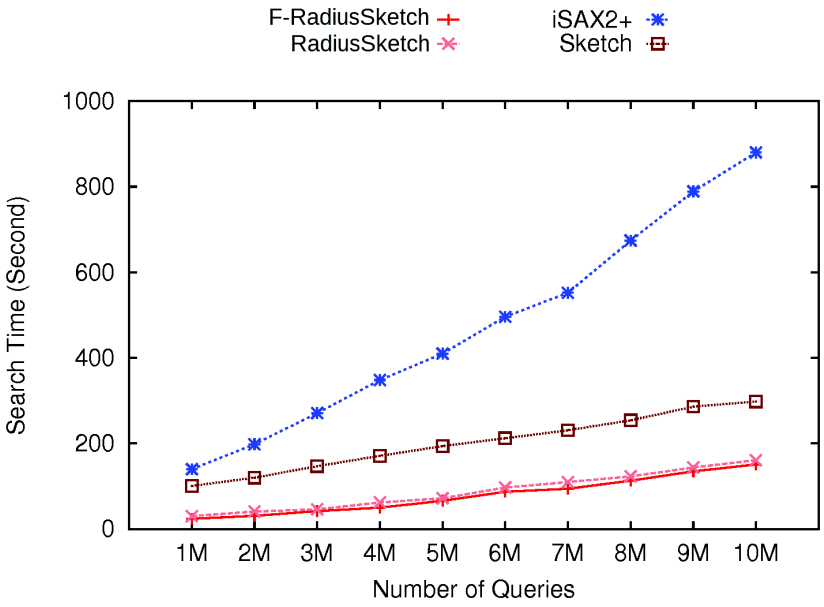


(b) seismic dataset

Figure 4.7 – Shuffling as a function of dataset size. The shuffling costs of RadiusSketch and F-RadiusSketch increase linearly.

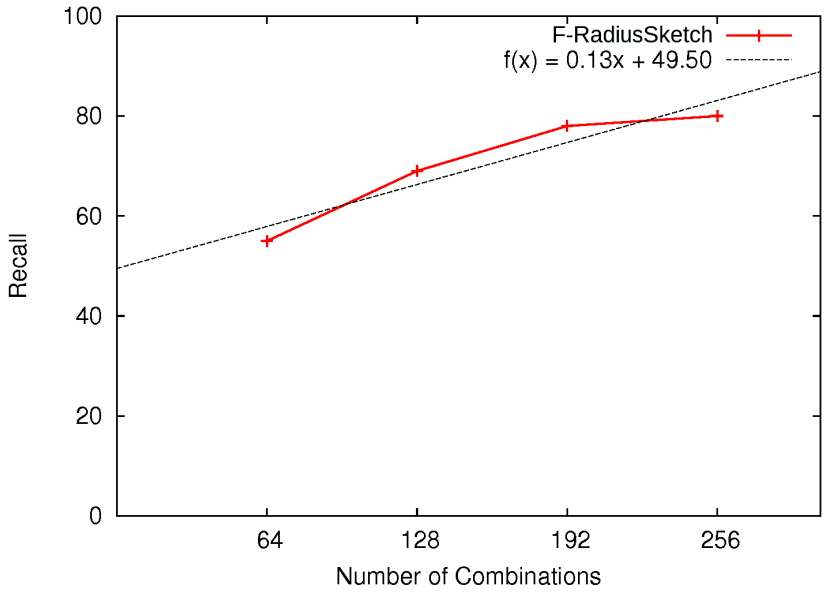


(a) random walk dataset

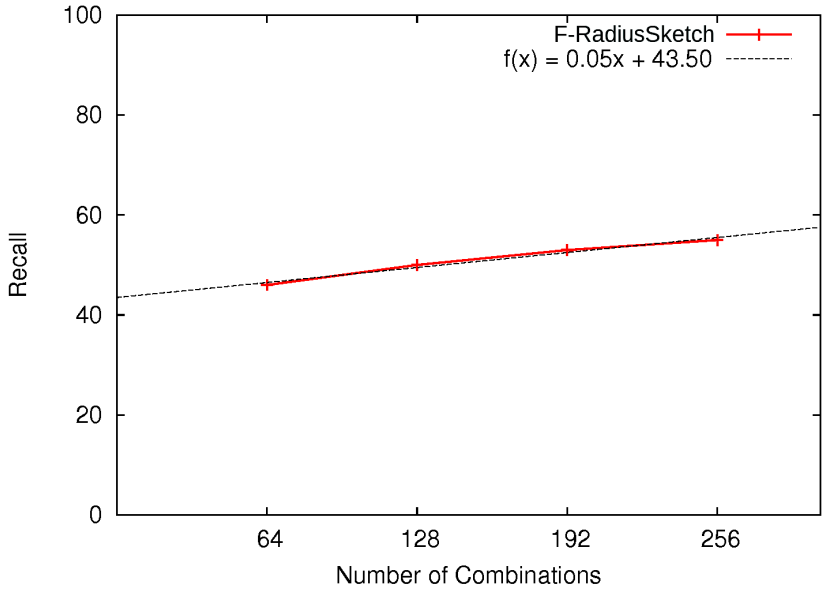


(b) seismic dataset

Figure 4.8 – Search time of sketch versions and iSAX2+. Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.

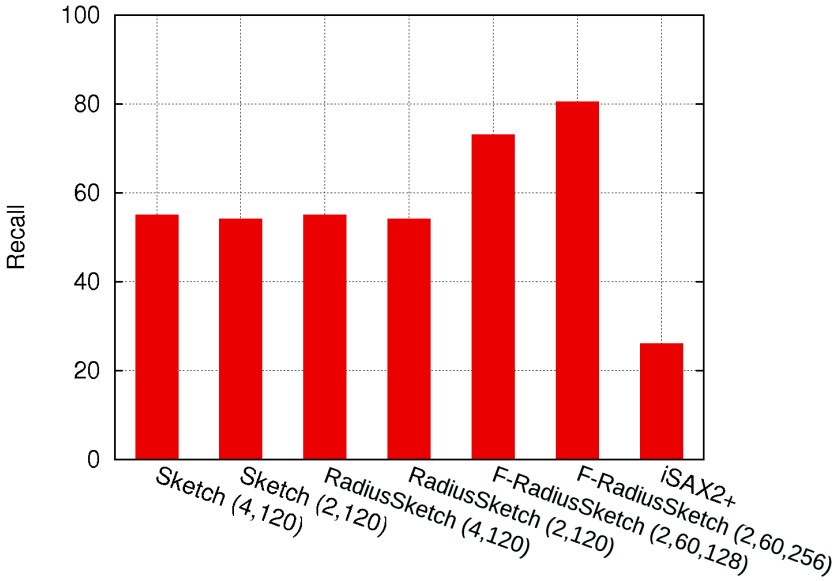


(a) random walk dataset

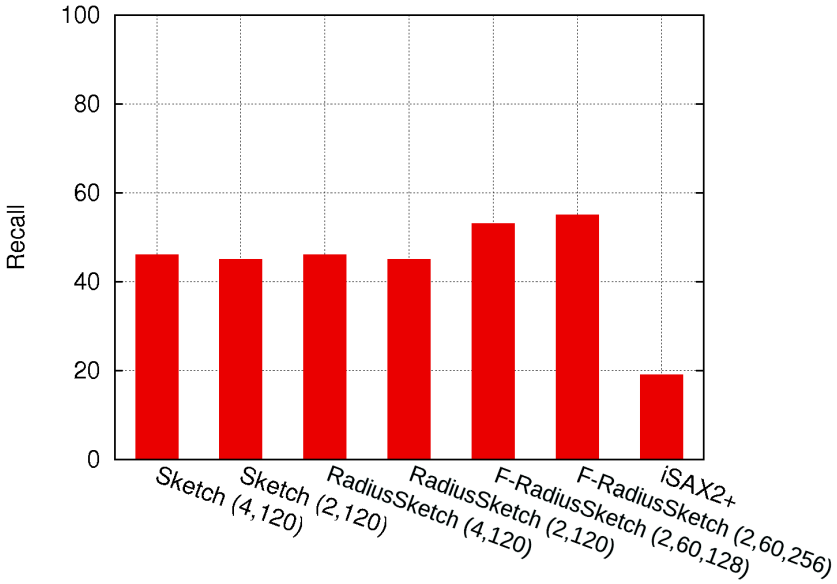


(b) seismic dataset

Figure 4.9 – The effect of the number of combinations on recall is roughly logarithmic and monotonically increasing (Avg. value for 1M queries).

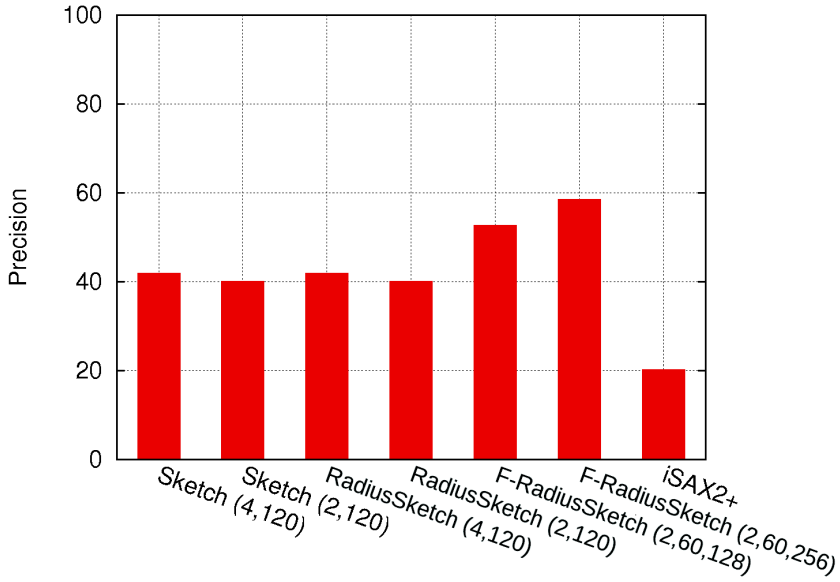


(a) random walk dataset

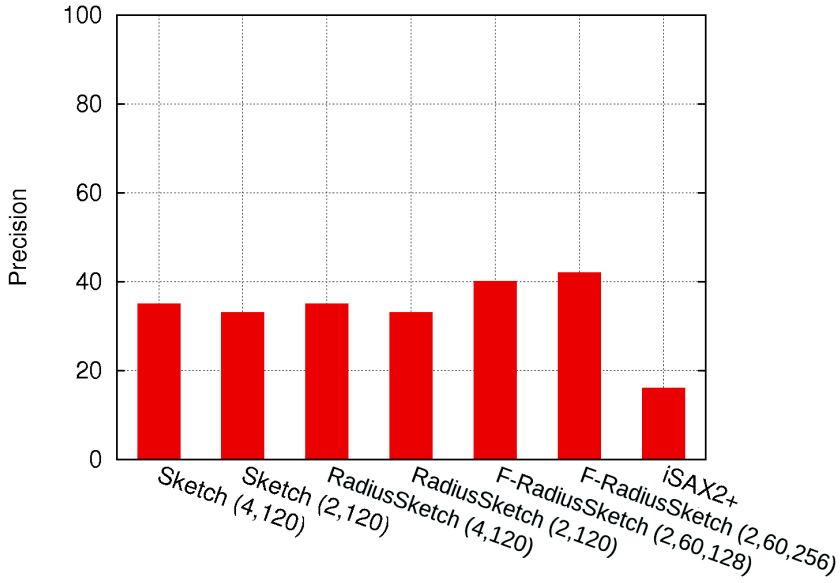


(b) seismic dataset

Figure 4.10 – Recall of sketches and iSAX2+ (Avg value for 1M queries). Increasing the number of grids with F-RadiusSketch gives higher recall. Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.



(a) random walk dataset



(b) seismic dataset

Figure 4.11 – Precision of sketches and iSAX2+ (Avg value for 1M queries). Increasing the number of grids with F-RadiusSketch gives higher precision. Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.

(*i.e.*, the in-memory construction time is reduced by a factor of two when the number of nodes is doubled).

Figures 4.7a and 4.7b show the shuffling cost for a varying number of time series. We observe that the shuffling cost increases linearly, which illustrates that our approaches are able to scale on massively distributed environments. We also observe a very similar shuffling cost between F-RadiusSketch and RadiusSketch. This result is mainly due to the shuffling optimization presented in Section 4.1.3.2.

4.2.3 Query Performance

Given a query q, let TP, TN, FP and FN be the true positive/negative and false positive/negative results of an index, respectively. To evaluate the retrieval capacity of an index, we consider two measures:

- **Recall:** we search for the 20 most similar series to q according to the index. Then, we compare the result to a linear search with q on the whole dataset, where the top 10 similar series are returned. The number of true positive candidate series returned by the index is counted among the top 20 series given by the index.
- **Precision:** here, the same principle is applied but restricted to the top 10 series returned by the index.

In both cases, we set $precision = VP/(VP + FP)$ and $recall = VP/(VP + FN)$. In the following experiments, for the seismic dataset the queries are time series randomly picked from the dataset. For the random walk dataset, we generate random queries with the same distribution. For each time series t in the query, the goal is: i) to check if the approach is able to retrieve t (if it exists in the case of random walk); and ii) to find the nine other time series that are considered to be the most similar to t in the dataset.

Figures 4.8a and 4.8b compare the search time of the sketch approaches to that of iSAX2+ for answering queries with a varying size of query batch. These figures show that, in our experiments, the search time of RadiusSketch and F-RadiusSketch is better

than that of the iSAX2+ by a factor of 13, *e.g.*, the search time for 10 million queries is 2700s for iSAX2+ and 200s for F-RadiusSketch.

Figures 4.9a and 4.9b illustrate the impact of the number of combinations (i.e number of groups) on the recall of F-RadiusSketch. We observe that the recall increases with the number of groups. For instance, when the number of groups is 256, we observe a recall of 0.80 for the random walk dataset and 0.55 for the seismic dataset. This shows the trend of the recall as a function of the number of combinations. The effect is roughly logarithmic and monotonically increasing.

Figures 4.10a and 4.10b illustrate the recall of different tested approaches, with varying parameters for the sketch approaches. For all the settings, the recall performance of sketches is higher than iSAX2+. We observe that F-RadiusSketch outperforms all the other approaches when the number of combinations is maximum. For instance, with 256 groups, the recall of F-RadiusSketch is up to 80%, while that of iSAX2+ is 26%.

The same experiment has been done to study precision, with very similar results as reported in Figures 4.11a and 4.11b.

4.3 Conclusion

RadiusSketch is a simple-to-implement high performance method to perform similarity search at scale. It achieves better runtime performance and better quality than its state-of-the-art competitor iSAX2+ in a sequential environment. Further, RadiusSketch parallelizes naturally and nearly linearly.

Chapter 5

Efficient Parallel Methods to Identify Similar Time Series Pairs Across Sliding Windows

In this chapter, we address the problem of finding the highly correlated pairs of time series over a time window and then sliding that window to find the highly correlated pairs over multiple windows such that each successive window starts only a little time after the previous window. Doing this efficiently and in parallel could help in applications such as sensor fusion, financial trading, or communications network monitoring, to name a few. We propose a parallel incremental sketching approach that gives linear speedup over most of its steps and reduces the quadratic work by communicating time series identifiers instead of time series themselves. We compare our approach with the state of the art nearest neighbor method iSAX [12].

The rest of this chapter is organized as follows. In Section 5.1, we give the motivation and an overview of our work. In Section 5.2, we formally define the problem we address, and in Section 5.5, we describe the details of our solution. In Section 5.4, we evaluate the performance of our solution through experiments in a distributed environment using real and synthetics datasets. Finally, in Section 5.5 we conclude.

5.1 Motivation and Overview of the Proposal

An easy-to-understand motivating use case for finding sliding windows correlation comes from finance. In that application, the time series consist of prices of trades of different stocks. The problem is to find pairs of stocks whose return profiles look similar over the most recent time period (typically, a few seconds). A pair of time series (e.g. Google and Apple prices) that were similar before and have since diverged, where say Google went up more than Apple, might present a trading opportunity: sell the one that has gone up relative to the other and buy the other one. The return profile is based on the weighted average price (by volume) of the stock over time t (perhaps discretized in milliseconds), denoted wprice(t). The return at t is the fractional change, $(wprice(t) - wprice(t-1))/wprice(t-1)$.

While prices are stable over time (e.g. a stock whose price is 100 will tend to stay around 100), the returns resemble white noise. We call such time series "uncooperative", because standard dimensionality reduction techniques such as Fourier or Wavelet Transforms either sacrifice too much accuracy or reduce the dimensionality too little. Random sketch-based methods and some other explicit encoding methods work well for both cooperative and uncooperative time series. Moreover, the sketch-based methods work nearly as well as Fourier/Wavelet methods for cooperative time series. So, for the sake of generality, in this work we use the sketch method of [17], and compare the result with the state-of-the-art explicit encoding method iSax [12].

The need for speed comes from increasing scale and the advantage of reacting quickly. An irony of improving technology is that sensor speeds and numbers increase vastly faster than computational speed. For this reason, linear or near linear-time algorithms become increasingly vital to give timely responses in the face of the flood of data. In most applications, speed turns out to be of greater importance than completeness, so a minor loss in recall is often acceptable as long as precision is high. In trading, for example, there is only a fictitious monetary loss in missing an opportunity, but the opportunities a system reports should be real and must be timely to be actionable.

Our motivating example comes from sensor fusion for earth science. Correlations of distant sensors in seismic data may indicate a large scale event. Consider, for example,

a set of sensors spaced over several possible earthquake zones. Temporal correlations of pairs of sensors over a time window may suggest that these pairs are responding to the same seismic cause. Missing some correlations is acceptable, because a major event will reveal many correlations so a recall of 90% or more is quite enough.

In this work, we propose ParCorr, an efficient parallel solution for detecting similar time series across sliding windows. ParCorr uses the sketch principle for representing the time series. Our ParCorr solution includes the following contributions:

- A parallel approach for incremental computation of the sketches in sliding windows. This approach avoids the need for recomputing the sketches from scratch, after the modifications in the content of the sliding window.
- A partitioning approach that projects sketch vectors of time series into subvectors and builds a distributed grid structure for the subvectors in parallel. Each subvector projection can be processed in parallel.
- An efficient algorithm for parallel detection of correlated time series candidates from the distributed grids. In our algorithm, we minimize both the size and the number of messages needed for candidate detection.

5.2 Problem Definition

A streaming time series is a potentially unending series of values in time order. A data stream, for our purposes, is a set of streaming time series. They are normalized to have zero mean and unit standard deviation. Correlation over windows from the same or different series has many variants. This work focuses on the synchronous variation, defined as follows:

Given a data stream of N_s streaming time series, a start time p_s, and a window size w, find, for each time window W of size w, all pairs of streaming time series ts_1 and ts_2 such that ts_1 during time window W is highly correlated (over 0.7 typically) with ts_2 during the same time window.

Euclidean distance is the target metric of the state of the art iSAX algorithm. In addition, Euclidean distance is related to Pearson correlation as follows:

$$D^2(\hat{x}, \hat{y}) \quad = \quad 2 \times m \times (1 - corr(x, y)) \tag{5.1}$$

Here $\hat{x}$ and $\hat{y}$ are obtained from the raw time series by computing $\hat{x} = \frac{x - avg(x)}{\sigma_x}$, where $\sigma_x = \sqrt{\sum_{i=1}^{m}(x_i - avg(x))^2}$. m is the length of the time series. So, we offer parallel algorithms for both sliding window Euclidean and correlation metrics in this work.

5.3 Algorithmic Approach

Following [17], our basic approach to find similar pairs of sliding windows in time series (whether Euclidean distance or Pearson correlation, for starters) is to compute the dot product of each normalized time series over a window size w with a set of random vectors. That is, for each time series t_i and window size w and time period $k..(k+w-1)$, we compute the dot product of $t_i[k..k+w-1]$ with r random $(-1/+1)$ vectors of size w. The r dot products thus computed constitute the "sketch" of t_i at time period $k..(k+w-1)$. Next we compare the sketches of the various time series to see which ones are close in sketch space (if $w >> r$, which is often the case, this is cheaper than working directly on the time series) and then identify those close ones.

The theoretical underpinning of the use of sketches is given by the Johnson-Lindenstrauss lemma [33].

Lemma 1. *Given a collection C of m time series with length n, for any two time series $\overrightarrow{x}, \overrightarrow{y} \in C$, if $\epsilon < 1/2$ and $n = \frac{9logm}{\epsilon^2}$, then*

$$(1 - \epsilon) \leq \frac{\| \overrightarrow{s}(\overrightarrow{x}) - \overrightarrow{s}(\overrightarrow{y}) \|^2}{\| \overrightarrow{x} - \overrightarrow{y} \|^2} \leq (1 + \epsilon)$$

holds with probability $1/2$, where $\overrightarrow{s}(\overrightarrow{x})$ is the Gaussian sketch of $\overrightarrow{x}$ of at least n dimensions.

The Johnson-Lindenstrauss lemma implies that the distance $\|\mathbf{sketch}(\mathbf{t_i}) - \mathbf{sketch}(\mathbf{t_j})\|$ is a good appproximation of $\|\mathbf{t_i} - \mathbf{t_j}\|$. Specifically, if $\|\mathbf{sketch}(\mathbf{t_i}) - \mathbf{sketch}(\mathbf{t_j})\| < \|\mathbf{sketch}(\mathbf{t_k}) - \mathbf{sketch}(\mathbf{t_m})\|$, then it's very likely that $\|\mathbf{t_i} - \mathbf{t_j}\| < \|\mathbf{t_k} - \mathbf{t_m}\|$.

The sketch approach, as developed by Kushilevitz et al. [42], Indyk et al. [28], and Achlioptas [2] makes use of these guarantees. Note that the sketch approach is closely related to Locality Sensitive Hashing [25], by which similar items are hashed to the same buckets with high probability. In particular, the sketch approach is very similar in spirit to SimHash [14], in which the vectors of data items are hashed based on their angles with random vectors. The major contribution of our work consists of combining an incremental strategy with a parallel mixing algorithm and an efficient communication strategy.

5.3.1 The case of sliding windows

In the case of sliding windows, we want to find the most similar time series pairs at jumps of a basic window b, e.g. for windows in time ranges 0 to $w - 1$ seconds, b to $b + w - 1$ seconds, $2b..2b + w - 1$, ... where $b << w$.

There are two main challenges:

1. If we compute the sketches from scratch at each basic window, we are doing some redundant computation. We call that the naive method. Instead, we want to compute the sketches incrementally and in parallel.

2. When scaling this to a parallel system, we want to reduce communication costs as much as possible. We need to develop good strategies for this as communication is quadratic in the number of execution nodes, so constant coefficients matter.

5.3.2 Parallel incremental computation of sketches

To explain the incremental algorithm consider the example of Figure 5.1. The sketch for random vector v_1 is the dot product of the time series with v_1, i.e. $1 \times (-1) + 2 \times 1 + ... +$

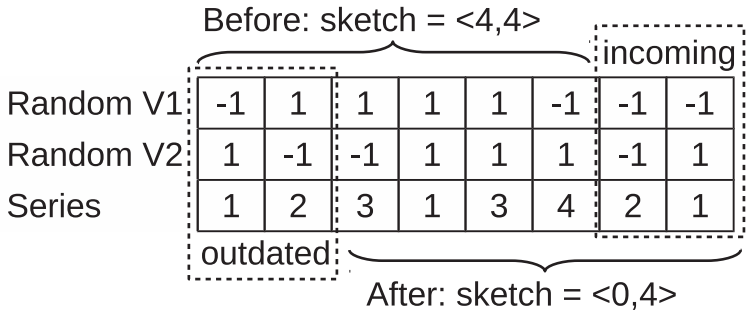


Figure 5.1 – A streaming time series, two random vectors, and the sketches that correspond to their dot product before and after the update on the data stream. The first sketch of the time series is computed on the six first values, and the second sketch is computed on the six last values. After the update, the "outdated" values are removed and the "incoming" ones are added to the streaming time series, so the work is proportional to the size of the basic window rather than the full window.

$4 \times (-1) = 4$.

Now if the basic window is of size 2, as illustrated by the "outdated" and "incoming" boxes of Figure 5.1, then we add the next two points of the time series (in this case having values $(2, 1)$ and generate two more random +/- 1 numbers for each random vector, in this case $(-1, -1)$ for v_1 and $(-1, 1)$ for v_2. To update the dot product for v_1 we subtract the contribution of the oldest two time points, viz. $1 \times (-1) + 2 \times 1 = 1$, and add in the contribution of $2 \times (-1) + (1 \times -1) = -3$ yielding a new sketch entry of $4 - 1 + (-3) = 0$. That illustrates the idea of incremental updating.

In general, the algorithm proceeds as following:

1. Partition time series among parallel sites. Replicate r random +1/-1 vectors each of size w to all sites. These random vectors will later be updated in a replicated fashion.

2. For each site,

(a) Initially, take the first w data points of each time series at that site and form the dot product with all r random vectors. So each time series t will be represented by r dot products. They constitute $sketch(t)$.

(b) while data comes in, when data for the i_{th} basic window of size b appears for all time series, extend each random vector by a new random +1/-1 vector of size b. Then for each time series t and random vector v, update the dot product of t with v by subtracting $v[0..b-1] \cdot t[(i-1)b-w...ib-w-1]$ and adding $v[w..w+b-1] \cdot t[(i-1)b..ib-1]$. Change the $sketch(t)$ with all the updated dot products.

This step has time complexity proportional to the number of time series $\times$ size of basic windows $\times$ the number of random vectors. It is perfectly parallelizable.

Step 1 calls for parallel updates of the local random vectors on each site. It is mandatory that all the sites share the same random vectors. A possible approach would be for the master node, after the completion of each new sliding window, to generate new vectors of +/- 1 having the basic window size, and send them to the sites. This takes little time but is awkward to do in Spark. Our approach is therefore to generate and send oversized random vectors (say, twice the size of the sliding window) at setup time. A site then just has to loop inside the (oversized) random vector, simulating an endless source of +/- 1 values that are the same for all the sites.

5.3.3 Parallel mixing

Once the sketch vectors have been constructed incrementally, the next step is to find sketch vector pairs that are close to one another. Such pairs might then indicate that their corresponding time series are highly correlated (or similar based on some other distance metric).

Multi-dimensional search structures do not work well for more than four dimensions in practice [64]. For this reason, as indicated in the following example, we adopt a framework that partitions each sketch vector into subvectors and builds grid structures for the subvectors.

We first explain how this works by example and then show the pseudo-code.

Example 1. Suppose we have seven time series with sketch values as shown in Table 5.1.

Table 5.1 – A sample S of 7 time series with sketch values of length 6

time series	sketch values
sketch(ts_1)	(11, 12, 23, 24, 15, 16)
sketch(ts_2)	(11, 12, 13, 14, 15, 16)
sketch(ts_3)	(21, 22, 13, 14, 25, 26)
sketch(ts_4)	(21, 22, 13, 14, 25, 26)
sketch(ts_5)	(11, 12, 33, 34, 25, 26)
sketch(ts_6)	(31, 32, 33, 34, 15, 16)
sketch(ts_7)	(21, 22, 33, 34, 15, 16)

First, we partition these into pairs and send the values [0 1] of each sketch vector to site 1 (Table 5.2) where this will be formed into a grid (and the time series identifiers will be placed in cells (i,j), e.g., (31,32)). Analogously, we send partitions [2 3] and [4 5] of each sketch vector to sites 2 and 3 respectively, where the second and third grids will be formed. In the first grid, ts_1, ts_2, and ts_5 map to the same grid cell; ts6 is by itself; and ts_3, ts_4, and ts_7 all map to the same cell (Table 5.3). Thus, in grid 1 we have three partitions of time series identifiers. If two time series are in the same partition, then they are candidates for similarity.

Now we construct a mapping ts_to_node that maps time series identifiers to nodes (for now, think of each node as a single computational site, but one could imagine placing many nodes on a single site or spreading a node among many sites). For this example, let us say ts_to_node is the identity function. So we send the relevant parts of the partition ts_1, ts_2, ts_5 to nodes 1, 2, and 5. Similarly, we send the relevant parts of ts_3, ts_4, and ts_7 to nodes 3, 4, and 7. And so on (Table 5.4). Assuming the "opt" communication strategy (see next subsection), the relevant part of a partition with respect to a time series t consists of t itself and the time series with identifiers higher than t. We call that a "candidate cluster of time series". We ignore clusters with just one element, as pairs cannot be derived out of them.

Table 5.2 – Step 1 of the algorithm: sketch partitioning. Each sketch vector is partitioned into three pairs. The ith pair of the sketch vector for each time series s goes to a grid i. The values of the ith pair determine where in that grid the identifer s is placed.

	sketch subvectors		
	[0 1]	**[2 3]**	**[4 5]**
sketch(ts_1)	(11, 12)	(23, 24)	(15, 16)
sketch(ts_2)	(11, 12)	(13, 14)	(15, 16)
sketch(ts_3)	(21, 22)	(13, 14)	(25, 26)
sketch(ts_4)	(21, 22)	(13, 14)	(25, 26)
sketch(ts_5)	(11, 12)	(33, 34)	(25, 26)
sketch(ts_6)	(31, 32)	(33, 34)	(15, 16)
sketch(ts_7)	(21, 22)	(33, 34)	(15, 16)
	assigned to grid / at site		
	1	**2**	**3**

Table 5.3 – Step 2 of the algorithm: grid construction. Time series placed in the same grid cells are grouped in partitions.

grid	**cell**	**time series IDs**
1	(11, 12)	ts_1, ts_2, ts_5
	(21, 22)	ts_3, ts_4, ts_7
	(31, 32)	ts_6
2	(13, 14)	ts_2, ts_3, ts_4
	(23, 24)	ts_1
	(33, 34)	ts_5, ts_6, ts_7
3	(15, 16)	ts_1, ts_2, ts_6, ts_7
	(25, 26)	ts_3, ts_4, ts_5

Let us say we require that some fraction f of the grids should put two time series in the same grid cell for us to be willing to consider that pair of time series to be worth checking in detail. For this example, set f to 2/3 (Figure 5.2).

Each node takes care of those time series that map to that node. So for example, node 1 shows that ts_1 and ts_2 satisfy the requirement. Node 2 shows nothing new concerning ts_2. Node 3 shows that ts_3 and ts_4 satisfy the requirement. Node 4 and node 5 show nothing new concerning ts_4 and ts_5 respectively. Node 6 shows that ts_6 and ts_7 satisfy the requirement. Node 7 shows nothing new (in fact the last node will never

Table 5.4 – Steps 4 and 5 of the algorithm: finding frequently collocated pairs (in the example, at least 2 out of 3 grids).

node	**TS clusters**	**candidate pairs f ≥ 2/3**
ts_to_node(ts_1)	ts_1, ts_2, ts_5 ts_1, ts_2, ts_6, ts_7	ts_1, ts_2
ts_to_node(ts_2)	ts_2, ts_5 ts_2, ts_3, ts_4 ts_2, ts_6, ts_7	
ts_to_node(ts_3)	ts_3, ts_4, ts_7 ts_3, ts_4 ts_3, ts_4, ts_5	ts_3, ts_4
ts_to_node(ts_4)	ts_4, ts_7 ts_4, ts_5	
ts_to_node(ts_5)	ts_5, ts_6, ts_7	
ts_to_node(ts_6)	ts_6, ts_7 ts_6, ts_7	ts_6, ts_7

show anything new, so need not be considered). All those that satisfy the requirement can be tested for direct correlation. In this example, this would entail computing correlations on the last windows of length w of ts_1 and ts_2; ts_3 and ts_4; and ts_6 and ts_7 (Table 5.4).

Generalizing from this example, here is the algorithm:

1. Partition the sketch vectors, which all have length r, into groups of size k (e.g. if r is 60 and k is 2, then the partition would be 0,1, 2,3, 4,5, ..., 58,59 and we would take indexes 0 and 1 of each sketch vector and put it in the first partition. So each partition would consist of N mini-vectors of size 2 each.).
2. Each site computes a grid and puts time series identifiers in grid cell (Table 5.3). So, for each site s,
   (a) for each time series t, place the identifier of t in a grid cell corresponding to sketch(t)[I_s], where I_s are the indexes assigned to site s.
   (b) Next form a partition of the time series identifiers such that each member of

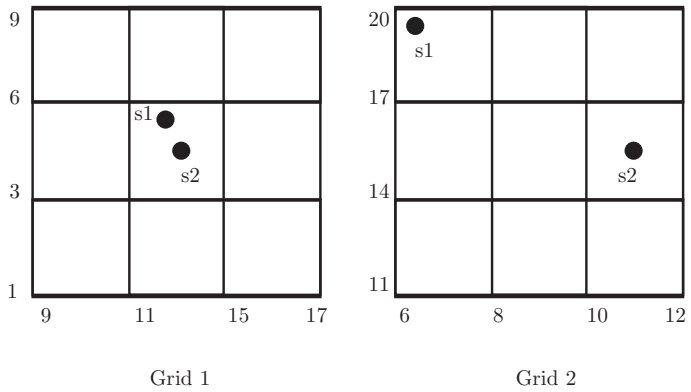


Figure 5.2 – Two series (s_1 and s_2) may be similar in some dimensions (here, illustrated by $Grid_1$) and dissimilar in other dimensions ($Grid_2$). If the series are close in a large fraction of the grids, they are likely to be similar. So, if that fraction exceeds some threshold f (2/3 in the toy example), then the algorithm checks for explicit correlation.

the partition corresponds to a non-empty grid cell. So, two time series t_1 and t_2 will fall into the same partition if sketch(t_1)[I_s] maps to the same grid cell as sketch(t2)[I_s]. Denote the partitioning induced by this grid search on site s as partitioning(s).

(c) Each element of the partition p in partitioning(s) represents a set of time series. If we sort them by their id, then p can represent $ts\_p\_1, ts\_p\_2, \ldots$.

3. We estimate that two time series are close if they are in the same grid cells in a fraction f of the grids. (The parameter f is determined by a calibration step that in turn depends on the desired correlation threshold, as we will explain in the experimental section.) We start by constructing "candidate clusters of time series" based on each grid.

4. Send each candidate cluster of time series identifier to every node corresponding to the time series in that cluster (Table 5.4). Call the mapping function between time series ids and nodes ts_to_node, to be defined as the "opt" strategy in the next subsection (For this discussion, we assume that ts_to_node is 1 to 1. If not, then if a node has say the time series groups corresponding to i_1, i_2, and i_3, then keep those groups separate.)

5. At each destination node, two time series are candidates for explicit analysis if they are in the same grid cell for some fraction f of the grids (Table 5.4). If so, compute the Pearson correlation on those two time series.

5.3.4 Communication strategies for detecting correlated candidates

Step 4 of the above algorithm requires the communication of information about each pair (t_i, tj) to one node of the system where its *grid score* (i.e., the number of grids in which the two time series are in the same cell) is computed. This communication may be done using different strategies, which in turn can have a large impact on the performance of our approach. This should come as no surprise: parallel approaches often require an optimization of communication. We compare three strategies for communicating the pairs of each grid cell:

- **All pairs communication (basic)**: In this strategy, for each cell c that contains $|count(c)|$ time series, all pairs (t_i, t_j) are generated and sent to a reducer using the pair as key (in the pair, we assume $i < j$). This ensures that all information about a pair will be sent to one reducer where its grid score can be compared with threshold f. This is the straightforward approach and will be denoted as "basic" in the rest of this chapter.
- **All time series to each responsible reducer (semi-opt)**: In this strategy, for each time series t there is a reducer r_t that is responsible for detecting the candidate time series that are correlated to t. Given a grid cell c, for each time series $t \in contents(c)$, all time series of c are sent to r_t. If, among the time series that r_t receives, the number of occurrences of a time series t' is more than the threshold f, then the pair (t, t') is considered as a candidate pair. This is the semi-optimized (semi-opt in the rest of this chapter) strategy.
- **Part of time series to each responsible reducer (opt)**: In this strategy (embodied in step 4 of the algorithm of the previous sub-section), as in the previous strategy, for each time series t there is a reducer r_t that is responsible for detecting

the time series that are potentially correlated to t. But here, only some of time series of the cell are sent to r_t. Let's assume a total order on the ids of the time series, say $t_1 < t_2 < \ldots < t_n$. Given a grid cell $contents(c) = \{t_1, \ldots, t_s\}$, for each time series $t \in contents(c)$, the time series with ids higher than that of t are sent to r_t. The idea behind this strategy is that for a potential candidate pair (t_i, t_j), we need only to count its occurrences in the one with the lower identifier (i) of the time series, not in both of them. As explained in the following analysis, this strategy requires the least amount of communication and is denoted "opt" in the rest of this chapter.

Below, we analyze the communication cost of the three strategies in terms of the size and the number of messages to be communicated for each cell. In the "basic" strategy, for the contents of each cell $contents(c) = \{t_1, \ldots, t_s\}$, all pairs (t_i, t_j) are generated and sent to the reducers. Thus, the number of messages for the cell c is equal to $count(c) \times (count(c) - 1)/2$. The size of each message is 2, so the size of data transferred for cell c is $count(c) \times (count(c) - 1)$. Note that in a distributed system, the number of messages is the principal factor for measuring communication cost of the algorithms. Thus, this approach does not have a good communication cost, as the number of messages for a cell c is $O(count(c)^2)$.

In the "semi-opt" strategy, for each cell c, the node containing each grid communicates $(count(c) - 1)$ time series to the node that must compute the grid score. This means that the number of sent messages is $count(c)$, and the total size of the communicated data is $count(c) \times count(c) - 1$ time series ids per grid cell. In this strategy the number of messages is $O(count(c))$ which is much better than the basic strategy.

In the last strategy, i.e., "opt", for each cell $contents(c) = \{t_1, \ldots, t_s\}$, we communicate $count(c)$ messages per grid cell, *i.e.*, one message to each node that is responsible for a time series in $contents(c)$. The size of the message depends on the id of the time series. Let $t_1, \ldots, t_s$ be the order of the time series ids. Then, we send $\{t_2, \ldots, t_s\}$ to r_{t_1}, $\{t_3, \ldots, t_s\}$ to r_{t_2}, etc. Therefore the total size of communicated data for cell c is $(count(c) - 1) + (count(c) - 2) + \ldots + 1 = count(c) \times (count(c) - 1)/2$. This strategy sends the same number of messages as "semi-opt" (*i.e.*, $O(count(c))$) for each cell, but the size of communicated data is smaller. Our experiments illustrate the benefits of this reduction in size.

5.3.5 Complexity analysis of parallel mixing

Let us analyze the time and space needed by our approach to perform parallel mixing.

The redistribution in Step 1 is proportional to the number of time series times the number of random vectors (because the number of random vectors equals the size of each sketch vector). Note that it is independent of the size of the window. This step is very well parallelizable at the level of nodes and linear in the number of time series. Step 2 (inserting into grids) is linear in the number of time series, cheaper than Step 1.

The dominant time of our approach is that of Steps 3 and 4 in which the responsible node of each grid constructs the candidate clusters of time series, and sends them to the corresponding node based on ts_to_node. If ts_to_node is many to one, then even in the worst case the number of messages is proportional to the number of destination nodes and the total message traffic from a node is proportional to the square of the number of time series $\times$ the size of each time series identifier. That is a very pessimistic worst case because it corresponds to all time series mapping to the same grid cell in every grid. As we will see in the experimental section, the total traffic per node is linear in the number of time series in practice. Because time series ids are under 32 bits, the total traffic is light.

The last step, i.e. 5, is proportional to the size of the output, because a large fraction of pairs that pass the sketch filtering step in fact meet the correlation threshold.

The bulk of the space required for our approach is the space needed for keeping the grids for indexing the sliding windows. This space depends on the number of grids and the number of time series. The number of grids itself depends on the size of the sketches in the sliding window, and the group size (number of dimensions in each grid). Let g be the group size, s be the sketch vector size, and n the number of time series. Then, the number of grids required for indexing the sliding window data is $\frac{s}{g}$. In each grid, we need to keep the id of each time series in its corresponding cell. Thus, the total space required for storing the grids is $O(n \times idsize \times \frac{s}{g})$, where n is the number of time series, $idsize$ is the size of an id, s is the sketch vector size and g the group size. Notice that in practice, the size of our grid-based index is much less than the space required for keeping

the time series in the sliding windows. For example, suppose the group size is 2, and the sketch vector size is 32 (for a sliding window of size 256). Then, the space required to store all grids is equivalent to $16 \times n$ identifiers, which is less than the space needed to store n sliding windows of size 256.

So, in practice, the entire procedure requires work that is the sum of i (formation of sketch vectors): number of time series × size of basic windows × number of random vectors, ii (parallel mixing, grid computation): number of time series × number of random vectors, iii (parallel mixing, candidate identification) for each grid cell, square of the number of time series × size of time series identifiers, iv (for verification of candidate pairs): number of highly correlated pairs × window size, This work, except for the communication step (which depends on the communications infrastructure), is entirely parallelizable. Which term dominates depends on how high the threshold is. For very high thresholds, part iv will be negligible, iii will be small, and so i and ii will dominate. If the threshold is low (not normally an interesting case) the algorithm could be nearly as expensive as comparing every pair of time series.

5.4 Experiments

In this section, we report experimental results that show the quality and the performance of our parallel incremental sketching approach, illustrating performance, scalability, recall, and precision. We compare our work with iSAX and show vastly improved speed at some cost in recall.

The parallel experimental evaluation was conducted on a cluster of 32 machines, with operating system Linux x86_64 kernel 3.10.0, each machine having 64 Gigabytes of main memory, an Intel Xeon CPU with 8 cores and a 256 Gigabytes hard disk.

We implemented the approaches on top of Apache-Spark 1.6.2 [78], using the Java programming language.

Data streams are simulated by distributing the data beforehand and using synchronized sliding windows on each site. This setup allowed us to better evaluate the perfor-

mance gains of our approach without depending on the specific characteristics or optimization of any dedicated streaming environment (e.g. Spark streaming, Flink, Storm, etc.).

5.4.1 Comparisons

We compare ParrCorr to:

- **Parallel Linear Search**. This is the straightforward comparison that compares each time series to all the other ones, computing a Pearson correlation. Correlations are sorted by decreasing order and the top-correlated ones are kept. It is implemented in parallel (each computing node compares the series it contains to all the series of the other nodes).
- **iSax** [12]. This index allows processing similarity queries using both an exact and an approximate approach. iSax shows an improvement over Parallel Linear Search: when a computing node receives a time series to be compared to its local time series, rather than applying a linear search it will use a local iSax index as a filter to identify the most similar time series.

In the data stream context, these algorithms are applied from scratch, after each update (each basic window-sized move of the sliding window). For iSax, the local indexes have to be built again after each update.

5.4.2 Datasets

We carried out our experiments on both synthetic and seismic datasets.

Synthetic dataset: Each time series in our synthetic dataset consists of 2000 values. At each time point, the generator draws a random number from a Gaussian distribution N(0,1), then adds the value of the last number to the new number. The number of time

series varies from one million to 100 million depending on the experiment. This type of random walk generator has been widely used in the past. [3, 23, 6, 65, 11, 12, 80].

Seismic dataset: The real world data represents seismic time series collected from the IRIS Seismic Data Access repository [30] at various earthquake zones. After preprocessing, the seismic dataset contains 5 million time series of 2000 values each.

To detect seismic events, there are three main types of algorithms: energy detectors, array detectors and matched filter detectors. The latter is a new kind of detector, where a representative time series is used as a template (i.e., a "matched filter") and correlated against a continuous data stream to detect new occurrences of that same signal. However, such filters require a large number of templates, making indexing an appealing approach. A time series at a given sensor functions like a geophysical fingerprint for earthquakes. A seismic signal that closely matches a previous observation can be used as evidence that the newly observed event must have occurred very close to the event that generated the first observation. Moreover, if the signals are similar we can assume that the characteristics of the earthquakes are similar. There are many examples where almost identical signals produced by different earthquakes have been observed. This is typically the case during seismic crises that can last days or months, while similar signals can be recorded even if years apart. Detecting such correlations is a small variation of our problem, where all time series are compared with a few templates. Here we address the harder problem of finding all correlations among the set of time series. This might be useful in an application in which we want to detect, in a real time fashion, where similar seismic events are occurring.

5.4.3 Parameters

Table 5.5 shows the default parameters used for each experiment, unless otherwise specified. A typical application might have a large ratio between the sliding window size and the basic window size, where the basic window indicates the time interval between the recalculation of similarity. We've chosen a ratio of 50, which we have found to be reasonable for many applications. ParCorr does better relative to the other algorithms with a smaller basic window size of 10 for example, but 50 is more reasonable for high

Table 5.5 – Default parameters

Parameters	Value
Sliding Window Size	500
Basic Window Size	20
iSAX Word Length	8
Leaf Capacity Threshold	1,000
Basic Cardinality	2
Maximum Cardinality	512
Number Of Machines	32
Correlation Threshold	0.7

frequency measurements. The iSAX word length, leaf capacity, basic cardinality, and maximum cardinality were chosen to be optimal for iSAX (and were taken from [12]). All histograms in the figures have error bars (usually so small as to be invisible) that go from a minimum value to a maximum value (i.e. 100% confidence interval) with the histogram height representing the mean.

We calibrate the fraction f (needed for detecting candidate items in the grids) by using a small sample database. We increase f until reaching the desired recall (*e.g.*, 0.95) on the small sample, and then we use the found fraction in our experiments on big datasets.

5.4.4 Recall and Precision Measures

To understand these concepts in our applications, consider the correlation problem: we want to find all pairs of time series that have at least a correlation of some specified threshold during a given window. Call that set S_{true} In that context, the *recall* of a method that finds a set S_{method} is $|(S_{true} \cap S_{method})|/|S_{true}|$ and the *precision* is $|(S_{true} \cap S_{method})|/|S_{method}|$. This would also be true for Euclidean distances. These are completely standard uses of these terms applied to pairs and similarity metrics.

In our experiments, the default correlation threshold for Pearson is 0.7. We have also tried 0.8 and 0.9. With a Pearson threshold of 0.8, the sketch recall was over 96% and the speedup compared with iSAX was a factor of 17.56. With a Pearson threshold of 0.9,

the sketch recall was over 95.7% and the speedup compared with iSAX was a factor of 18. Given any Pearson correlation, the threshold for Euclidean distance is computed by using formula 5.1.

5.4.5 Communication Strategies

Before presenting the results of our approach in detail, we evaluate here the impact of the communication strategy to detect correlated pairs. This corresponds to the discussion and analysis given in Section 5.3.4. We conducted this experiment on 5 million time series, with a basic window of 32 and a sliding window of 256. As expected and illustrated by Figure 5.3, our optimized strategy gives the best performance (response time), but the size of the gain is surprising. Therefore, in the experiments presented below, we use this optimized strategy.

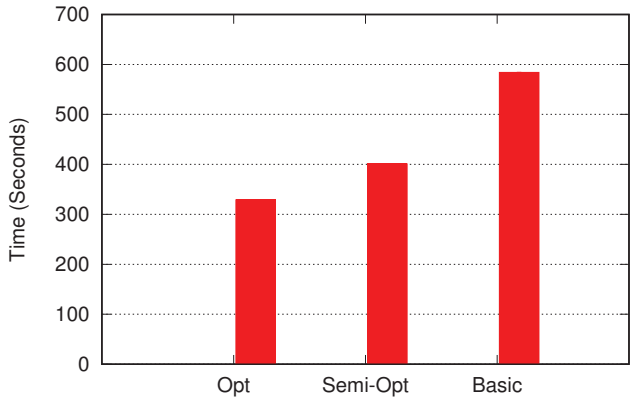


Figure 5.3 – Execution time (not including the pair checking time) for each of the communication strategies introduced in Section 5.3.4. The algorithms are run on a cluster of 32 nodes and 5 million time series (basic window of 32 and sliding window of 256). The optimized strategy gives the best response time.

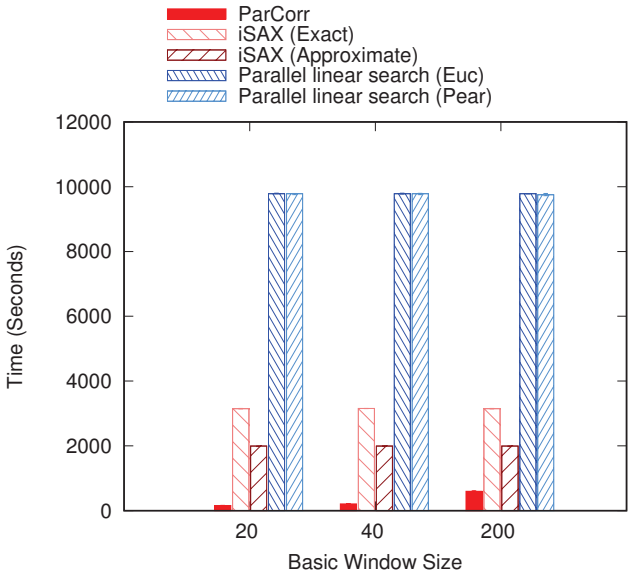


Figure 5.4 – Execution time for the calculation of the correlations for each sliding window as a function of basic window size for the random walk dataset. The algorithms are run on a cluster of 32 nodes and 5 million time series. The time for ParCorr increases as the basic window size increases, because updating the sketch vector takes slightly longer. All parameters other than basic window size are set to their values from Table 5.5.

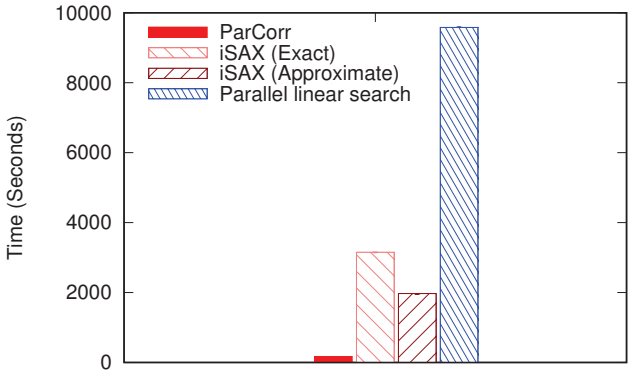


Figure 5.5 – Execution time for the calculation of the correlations for each sliding window for the seismic dataset. The algorithms are run on a cluster of 32 nodes and 5 million time series. All parameters are set to their values from Table 5.5.

5.4.6 Results

Figure 5.4 shows that ParCorr is orders of magnitude faster for parallel correlation than the iSAX methods for the random walk dataset, though its time increases as the basic window size increases. For instance, with a basic window of 20, ParCorr takes at most 160 seconds to process a sliding window, while iSAX Approximate needs 1990 seconds. We attribute this advantage to two factors: the calculation of sketches is incremental and the parallelization of the algorithm is natural. These results also hold for the seismic data as can be seen in Figure 5.5.

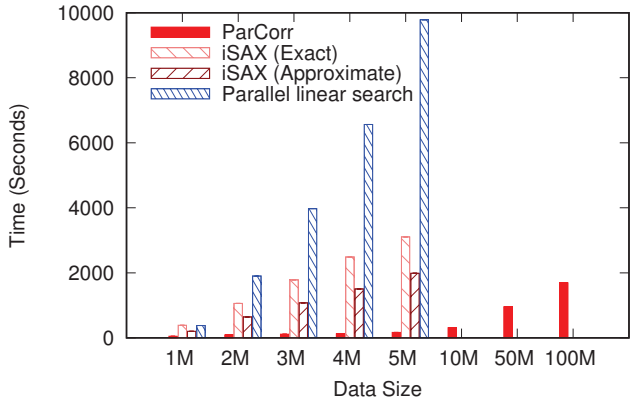


Figure 5.6 – Execution time for the calculation of the correlations for each sliding window as a function of dataset size for the random walk dataset. The algorithms are run on a cluster of 32 nodes. All parameters are set to their values from Table 5.5. Note that ParCorr scales to larger datasets nearly linearly and the times remain practical. The other methods exceeded the measurement window.

Figure 5.6 shows that ParCorr scales well to large datasets containing up to 100 million time series. iSAX approximate is consistently about 50% faster than ISAX exact. Our competitors (Parallel linear search, iSAX Approximate/Exact) do not scale since they cannot handle more than 5 million time series due to the fact that both memory usage and communication costs become hard to bear.

Figure 5.7 shows that both iSAX and ParCorr enjoy a roughly linear speedup, whereas Figure 5.8 shows that ParrCorr is orders of magnitude faster in absolute time at all de-

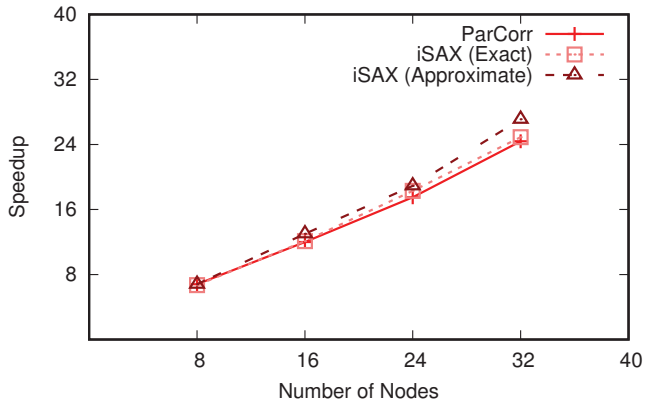


Figure 5.7 – SpeedUp: All algorithms enjoy linear speedup with roughly the same slope as the number of processing nodes increase. All parameters are set to their values from Table 5.5.

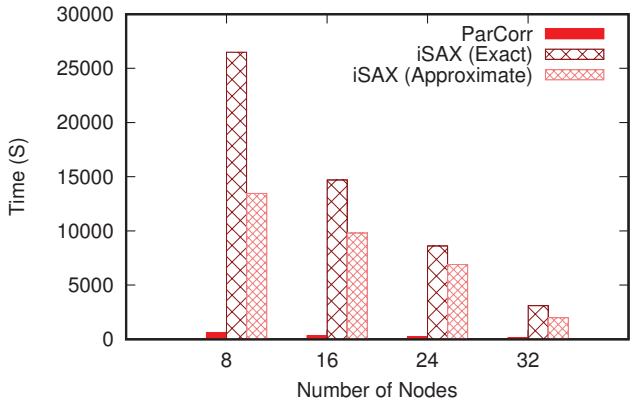


Figure 5.8 – Execution time for the calculation of the correlations for each sliding window as a function of the number of processing nodes for the random walk dataset. The algorithms are run on 5 million time series. All parameters are set to their values from Table 5.5.

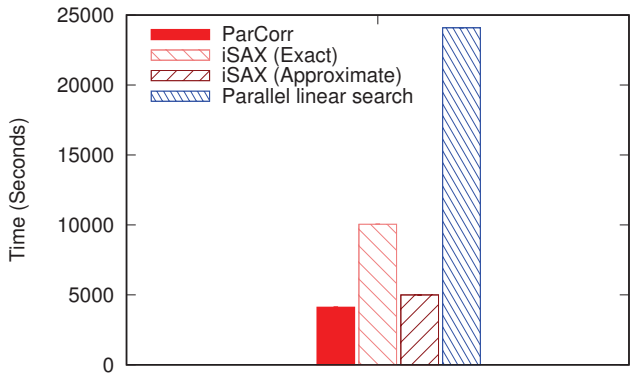


Figure 5.9 – Execution time for each sliding window on a single node for the random walk dataset. The dataset is 1 million time series. All parameters are set to their values from Table 5.5.

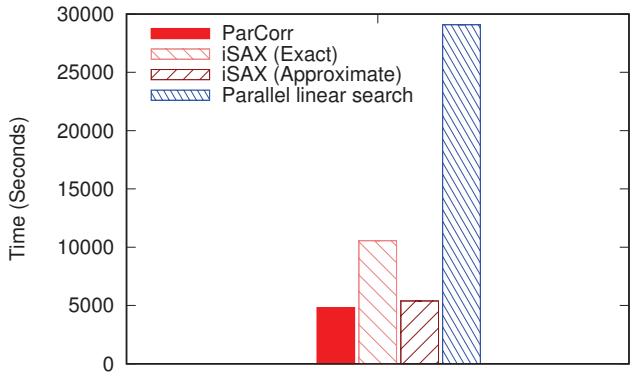


Figure 5.10 – Execution time for each sliding window on a single node for the seismic dataset. The dataset is 1 million time series. All parameters are set to their values from Table 5.5.

grees of parallelization. ParCorr needs at most 598 seconds on 8 nodes (169 seconds on 32 nodes) while iSAX Approximate needs at most 13460 seconds (9784 seconds on 32 nodes).

Figure 5.9 shows that ParCorr's performance (using Spark) is comparable to iSAX (running natively without Spark) on a single node. ParCorr shows a small advantage but not as much as in a parallel setting, because Spark entails some overhead. These results are consistent with the results for the seismic data as shown in Figure 5.10.

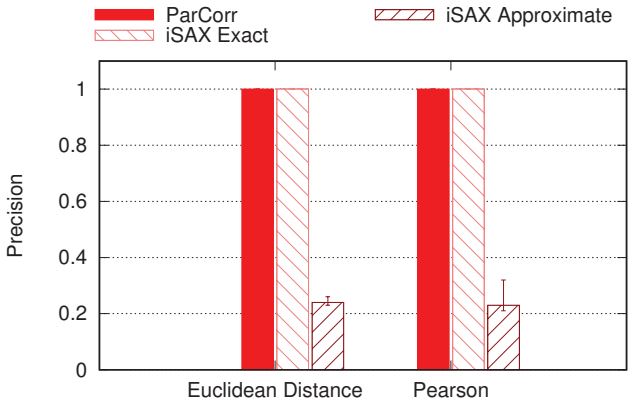


Figure 5.11 – Let Peuc be the set of pairs of time series whose final w values fall within the distance threshold in the case of Euclidean distance. And let Pcorr be the set of pairs of time series whose final w values fall above the threshold in the case of Pearson. The precision is the fraction of the set of pairs found by each algorithm that belong to Peuc or Pcorr, respectively. ParCorr has 100% precision because it checks candidate pairs that are produced by the sketch algorithm.

Figure 5.11 shows the high precision of ParCorr and iSAX. ParCorr verifies all the candidate pairs that the sketch filter produces. iSAX Exact incorporates a verification step as well. These results hold also for seismic data as seen in Figure 5.13.

Figure 5.12 shows that iSAX Exact gives perfect recall because of its bounding box guarantee. ParCorr gives no such guarantee, so for applications that require 100% recall, iSAX Exact should be used. Empirically, ParCorr yields a recall of over 90%, as shown in Figure 5.14.

The experiments on real and synthetic data show that ParCorr is fast, scales well,

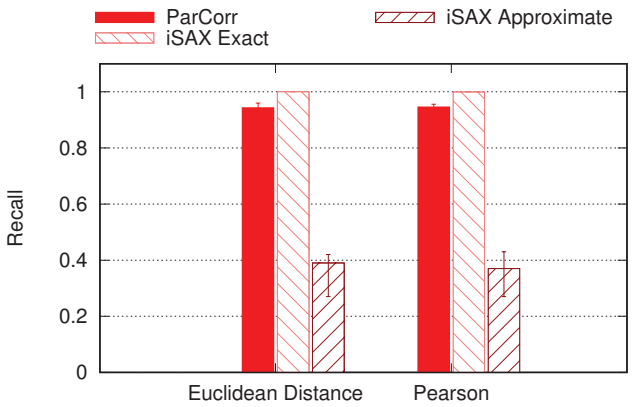


Figure 5.12 – Let Peuc and Pcorr be defined as in the caption of Figure 5.11. The recall is the fraction of Peuc or Pcorr, respectively, that is found by each algorithm. Note that iSAX exact gives higher recall than ParCorr.

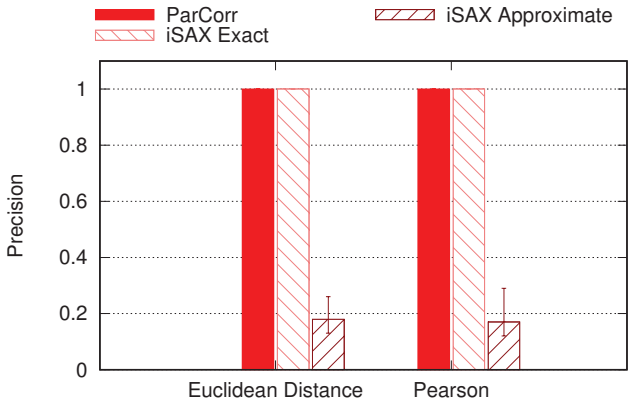


Figure 5.13 – For seismic precision, iSAX Exact and ParCorr both achieve 100% precision for Euclidean. ParCorr also achieves 100% precision for Pearson correlation.

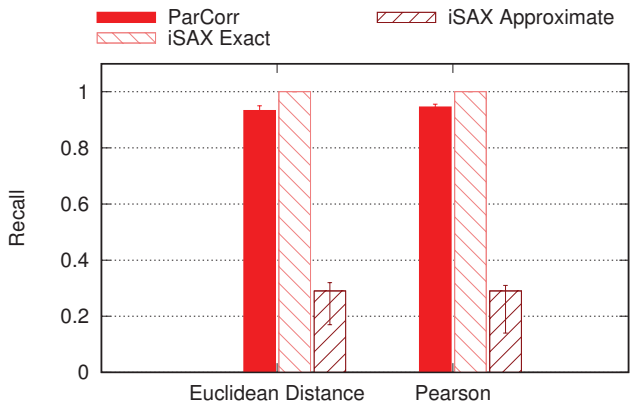


Figure 5.14 – For seismic data, iSAX Exact achieves perfect recall. ParCorr achieves over 90% for both Euclidean and Pearson correlation.

guarantees 100% precision, and achieves very high recall. This reduction in recall is acceptable for many applications, especially given the high gain in response time.

5.5 Conclusion

Finding similar pairs of time series on sliding windows is useful for many applications. Methods to do so for hundreds of millions of time series in a highly efficient and scalable fashion is the contribution of this work. Compared with the previous state of the art iSAX, our solution is faster and scalable while showing only very little loss in recall. For many applications, where scalability is mandatory, this is highly beneficial.

Chapter 6

Conclusions

This thesis is done in the context of parallel mining of time series in massively distributed environments. We have focused on the problem of time series indexing in big data, aiming to improve and accelerate similarity query processing which is of high interest for various applications that deal with big data sets. In this chapter, we summarize and discuss our main contributions and we give some research directions for future work.

6.1 Contributions

This thesis includes the following main contributions related to time series indexing in massively distributed environments.

6.1.1 Massively Distributed Time Series Indexing and Querying with DPiSAX

In this contribution, our main challenge was the parallelization of the index creation and the processing of queries in parallel using that index. The index, in this part of the thesis, builds on the principles of iSAX. We proposed DPiSAX (Distributed Partitioned

iSAX), our solution for large-scale indexing of time series using the iSAX representation. By means of our distributed approach, indexing scales up to billions of time series. Our proposal includes algorithms to query the index and to support exact and approximate k-nearest neighbor queries. Our approach has been extensively evaluated with both synthetic and real-world datasets. The high scalability of our solution DPiSAX comparing to other alternatives confirms its effectiveness.

6.1.2 RadiusSketch: Parallel Indexing of Time Series

In this part of the thesis, we addressed the problem of indexing and retrieving time series by means of sketches representations. Our main challenge was the parallelization of the sketches and answering the queries in parallel using these structures. We have proposed a random projection-based approach that scales nearly linearly in parallel environments and provides high-quality answers. Our proposal includes a parallel index construction algorithm that takes advantage of distributed environments to efficiently build sketch-based indices over very large volumes of time series. We also proposed a parallel query processing algorithm for approximate k-nearest neighbor queries. We have extensively evaluated our approach using both synthetic and real-world datasets. The sketch method, as we have implemented, is superior in both quality and response time compared to the centralized state of the art approach.

6.1.3 ParCorr: Efficient Parallel Methods to Identify Similar Time Series Pairs Across Sliding Windows

Here, we addressed the problem of all-pair parallel correlation detection across sliding windows of time series in a data stream. The goal is to have an up-to-date result, at any point in time, despite the demanding constraints of data streams. We have proposed an efficient parallel solution for detecting similar time series across sliding windows. Our proposal uses the sketch principle for representing the time series. It includes a parallel approach for incremental computation of the sketches in sliding windows, and a partitioning approach that projects sketch vectors of time series into subvectors and

builds a distributed grid structure. Our approach has been evaluated using real and synthetics datasets. The results confirm the efficiency and almost linear scalability of the proposed solution compared to the state of the art.

6.2 Directions for Future Work

The results presented in this thesis leave room to further improvement. For instance, We may use Spark Streaming for finding the highly correlated pairs of time series over a time window. The main goal would be to make the system more scalable since Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. This would require implementing ParrCorr using DStreams and windowed DStream. Another possible future work concerns a joint study of DPiSAX and RadiusSketch, in order to reach an efficient approach that would benefit from both principles. This is an ongoing work that has already started and builds on top of the results of this thesis. We give some preliminary results in the following, with a first synthetic study of RadiusSketch and DPiSAX.

We started to compare the performance of two versions of our solution: 1) DPiSAX that is the parallel implementation of iSax with the statistical partitioning described in 3; 2) RadiusSketch is the parallel sketch indexing approach with partitioning described in 4. To that end, we measured the index construction times with different parameters (e.g., dataset size, cluster size, communication cost), and the query performance of the two approaches, both in terms of response time and accuracy.

The experimental evaluation was conducted on the Nef platform, using a cluster of 16 machines. We carried out our experiments on the Random Walk dataset of 1 Billion time series, where the length of each time series was 256. Table 6.1 shows the default parameters (unless otherwise specified in the text) used for each approach.

	Parameters
RadiusSketch	Group size = 2 Sketch size = 60
DPiSAX	iSAX word length = 8 Leaf capacity = 1,000 Basic cardinality = 2 Maximum cardinality = 512 Sampling fraction = 10%

Table 6.1 – Default parameters

First, we measured the index construction time of DPiSAX and RadiusSketch. Figure 3.3 reports the index construction times for all approaches on our Random Walk dataset. We observe that the index construction time of RadiusSketch is slightly better then DPiSAX, but they are generally not significantly different in our experiments.

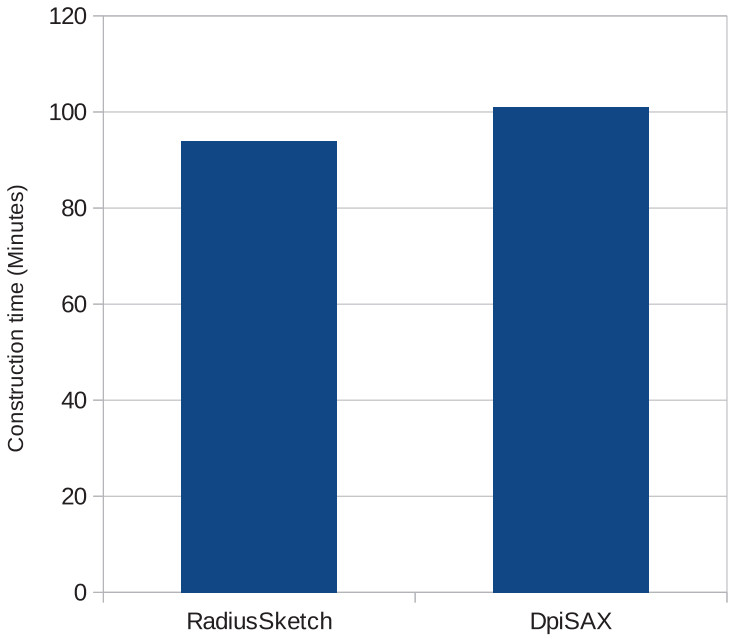


Figure 6.1 – Construction time of DPiSAX and RadiusSketch for 1 billion Random Walk time series

Second, we evaluated the querying performance of DPiSAX, and compared them to

RadiusSketch. Given a query q, let T be the set of the true candidates and C the set of the given candidates. In our experiments, we measured two metrics:

- **Recall:** $|T \cap C|/|T|$
- **Precision:** $|T \cap C|/|C|$

Note that in our experiments, we have $|T| = |C|$, thus recall is equal to precision. This is why, we only report the results of precision.

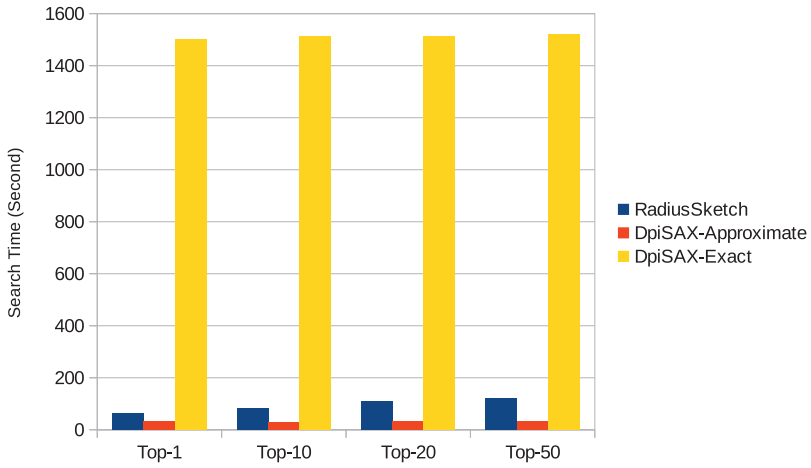


Figure 6.2 – Search time for 1000 queries as a function of k nearest neighbors of DPiSAX-exact, DPiSAX-Approximate, RadiusSketch

Figures 6.2 and 6.3 compares the search time of approximate and exact k nearest neighbors queries for the parallel approaches. Figure 6.2 reports the response time of DPiSAX with exact search, which is much longer. For readability, we report in Figure 6.3 the response times of approximate approaches only. We can observe that the response time increases with the number k nearest neighbors for RadiusSketch. However, for DPiSAX the search time of approximate k nearest neighbors is nearly constant and lower than RadiusSketch and exact k nearest neighbors.

Figures 6.4 illustrate the precision of different tested approaches, with varying values of k, the number of nearest neighbors to be found. We observe that the recall of DPiSAX-exact is the best (as expected). We also observe that RadiusSketch offers better

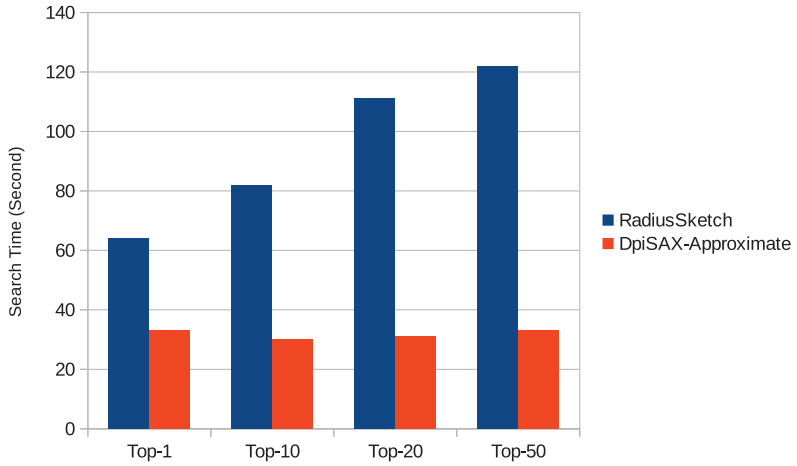


Figure 6.3 – Search time for 1000 queries as a function of k nearest neighbors of DPiSAX-Approximate and RadiusSketch

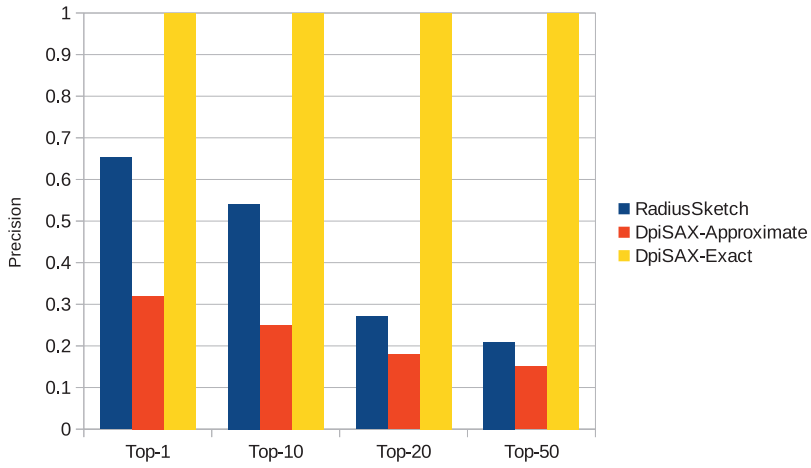


Figure 6.4 – Precision of RadiusSketch and DPiSAX (average value for 1000 queries).

performances than DPiSAX-Approximate. This is the trade-off that has to be studied in detail. We want to find new principles in each of these approaches, getting inspired from the other, in order to improve and obtain better response times and/or better precision. RadiusSketch, for instance, may be fine tuned for faster execution by reducing the length of sketches. However, that has a negative impact on precision. This trade-off between response time and precision is classic for a large number of problems in com-

puter science in general, and particularly for data analytics. It is at the core of this future work.

Bibliography

[1] La gilberta. http://lagilberta.pi.ingv.it/seismo/en/new-earthquake-in-the-chianti-area/.

[2] Achlioptas, D. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci. 66*, 4 (2003), 671–687.

[3] Agrawal, R., Faloutsos, C., and Swami, A. N. Efficient similarity search in sequence databases. In *Proceedings of International Conference on Foundations of Data Organization and Algorithms (FODO)* (1993), Springer-Verlag, pp. 69–84.

[4] An, J., Chen, H., Furuse, K., Ohbo, N., and Keogh, E. *Grid-Based Indexing for Large Time Series Databases.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[5] Ann, C., and Keogh, R. E. Everything you know about dynamic time warping is wrong.

[6] Assent, I., Krieger, R., Afschari, F., and Seidl, T. The ts-tree: efficient time series search and retrieval. In *Proceedings of International Conference on Extending Database Technology (EDBT)* (2008), pp. 252–263.

[7] Balzanella, A., Adelfio, G., Chiodi, M., D'Alessandro, A., and Luzio, D. *Time-Frequency Filtering for Seismic Waves Clustering.* Springer International Publishing, Cham, 2014, pp. 1–9.

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

[9] Bizer, C., Boncz, P. A., Brodie, M. L., and Erling, O. The meaningful use of big data: four perspectives - four challenges. *SIGMOD Record 40*, 4 (2011), 56–60.

[10] Cai, Y., and Ng, R. Indexing spatio-temporal trajectories with chebyshev polynomials. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2004), SIGMOD '04, pp. 599–610.

[11] Camerra, A., Palpanas, T., Shieh, J., and Keogh, E. isax 2.0: Indexing and mining one billion time series. In *ICDM Conf.* (2010), pp. 58–67.

[12] Camerra, A., Shieh, J., Palpanas, T., Rakthanmanon, T., and Keogh, E. J. Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. *Knowl. Inf. Syst.* (2014).

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

[14] Charikar, M. S. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing* (2002), STOC '02, pp. 380–388.

[15] Chen, L., and Ng, R. On the marriage of lp-norms and edit distance. In *VLDB Conf.* (2004), vol. 30, pp. 792–803.

[16] Chen, Q., Chen, L., Lian, X., Liu, Y., and Yu, J. X. Indexable pla for efficient similarity search. In *VLDB Conf.* (2007), pp. 435–446.

[17] Cole, R., Shasha, D., and Zhao, X. Fast window correlations over uncooperative time series. In *KDD Conf.* (2005), pp. 743–749.

[18] Cormode, G., Indyk, P., Koudas, N., and Muthukrishnan, S. Fast mining of massive tabular data via approximate distance computations. In *Proceedings of International Conference on Data Engineering (ICDE)* (2002), pp. 605–614.

[19] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[20] Ding, R., Wang, Q., Dang, Y., Fu, Q., Zhang, H., and Zhang, D. YADING: fast clustering of large-scale time series data. *PVLDB 8*, 5 (2015), 473–484.

[21] Donald, J. B., and James, C. Using dynamic time warping to find patterns in time series. In *Knowledge Discovery in Databases* (1994), pp. 359–370.

[22] Esling, P., and Agon, C. Time-series data mining. *ACM Comput. Surv. 45*, 1 (Dec. 2012), 12:1–12:34.

[23] Faloutsos, C., Ranganathan, M., and Manolopoulos, Y. Fast subsequence matching in time-series databases. *In Proceedings of the International Conference on Management of Data (SIGMOD) 23*, 2 (1994), 419–429.

[24] Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., and Strauss, M. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)* (2001), pp. 79–88.

[25] Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proceedings of International Conference on Very Large Data Bases (VLDB)* (1999), pp. 518–529.

[26] Hand, D. J., Smyth, P., and Mannila, H. *Principles of Data Mining*. MIT Press, Cambridge, MA, USA, 2001.

[27] Huijse, P., Estévez, P. A., Protopapas, P., Principe, J. C., and Zegers, P. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Comp. Int. Mag. 9*, 3 (2014), 27–39.

[28] Indyk, P. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *41st Annual Symposium on Foundations of Computer Science (FOCS)* (2000), pp. 189–197.

[29] Indyk, P., Koudas, N., and Muthukrishnan, S. Identifying representative trends in massive time series data sets using sketches. In *Proceedings of International Conference on Very Large Data Bases (VLDB)* (2000), pp. 363–372.

[30] IRIS. Seismic data access. http://ds.iris.edu/data/access/.

[31] Jégou, H., Tavenard, R., Douze, M., and Amsaleg, L. Searching in one billion vectors: re-rank with source coding. In *ICASSP* (2011).

[32] Jeong, Y.-S., Jeong, M. K., and Omitaomu, O. A. Weighted dynamic time warping for time series classification. *Pattern Recogn. 44*, 9 (Sept. 2011), 2231–2240.

[33] Johnson, W. B., and Lindenstrauss, J. Extensions of Lipschitz mapping into Hilbert space. In *Conf. in modern analysis and probability* (1984), vol. 26 of *Contemporary Mathematics*, pp. 189–206.

[34] Kashino, K., Smith, G., and Murase, H. Time-series active search for quick retrieval of audio and video. In *ICASSP* (1999).

[35] Kenney, J., and Keeping, E. *Mathematics of Statistics*. van Nostrand, 1963.

[36] Keogh, E., Chakrabarti, K., Pazzani, M., and Mehrotra, S. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems 3* (2000), 263–286.

[37] Keogh, E., and Kasetty, S. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov. 7*, 4 (Oct. 2003), 349–371.

[38] Keogh, E., and Lin, J. Clustering of time-series subsequences is meaningless: implications for previous and future research. *Knowledge and Information Systems 8*, 2 (Aug 2005), 154–177.

[39] Keogh, E. J. Exact indexing of dynamic time warping. In *VLDB* (2002).

[40] Khiali, L., Ienco, D., and Teisseire, M. Object-oriented satellite image time series analysis using a graph-based representation. *Ecological Informatics* (2017).

[41] Kin-Pong, C., and Ada, W. F. Efficient time series matching by wavelets. In *ICDE Conf.* (1999), pp. 126–133.

[42] Kushilevitz, E., Ostrovsky, R., and Rabani, Y. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC)* (1998), pp. 614–623.

[43] L., P., C., G., Ooi, B. C., V., H. T., and W., S. Scalagist: Scalable generalized search trees for mapreduce systems. *PVLDB 7*, 14 (2014), 1797–1808.

[44] Lin, J., Keogh, E., Lonardi, S., and Chiu, B. A symbolic representation of time series, with implications for streaming algorithms. In *SIGMOD* (2003).

[45] Lin, J., Keogh, E., Lonardi, S., Lankford, J. P., and Nystrom, D. M. Visually mining and monitoring massive time series. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2004), KDD '04, ACM, pp. 460–469.

[46] Lin, J., Keogh, E., Lonardi, S., and Patel, P. Finding motifs in time series. pp. 53–68.

[47] Lin, J., Keogh, E., Wei, L., and Lonardi, S. Experiencing sax: A novel symbolic representation of time series. *Data Min. Knowl. Discov.* (2007).

[48] Matsubara, Y., and Sakurai, Y. Regime shifts in streams: Real-time forecasting of co-evolving time sequences. In *Proceedings of International Conference on Knowledge Discovery and Data Mining (SIGKDD)* (2016), pp. 1045–1054.

[49] Mueen, A., Nath, S., and Liu, J. Fast approximate correlation for massive time-series data. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2010), pp. 171–182.

[50] Palpanas, T. Data series management: The road to big sequence analytics. *SIGMOD Record 44*, 2 (2015), 47–52.

[51] Palpanas, T. Big sequence management: A glimpse of the past, the present, and the future. In *SOFSEM* (2016).

[52] Papadimitriou, S., Sun, J., and Faloutsos, C. Streaming pattern discovery in multiple time-series. In *Proceedings of International Conference on Very Large Data Bases (VLDB)* (2005), pp. 697–708.

[53] Papadimitriou, S., and Yu, P. S. Optimal multi-scale patterns in time series streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2006), pp. 647–658.

[54] Perng, C., Wang, H., and Ma, S. Fast relevance discovery in time series. In *Proceedings of the International Conference on Data Mining (ICDM)* (2006), pp. 1016–1020.

[55] Rafiei, D., and Mendelzon, A. O. Efficient retrieval of similar time sequences using DFT. *CoRR cs.DB/9809033* (1998).

[56] Rakthanmanon, T., Campana, B., Mueen, A., Batista, G., Westover, B., Zhu, Q., Zakaria, J., and Keogh, E. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD* (2012).

[57] Ralanamahatana, C. A., Lin, J., Gunopulos, D., Keogh, E., Vlachos, M., and Das, G. *Mining Time Series Data.* Springer US, Boston, MA, 2005, pp. 1069–1103.

[58] Raza, U., Camerra, A., Murphy, A. L., Palpanas, T., and Picco, G. P. Practical data prediction for real-world wireless sensor networks. *IEEE Trans. Knowl. Data Eng.* (accepted for publication, 2015).

[59] Reinert, G., Schbath, S., and Waterman, M. S. Probabilistic and statistical properties of words: An overview. *Journal of Computational Biology 7*, 1-2 (2000), 1–46.

[60] S. Soldi, V. Beckmann, W. e. a. G. C. P. L. H. F. M. J. T. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics* (2014).

[61] Sakurai, Y., Faloutsos, C., and Yamamuro, M. Stream monitoring under the time warping distance. In *Proceedings of International Conference on Data Engineering (ICDE)* (2007), pp. 1046–1055.

[62] Shafer, J., Rixner, S., and Cox, A. L. The hadoop distributed filesystem: Balancing portability and performance. In *Int. ISPASS* (2010).

[63] Shasha, D. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull. 22*, 2 (1999), 40–46.

[64] Shasha, D., and Zhu, Y. *High Performance Discovery in Time series, Techniques and Case Studies.* Springer, 2004.

[65] Shieh, J., and Keogh, E. isax: Indexing and mining terabyte sized time series. In *KDD Conf.* (2008), pp. 623–631.

[66] Shieh, J., and Keogh, E. isax: Disk-aware mining and indexing of massive time series datasets. *DMKD 19*, 1 (2009), 24–57.

[67] Thaper, N., Guha, S., Indyk, P., and Koudas, N. Dynamic multidimensional histograms. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2002), pp. 428–439.

[68] V., M., K., G., and G., D. Discovering similar multidimensional trajectories. In *ICDE Conf.* (2002), pp. 673–684.

[69] W., Y., W., P., P., J., W., W., and H., S. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB* (2013).

[70] Wang, X., Ye, L., Keogh, E., and Shelton, C. Annotating historical archives of images. *International Journal of Digital Library Systems 1*, 2 (2010), 59–80.

[71] White, T. *Hadoop : the definitive guide.* O'Reilly, 2012.

[72] Witten, I. H., Frank, E., and Hall, M. A. *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[73] Xie, Q., Shang, S., Yuan, B., Pang, C., and Zhang, X. Local correlation detection with linearity enhancement in streaming data. In *Proceedings of International Conference on Information & Knowledge Management (CIKM)* (2013), pp. 309–318.

[74] Yagoubi, D.-E., Akbarinia, R., Masseglia, F., and Palpanas, T. Dpisax: Massively distributed partitioned isax. In *IEEE International Conference on Data Mining (ICDM)* (2017).

[75] Yagoubi, D.-E., Akbarinia, R., Masseglia, F., and Shasha, D. Radiussketch: Massively distributed indexing of time series. In *IEEE International Conference on Data Science and Advanced Analytics (DSAA)* (2017).

[76] Ye, L., and Keogh, E. J. Time series shapelets: a new primitive for data mining. In *KDD* (2009).

[77] Yeh, C. M., Zhu, Y., Ulanova, L., Begum, N., Ding, Y., Dau, H., Silva, D., Mueen, A., and Keogh, E. Matrix profile I: all pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *ICDM, 2016*.

[78] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

[79] Zhu, Y. *High Performance Data Mining in Time Series: Techniques and Case Studies*. Phd thesis, New York University, 2004.

[80] Zoumpatianos, K., Idreos, S., and Palpanas, T. Indexing for interactive exploration of big data series. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2014), pp. 1555–1566.

[81] Zoumpatianos, K., Idreos, S., and Palpanas, T. ADS: the adaptive data series index. *VLDB J. 25*, 6 (2016), 843–866.