SNIF TOOL - Sniffing for Patterns in Continuous Streams

by

Abhishek Mukherji

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

February 2008

APPROVED:

Professor Elke A. Rundensteiner, Thesis Advisor

Professor David C. Brown, Reader

Professor Michael A. Gennert, Head of Department

ABSTRACT

Recent technological advances in sensor networks and mobile devices give rise to new challenges in processing of live streams. In particular, time-series sequence matching, namely, the similarity matching of live streams against a set of predefined pattern sequence queries, is an important technology for a broad range of domains that include monitoring the spread of hazardous waste and administering network traffic. In this thesis, I use the time critical application of monitoring of fire growth in an intelligent building as my motivating example. Various measures and algorithms have been established in the current literature for similarity of static time-series data. Matching continuous data poses the following new challenges: 1) fluctuations in stream characteristics, 2) real-time requirements of the application, 3) limited system resources, and, 4) noisy data. Thus the matching techniques proposed for static time-series are mostly not applicable for live stream matching.

In this thesis, I propose a new generic framework, henceforth referred to as the n-Snippet Indices Framework (in short, SNIF), for discovering the similarity between a live stream and pattern sequences. The framework is composed of two key phases: (1.) Off-line preprocessing phase: where the pattern sequences are processed offline and stored into an approximate 2-level index structure; and (2.) On-line live stream matching phase: streaming time-series (or the live stream) is on-the-fly matched against the indexed pattern sequences. I introduce the concept of n-Snippets for numeric data as the unit for matching. The insight is to match small snippets of the live stream against prefixes of the patterns and maintain them in succession. Longer the pattern prefixes identified to be similar to the live stream, better the confirmation of the match. Thus, the live stream matching is performed in two levels of matching: bag matching for matching snippets and order checking for maintaining the lengths of the match. I propose four variations of matching algorithms that allow the user the capability to choose between the two conflicting characteristics of result accuracy versus response time.

The effectiveness of SNIF to detect patterns has been thoroughly tested through extensive experimental evaluations using the continuous query engine CAPE as platform. The evaluations made use of real datasets from multiple domains, including fire monitoring, chlorine monitoring and sensor networks. Moreover, SNIF is demonstrated to be tolerant to noisy datasets.

ACKNOWLEDGEMENTS

Firstly, it is not possible to give enough thanks to my thesis advisor, Professor Elke A. Rundensteiner, for her guidance and support. She has truly represented a guide and stimulus towards the best that I could produce. Her expertise in Database Mangement Systems has helped me to improve my research skills and to prepare for future challenges.

My special appreciation goes to my thesis reader, Professor David C. Brown, for his encouragement and numerous fruitful discussions. He has been very generous in giving his time and careful attention to my research work. His prompt suggestions and constructive criticism were very useful.

I would also like to show my appreciation for Professor John P. Woycheese of the Fire Protection Engineering Department at Worcester Polytechnic Institute. He helped me understand the fire test dataset and provided great support as the domain expert. I would also like to acknowledge the advice and guidance of Professor Murali Mani and Professor Micha Hofri. I appreciate the members of the Database Systems Research Group (DSRG) of WPI for their companionship and unending support.

I am grateful to the Computer Science Department at Worcester Polytechnic Institute for providing us students with excellent infrastructure and a challenging environment. But for the infrastructural help, our research efforts would rarely bring about accomplishments. I express my gratitude towards the head of our department, Professor Michael A. Gennert, for his encouragement. I thank Professor Craig E. Wills and the Graduate Committee for supporting me as a teaching asistant in the department.

This acknowledgement will be incomplete without the mention of my family members. I express my thanks to my wife, Archana, without whose unfailing love this work would certainly have faltered. She was sometimes, totally unexpectedly, so refreshingly helpful and inspiring in her own ways. I am indebted to my wonderful parents - Amitava and Rita, to my two elder brothers - Arindam and Anirban and to my two sisters-in-law - Joyeeta and Sushma. They have always inspired me and provided practical as well as emotional support. I owe them a lot for the successful completion of my thesis work.

Contents

1 Introduction **9**

1.1 Motivation: Importance of Matching Applications 9

1.2 Time-series versus Streaming data 11

1.3 The State-of-the-Art 12

1.4 Approach 13

1.5 Outline of Thesis 14

2 Related Work **15**

2.1 Static Time-series Sequence Matching 15

2.2 Similarity Matching over Live Streams 18

2.3 Text Matching and Data Cleaning 20

3 Preliminaries **23**

3.1 Definitions 23

3.2 Similarity of Sequences 24

3.3 n-Snippet Extraction 26

3.4 Snippet index and matching 29

3.4.1 n versus m Ratio 30

3.4.2 The 2-Level Indices 30

4 Approach **33**

4.1 Overview 33

4.2 Preprocessing Phase 34

4.3 Live Stream Matching Phase 37

4.3.1 n-Snippet Index Lookup 38

4.3.2 m-SnippetCollection Index Lookup 41

5 Experimental Evaluation **46**
5.1 Experimental Setup 46
5.1.1 Datasets 46
5.1.2 Forming Pattern Sequences and Live Streams 48
5.1.3 Experimental Plan 48
5.2 Performance Evaluation 49
5.3 Robustness 53

6 Conclusion and Future Work **56**
6.1 Conclusion 56
6.2 Future Work 56

List of Figures

3.1 Forming n-Snippets from a sequence 27
3.2 n-Snippet Index 31
3.3 m-SnippetCollection Index 31

4.1 Building the Indices 35
4.2 Avg Stdev Sorted Tree 37
4.3 Performing a range query: a snippet probing the ASTree 39
4.4 Snapshot 1: Bag matching in progress 40
4.5 Snapshot 2: Bag matching in progress 41
4.6 Order Checking using Collection Index Lookup 42

5.1 Sample sequences from the EDaFS dataset 47
5.2 Sample sequences from the Motes dataset 1) Temperature 2) Humidity 47
5.3 Sample sequences from the Chlorine dataset 48
5.4 CPU costs for different S_{Col}Threshold and fixed S_PThreshold: 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle. 50
5.5 CPU costs for different S_PThreshold and fixed S_{Col}Threshold: 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle. 50
5.6 Count of Match Nodes (νs) maintained through the runs of the four algorithms 51
5.7 CPU costs for different pattern sizes 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle. 51
5.8 CPU costs for different S_{Col}Threshold and fixed S_PThreshold: 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle. 52
5.9 EDaFS dataset- Average scores of different live sequences with increasing noise. 53

5.10 Motes dataset- Average scores of different live sequences with increasing noise. a) n = 5 and m = 10 and b) n = 1 and m = 10 54
5.11 Chlorine dataset- Average scores of different live sequences with increasing noise. a) n = 5 and m = 10 and b) n = 5 and m = 3 55

List of Tables

3.1 List of notation used 23
3.2 Definition of Terms 26

4.1 Parameters used in the Match Framework 34

5.1 Parameters varied during Experiments on Live Stream Matching . . . 49

Chapter 1

Introduction

1.1 Motivation: Importance of Matching Applications

The recent technological advances in sensor networks and mobile devices have given way to new research challenges related to the efficient processing of data streams that they generated. Streaming time-series similarity matching is one such emerging active research challenges [GW02, GYW02, WSZ04, KPM07, HLMJ07].

Matching of streaming time-series sequences to a set of patterns stored in a database (we call them pattern sequences) is applicable to a broad range of applications. Temperature/humidity/CO readings from sensors installed in a building, stock prices, vital statistics of a patient and network traffic data all form suitable examples of streaming time-series. The core technology for matching could be used as solution to critical problems such as environmental monitoring of hazardous waste and poisonous attack clouds as well as more mundane purposes such as network traffic monitoring and click stream in web tracking.

Several such applications exist that require on the fly matching of live streams against pattern sequences. One such critical application that in part motivated this thesis work is *monitoring of fire and prediction of its propagation*. Fire propagation and containment of hazardous chemical spills and contamination are examples of modern disasters requiring crisis management support. The situation may arise from naturally occurring phenomenon (e.g., lightning ignition of a Wild land/Urban Intermix fire) or from induced threats (e.g., arson or terrorist attacks).

As part of this thesis work we addressed critical challenges relevant to the prob-

lem of fire monitoring and propagation prediction. In particular, we propose to explore the problem of run-time monitoring and prediction of fire spread in building structures, an interdisciplinary topic involving aspects of Fire Protection Engineering and Computer Science. Our proposed matching technique could be employed in *Building Control Systems* to monitor the location and spread of fire and smoke by observing sensors placed within the building structure; analyzing the measurement streams and controlling associated data acquisition rates at run-time. Therefore, we aim to match live sensor streams against pattern sequences captured during various fire events in an effort to predict fire spread.

In this thesis, we propose an efficient framework for matching a streaming time-series against a set of data sequences (we call them pattern sequences). We call it *n-Snippet Indices Framework* (in short SNIF). Our approach divides the matching task into two phases:

1. *Offline Preprocessing Phase*: where the pattern sequences are processed offline and stored into a 2-level index structure; and

2. *Online Live Stream Matching Step*: streaming time-series (or the live stream) is on-the-fly matched against the indexed pattern sequences.

Based on the notion of n-Grams used for textual information retrieval [Coh97] we now introduce the concept of n-Snippets for numeric data as the unit for matching. The insight is to match small snippets of the live stream against prefixes of the patterns and maintain them in succession. The longer the pattern prefixes identified to be similar to the live stream, the better the confirmation of the match. In our framework, the live stream matching is performed using two levels of matching: bag matching for matching snippets while allowing for partial disorder in and order checking for maintaining the lengths of the match.

As part of our experiments we worked on the EDaFS dataset [WVM04] to test the applicability of our stream matching system on the fire domain. We also used the Motes [SPF] and the Chlorine [EPA] datasets for evaluating the effectiveness of our proposed approach on data from other domains. We find that our framework is quite effective in matching sequences from various datasets.

In the next section we compare streaming data with static time-series data. Thereafter in Section 1.3 we discuss the current state of research in streaming time-series matching. Lastly, we list our contributions in Section 1.4.

1.2 Time-series versus Streaming data

A *live stream* is defined as a series of relational records, usually assumed to have infinite length. More recent data elements are considered more meaningful than the older ones [BBD+02]. A time-series is a sequence of real numbers representing values from some given domain at specific points in time. Time-series data stored in a database are commonly called data sequences. A live stream that is composed of a time-series data is called a *streaming time-series* [GW02, GYW02].

The processing of queries over streaming time-series should be handled in a different way from traditional time-series due to the following reasons according to Bobcock et. al. [BBD+02]. First, the elements in a live stream must be processed online due to the real-time nature of the application requirements. The data tends to be continuously appended to the end of the live stream at high arrival rates. Thus, the most recent elements typically are processed before the next elements arrive, unless some elements are processed collectively. As a comparison, in traditional static time-series stored in a databases, there is no limit on the processing time. For example in a financial application (online) analysing the daily stock trends from streaming data as opposed to another such application (offline) that requires to analyse the stock trends from data stored in a database for each decade (70′s, 80′s, 90′s, etc.).

Second, the streaming time-series are assumed to have infinite lengths, and hence cannot be stored in a database in their entirety. Since static time-series are assumed to be finite, algorithms for processing them can access the whole sequences either sequentially or by preprocessing into indexed form for faster access. For these reasons algorithms defined for time-series cannot be easily adapted for streaming time-series. Third, any portion of the streaming time-series obtained previously can not be assumed to be available again at a later time. Since the streaming time-series are assumed to have infinite lengths, the data obtained in the far past must either be explicitly stored by the system in a compressed form or simply discarded. On the contrary, in traditional static time-series database, the entire time-series can be retrieved at any time. Thus multiple passes or some indexed access can be performed.

1.3 The State-of-the-Art

Similarity queries have been classified in the literature into the following two classes:

1. *Whole Matching.* The sequences to be compared have the same length n.

2. *Subsequence Matching.* The query sequence is smaller; we look for one or more subsequences in the large sequence that best match the query sequence.

Within both the whole matching and the subsequence matching cases, another classification based on the query output [AFS93] is often given as:

1. *Range Query.* Given a query sequence, find all subsequences that are similar within distance ϵ instead of being identical.

2. *k-Nearest Neighbor Search.* Given a query sequence S_Q, find the top k subsequences from the patterns that are more similar to S_Q compared to all other possible subsequences.

3. *All-Pairs Query or Spatial Join.* Given n query sequences, find the pairs of sequences that are within ϵ of each other.

ϵ is the distance measure that controls when two sequences should be considered to be sufficiently similar. Typically the solutions to all these matching problems perform approximate matching rather than exact matching. Noisy data from real-time data sources require these match techniques to address gaps and skews between the sequences being matched.

For static time-series data, several well established algorithms [AFS93, ALSS95, FRM94, WW00, CN04] have been proposed. The solutions cover the complete classification of similarity queries given above. Various similarity measures such as Euclidean Distance [KPL04], Dynamic Time Warping [KP99], Fourier Transform [AFS93], etc. have been studied for use in similarity matching (finding pattern sequences from the database similar to the given query sequence).

For similarity matching over streaming time-series data, some solutions are based on domain specific models and techniques such as for medicine [WSS+05] and for finance [WSZ04, BS03].The challenges are aggrevated in a streaming context by issues such as fast arrival rates, infinite length of live streams, limited memory and the need for real-time response. Query sequences can be formed out of the live

streams up to the data points that have arrived till the current time. Performing a whole match between the query sequences and the pattern sequences is not possible unless the whole of the query sequence is available. Algorithms proposed for live stream matching mostly provide prediction based solutions range query [GW02] and k-NN Search [GYW02] respectively. Gao et. al. [GW02, GYW02] utilize extrapolation of the already arrived live stream data using some error models (square root, linear and square errors). However, they employ similar functions to generate synthetic pattern sequences as well as live stream. Such prediction-based systems may not be applicable for sensor data due to noise. Moreover, the use of Discrete Fourier Transform as the match measure is quite compute-intensive thus may not be most suitable for live streams where real-time performance is of utmost importance.

Han et. al. [HLMJ07] claim to be the only ranked subsequence matching (k-NN) solution. They propose to use Dynamic Time Warping approaches which suffer from dimensionality curse since the similarity measure computation requires each data point. One of the overreaching questions we may ask at this point is to what degree we can develop general-purpose stream matching query technology that can be applied to a broad classification of similarity queries?

1.4 Approach

We propose a new generic framework for discovering similarity between live stream and pattern sequences. We call it *n-Snippet Indices Framework* (in short SNIF). As the live stream is infinite we need to work with chunks of data from the live stream, we introduce the concepts of n-Snippets and m-SnippetCollections. The pattern sequences are processed offline into two levels of indices − n-Snippets and m-SnippetCollections. Our proposed live stream matching is performed in two stages:

1. *bag matching*, and
2. *order tracking*

The bag matching step performs approximate matching of small chunks of live stream data to discover subsequences of pattern sequences within the live stream. Moreover, the order tracking step is analogous to stitching the adjascent subsequences to discover which of the pattern sequences match the live stream and incrementally computing how closely each such pattern sequence matches. Therefore,

our approach can be used for subsequence matching as well as building upon the subsequences to find whole matches (if any). The method can perform range queries as well as nearest neighbor searches.

The SNIF framework addresses the concerns of the streaming environment. The preprocessing of the pattern sequences into an index structure saves processing time during the live stream matching step. The framework is also applicable to matching variable length pattern sequences (S_P), since the patterns are divided into two levels of indices. Our approach performs robustly and accurately for considerable amounts of noise in the live stream data. It is tolerant to noise such as missing data, extraneous data or out-of-order data (details provided in the evaluation Section 5.3). We present different variations of our algorithm, namely *Best One* and *Best K*, based on the number of matches maintained for each pattern sequence, which allow us to trade off between speed and accuracy.

SNIF uses a set parameters which can be used to tune the pre-processing and the matching steps according to the domain. We use data statistics (average & standard deviation) as the match measure. They are incrementally computable and not much compute-intensive. However, it is also fairly easy to switch these data statistics with any other matching measures such as DTW, DFT or Euclidean Distance. SNIF remains effective as long as the pattern sequences are preprocessed into the indices using the match measure M and the live streams are matched against the index structure using the same match measure M.

1.5 Outline of Thesis

The thesis document is organized as follows. Chapter 2 reviews existing work related to similarity matching in time-series databases. Chapter 3 presents the formal definitions of the live stream and pattern sequences as well as the live stream matching problem. It also includes discussions regarding the concepts of n-Snippets and m-SnippetCollections and the index structure. Chapter 4 describes the steps of the matching framework in details. Chapter 5 presents the results of performance and robustness evaluation. Chapter 6 summarizes and concludes our work.

Chapter 2

Related Work

In this section we explore the existing research in similarity matching for time-series data. Similarity matching problems can be broadly classified as Range Query and Nearest Neighbor Search. We explore proposed solutions for both. For static time series data, we find that several well established algorithms [AFS93, ALSS95, FRM94, WW00, CN04] have been proposed to target each of the different classifications of the similarity queries (defined in 1.3). Since continuous stream processing gained importance, there have been several attempts to extend the traditional sequence matching techniques to work for the streaming environment. We review such sequence matching techniques [GW02, GYW02, HLMJ07]. Finally, we investigate some text matching techniques [KWLL05, Coh97, LA96, BYRN99, MSLN00] from the information retrieval world that form the basis of our proposed n-Snippet inverted index solution.

2.1 Static Time-series Sequence Matching

Pioneering work in sequence matching for static time series has been conducted by Agrawal et. al. [AFS93, ALSS95]. They propose an indexing method for time sequences to process similarity queries. Their proposed solution works in two steps: index building and similar sequence matching. Their idea of an index suits the real-time response criteria of applications such as ours as an index helps quicken the processing. In the index building step each data sequence is transformed into a lower-dimensional representation and stored in an R*-Tree. They use the Discrete Fourier Transform (DFT) to map time sequences to the frequency domain.

Over the years several researchers have used other lower-dimensional transformations such as DTW, PAA, or SDV. In the sequence matching step, the query sequence is transformed into the lower-dimensional points similar to the patterns in index building step and a range query identifies the candidates lying within the tolerance. This eliminates chances of false negatives (potential candidates being dismissed), however there may be false positives (several non-candidates in the result) present in the candidates. A confirmation step follows this range query evaluation where the candidates are matched closely with the query sequence to eliminate the false alarms. This forms a great framework for a similarity matching solution with the sequence matching being performed in steps. However, the proposed technique is for whole sequence matching only. This makes their approach inapplicable for streaming time-series data. Moreover, a Fourier Transform is a very expensive operation [KPM07].

A range subsequence matching algorithm for static time-series data is given by Faloutsos et. al. [FRM94]. Their technique is an extension of the whole matching solution. They use window construction to divide time-series sequences into windows. In the index building step, the data sequences are partitioned into sliding windows, the data points in each window are transformed into lower-dimensional values and the transformed points are stored in a R*-Tree. For subsequence matching the query sequence is partitioned into disjoint windows, each window is transformed into low-dimensional values and a range query is performed against the R*-Tree to extract candidates. In our problem the live stream is infinite and processing in successive windows is required, we explore more options into how windows can be constructed. DualMatch [MWL01, LPK06] and GeneralMatch [MWH02] are variations of FRM with significant performance improvements. Both resemble FRM in index building and sequence matching steps but differ from FRM in the logic of window construction. DualMatch works on the notion of duality of construction window, data sequences are split into disjoint windows where as query sequence is partitioned into sliding windows. GeneralMatch defines J-Sliding windows and J-Disjoint windows.

Wang et. al. [WW00] have studied database techniques that support fast searches for time-series whose contents are similar to the users' specification. The content types include shapes, trends, cyclic components and so forth. Since similarity searches over such contents are complex, traditional database techniques are

slow particularly with high data volumes. They propose techniques to answer these queries based on approximation. They present two approximation methods. One method is based on the linear B-spline wavelet function. The wavelet transform decomposes a time series into a linear combination of given basis functions known as wavelets. The wavelets with the most significant coefficients in the decomposition are selected. The linear combination of the selected wavelets is the decomposition of the time series. The other method uses the least square method to fit a given time series into consecutive line segments. This line fitting method finds the best fitting line segment closest to the subseries. Thus starting from the first time point A of the series, they find the farthest point B such that when the subseries over [A,B] is considered, the distance of the given time series to its best fitting line is less than a given threshold. Also, no other point B nearer to A violates the above condition. This process continues with B as the starting point and so forth. These approximation methods can be combined with indexing. Thus it is possible to build indexing structures on the approximated series to further speed up the search.

Wong et. al. [WW03] advocate for time warping as a more robust distance measure than Euclidean distance. Dynamic time warping (DTW) allows matching variable length sequences as well as time skewed sequences. They present a method that supports dynamic time warping for subsequence matching within a collection of sequences. Their method takes full advantage of the sliding window approach and can handle queries of arbitrary length. Certain limitations of DTW are that it does not satisfy the triangle inequality, so that spatial indexing techniques cannot be applied. DTW also does not exploit dimensionality reduction,and also requires each data value for distance computation. As we are dealing with real numbers we would ideally like to be able to apply some staistical or transformation technique to summarize groups of data values. Hence DTW does not look like a potential candidate for application to the live stream matching problem.

The traditional k-NN search algorithms use a minimum priority queue to find the k-nearest objects from a query object. Hjaltason et al. [HS95] and Roussopoulos et al. [RKV95] proposed variations of k-NN. The object is assumed to be stored in the multidimensional index as an MBR (minimum bounding rectangle). Each time, for each such object the minimum priority queue holds the topmost k nodes based on distances from the query object. In the k-NN algorithms, the records of the queue are repeatedly replaced with the new nearest ones, and eventually k

objects that are nearest from the query object are identified. Here we note that several solutions for both range and k-NN similarity matching propose to use an index structure for processing time-series sequences before matching. As the pattern sequences are available to us, we believe that, prior to matching them against the live stream, we could process them into some index structure too. In the case of similar sequence matching, Keogh et al. [KCMP01] and Chan et al. [CFY03] proposed k-NN search algorithms for the whole matching problem. Keogh et al. proposed a novel dimension reduction technique, called APCA (adaptive piecewise constant approximation), and they describe a k-NN whole matching algorithm based on the basic k-NN solutions [HS95, RKV95] in order to demonstrate superiority of their reduction technique. Chan et al. proposed another k-NN whole matching algorithm that first finds an upper bound of search range using Roussopoulos et al.′s k-NN solution, and then performs the range whole matching using the bound.

2.2 Similarity Matching over Live Streams

Gao and Wang [GW02] proposed the first similarity matching solution for live streams, which is a prediction-based similarity matching technique. The system monitors the streams to search patterns that are relevant and solves the problems of Nearest Neighbor and h-Near Neighbor (h being the distance tolerance) for whole matching. The technique uses the already arrived data to predict future subsequences. They pre-compute the distances between the query sequence and the predicted subsequences employing Fast Fourier Transform (FFT). FFT computes cross correlations of the predicted series and patterns to get predicted distances between the incoming series at future time positions and the database patterns. When the actual data arrives, the prediction error with the predicted distances is used to filter patterns that cannot possibly be nearest neighbors. This provides fast responses. They observe that with reasonable prediction errors the performance gain is significant. However, there are inherent limitations in the method. The technique has the overhead of adjusting the prediction error, which can be significant if the actual data is much different from the predicted series. Also the technique must compute the distance for each of the query sequences in the database at each time unit, making it difficult to maintain large number of query sequences in the database. However, the very idea of forming large number of query sequences out of the live stream

seems a very naive approach for live stream matching. Synthetic data is used for both the patterns and the live stream generation. Hence applicability to real data is unknown.

Gao et. al. [GYW02] propose another sequence matching method which solves the k-Nearest Neighbor problem using prefetching. This approach finds the most similar k nearest query sequences from the database against the live sequence. Here the k is the number of query sequences and not the tolerance. This technique transforms the query sequences into lower-dimensional points, and stores them to disk in a multi-dimensional index. As the new data points arrive, k nearest query sequences are searched from the database similar to the live sequence. This method uses prefetching in which the arrived data values are used to predict k-NN candidates for the near future. The authors claim that the index and the candidate query sequences are processed during the idle time between data arrival, thus saving on CPU costs. However, this does not seem to be reducing the cost much. Due to the multidimensional index and the amortized disk reads, this technique can handle a large number of query sequences. However, disk storage is only useful for a very large number of queries and not if the queries can be handled in main memory. Another limitation of this technique is that it solves k-NN for only fixed length patterns and the method relies on a fixed tolerance of the pattern sequences. However, the datasets [WVM04, SPF, EPA] that we consider have pattern sequences of different lengths, thus this approach is not applicable to such real datasets.

Kontaki et. al. [KPM07] propose the IDC-index for streaming time-series data in which DFT computations are performed incrementally over the streaming sequences. They address both range query and k-NN search problems. An R*-tree storing the dimensionality-reduced points is maintained for the streaming time-series data. Application of computationally expensive FFT over the live stream and simultaneously building an index structure requires a response time longer than desired by critical real-time applications. They focus on both range and k-nearest neighbor queries for situations where the query sequence change over time. In their case the problem is that the DFT coefficients of a streaming time series must be updated when a new value arrives. If they update the index every time a new value becomes available, the overhead may be prohibitive due to additional page accesses. To avoid continuous deletions from and insertions into the R*-tree, they use a deferred update policy. They also use a simple heuristic approach to adapt to the update frequency of the

data streams and maintain it to a specified level. Since the purpose of the system is to run against the infinitely arriving time-series data and detect if it matches with any of the pattern data sequences, maintenance of the R*-Tree for the whole live streaming series is a big overhead.

Han et. al. [HLMJ07] present techniques for ranked subsequence matching under time warping, that finds top-k data sequences most similar to a query sequence. They introduce a notion of minimum-distance matching-window pair MDMWP. They claim that mdmwp-distance is a lower bound between the data subsequences and a query sequence. The mdmwp-distance can be computed prior to accessing the actual subsequence. Based on the mdmwp-distance, they then develop a ranked subsequence matching algorithm to prune unnecessary subsequence accesses. Next, to reduce random disk I/Os and bad buffer utilization, they develop a method of deferred group subsequence retrieval. They then derive another lower bound, the window-group distance, that can be used to effectively prune unnecessary subsequence accesses during deferred group-subsequence retrieval.

Overall, the state-of-the-art stream sequence matching algorithms have been extensions to the well-established sequence matching techniques for static time-series data. Most of them have been reusing the dimensionality reduction as used for static time-series. However, on-the-fly dimensionality reduction operations are very expensive. Moreover, most of the existing sequence matching techniques focus on either whole matching range query and k-NN Search [AFS93, CFY03, GW02, GYW02, KPM07] or range query in subsequence matching. Han et. al. [HLMJ07] claim to be the only ranked subsequence matching (k-NN) solution. Their method uses DTW approaches which suffer from the dimensionality curse. They also require all the data values for distance computation. Also, DTW does not satisfy the triangle inequality, so that spatial indexing techniques cannot be applied. These limitations motivated us to explore new avenues for solving the prefix matching problem. One such technique for matching that is yet unexplored for sequence time-series matching is n-Gram matching using inverted-index.

2.3 Text Matching and Data Cleaning

An inverted index [BYRN99] is a frequently used datastructure in the Information Retrieval world. Inverted index is a term-oriented technique for quickly searching

documents containing a given term. Here the document is a finite sequence of characters and a term is a subsequence of a document. Term and posting list combined together form an inverted index. The posting list is a list of postings and each posting contains information about the occurrence of the term. Based on the definition of the term, an inverted index is classified as [MSLN00, WMB99, LA96] :

1. a word-based inverted index, a word is used as a term;

2. an n-Gram inverted index, a sequence of n characters is used as a term.

An n-Gram inverted index uses n-Grams as indexing terms. If there is a document d consisting of characters C_0, C_1, ... C_N. An n-Gram is a subsequence of length n [KWLL05]. n-Grams can be extracted from document d using the 1-sliding technique, i.e., sliding a window of length n from C_0 to C_{N-n} and storing the characters located in the window. For instance the j^{th} n-Gram will be C_j, C_{j+1}, C_{j+2},..... C_{j+n-1}.

Query processing is done in two steps:

1. split a given query string into multiple n-grams and search the posting lists of those n-grams; and

2. perform merge join between those posting lists using the document identifier as the join attribute [BYRN99].

Kim et al. [KWLL05] propose the two-level n-gram inverted index (henceforth we will refer to it as the n-gram/2L index) that significantly improves the query performance while preserving the advantages of the n-gram inverted index. The proposed index eliminates the redundancy of the position information that exists in the n-gram inverted index. The proposed index is constructed in two steps:

1. extracting subsequences of length m from documents, and

2. extracting n-grams from those subsequences.

They prove that this two-step construction is identical to the relational normalization process that removes the redundancy caused by a non-trivial multivalued dependency. The n-gram/2L index has excellent properties:

1. it significantly reduces the size and improves the performance compared with the n-gram inverted index with these improvements becoming more marked as the database size gets larger;

2. the query processing time increases only very slightly as the query length gets longer.

Experimental results using databases of 1 GByte show that the size of the n-gram/2L index is reduced by up to 1.9∼2.7 times and, at the same time, the query performance is improved by up to 13.1 times compared with those of the n-gram inverted index.

However, the scope of n-Gram is restricted to text matching. We propose to develop a framework for sequence matching for streaming time series data using an underlying datastructure similar to n-Gram/2L. We extend this concept of n-Grams to apply it to numeric time-series data (we call it n-Snippets) and develop approximate matching methods for subsequence matching of the live streaming data. Our framework maintains the count of the subsequences matched up to the current time to give us a picture of how much of the live stream has matched a pattern. We do not need to maintain the live stream for the whole matching.

Chapter 3

Preliminaries

3.1 Definitions

In this section we define the basic concepts and terminology as backgound for our work. We begin by listing the notations we use (Table 3.1).

Symbols	Definitions
S_{ID}	Sequence with its unique identifier ID
Len(S)	Length of a sequence S, S is either the S_L or the S_P
S[i]	i^{th} data value in the sequence S
S[i : j]	The sub-sequence of S from i^{th} to j^{th} data value, inclusive for any i, j $\in$ I ; i $\leq$ j.

Table 3.1: List of notation used

A live stream sequence S_L is a time-series data stream to which new data entries are continuously appended at every time unit. The arrival of data may be in equal or unequal intervals. At any time t_i, the live stream sequence consists of a sequence of data values collected starting at time t_0 until the current time t_i.

$S_L[\ t_0 : t_i\ ] =$ d[t_0], d[t_1], ..., d[t_i]

A d[t_i] can be a single value or multiple values. For example, a temperature reading from a sensor or a combination of temperature and humidity values at time t_i.

A pattern sequence S_P is a finite time-series data sequence, such as a sequence of sensor readings collected over time, which is designed to record the characteristic behavior during a phenomenon (such as a fire event). The S_L is matched against a

set of S_P to identify the most similar S_P in the set.

For simplicity we consider our sequences (both S_L and S_P) to consist of single valued data points. However, the techniques would apply to multi-valued data points.

The live stream matching problem can be defined as follows:

Given a set of S_P, continuously match the data of the S_L with the set of S_P. As new data keeps appending to the S_L, we need to dynamically include the current data and detect the S_P similar to the S_L. The definition of similarity is explained in the next paragraph.

3.2 Similarity of Sequences

For the purpose of finding a suitable similarity measure for our matching framework, we examined the following datasets using MATLAB: 1) the EDaFS fire dataset [WVM04], 2) the Chlorine dataset [EPA], and 3) the Motes dataset [SPF]. Several similarity measures for time-series numeric data have been proposed over the years. Some of the commonly used ones are Euclidean Distance [KPL04], Dynamic Time Warping [KP99], Fourier Transform [AFS93], etc. Descriptive statistics are also good candidates for measuring similarity between numeric sequences as they summarize the data values they represent.

The statistics and similarity measures that we tested for the sequences of each of the datasets are: Fast Fourier Transform, Euclidean Distance, average, slope and standard deviation. There are several requirements that a similarity measure must meet in order to be suitable for comparing streaming time-series numeric data. We compared the candidate similarity measures with respect to the following requirements of the live stream matching problem.

Firstly, the data source may be noisy. The noise from the sensors necessitates smoothing of data before matching. Matching against noisy data can increase chances of false alarms. For this Fourier Transform (FT) has been recognized as a suitable candidate. Using FT one can limit the number of Fourier coefficients to include starting from the lowest frequency. High frequency coefficients can be eliminated as noise. The choice of how many coefficients to select greatly depends on the dataset and may also vary according to the domain. Smoothing could also be achieved by applying a moving average over the sequence.

Secondly, Reduction of data points required for distance computation is considered to be a big plus in time-series sequence matching. Distance computation after applying Fourier transform achieves this reduction from time-series data points to frequency coefficients. Euclidean Distance requires each data point for computation. Hence it is not a good choice.

Thirdly, since the live stream is continuous, a match measure that is incrementally computable is preferred over having to recompute distance measures from scratch upon arrival of each new data. Fourier Transforms [KPM07] and Data Statistics (although not all of them are incrementally computable) satisfy this criterion. Moreover, another criterion is that the measure be efficiently computable. On-the-fly computations are required due to the dynamic nature of S_L. More computation means more response time if we consider limited CPU resources. We note here that the Fourier Transform is considered quite compute intensive [KPM07]. So for this reason one may want to look elsewhere.

The conclusion from our analysis of the datasets was that FFT, average and standard deviation all formed good distinguishers between the sequences of the datasets. However, application of FFT over chunks of live stream is computationally expensive. Average is one suitable candidate as it takes care of data smoothing; thus eliminating chances of false alarms. It is also incrementally computable and not so compute intensive. However, average itself does not serve as a sufficient criterion since it may not facilitate matching the shape of the sequence more precisely. Also the degree of smoothing is an important factor. We select standard deviation as another suitable data statistic. It also satisfies all the above requirements, just like the average statistics.

Moreover, we were able to observe empirically that it forms a significant distinguisher between the pattern sequences in the datasets we examine. However, standard deviation by itself is not a strong candidate since the same standard deviation value can occur at totally different temperature bandwidths,even though we would not call them similar. Hence, we choose a combination of the two (average & standard deviation) as our similarity criteria, making it a more reliable match measurethan either alone. This also helps us with reduction of data points for the distance computation as we match only the two data statistics over n data values assuming $n \geq 2$.

3.3 n-Snippet Extraction

In this section we discuss n-Snippets and m-SnippetCollections that form the building blocks for our match framework. We first define some terms we use in the discussion through the rest of this section (see Table 3.2).

Term	*Definition*
n-Snippet	It is our unit for matching. Average and Standard Deviation value pair over a collection of n consecutive data values. e.g. $\langle$ t , Avg(S[1:5]), Stdev(S[1:5])$\rangle$. For simplicity it will be referred to as a snippet.
1-Sliding technique	It is the act of collecting groups of n consecutive data values by shifting through a dataset by 1 data point at a time. snippets are extracted from the S_P using 1-sliding technique.
m-SnippetCollection	Collection of m consecutive snippets, m+n-1 consecutive data values form a single m-SnippetCollection. For simplicity we will refer to it as a collection of m snippets or simply a collection
Occurrence list	For a term (here snippets or collections), it consists of the identifier of the S_P and the list of offsets where the term occurs within the S_P.
Inverted index	Consists of map between the terms (here snippets or collections) and their occurrence list. Refer to Figure 3.2 or Figure 3.3
Probe	Another term for *index lookup* to extract the occurrence list.
Bag matching	When snippets of the S_L are matched against the set of S_P without considering the order in which n-Snippets occur in the S_P. Usually the extent of match is determined by the number of n-Snippets matched / total number of n-Snippets.
Order tracking	Sequence matching step where the order of occurrence of components (here, collections) is checked to establish a match between them. The match can be reported till any position i for the i^{th} component and corresponding extent of match between the sequences is computed over all components starting the first upto the i^{th} component.

Table 3.2: Definition of Terms

Having chosen the similarity measure as (average & standard deviation), the next question is the choice of the window *size* and *type* over which to compute the average and standard deviation.

An n-Snippet is our unit for matching. An n-Snippet consists of a collection of n consecutive data values, where n determines the degree of smoothing involved

during matching. For simplicity we will refer to n-Snippets as simply s̈nippetsäs n simply denotes the size of the snippet.

Here is an example of how snippets of size n can be extracted from a sequence S. Without loss of generality, say n = 5 for the rest of the discussion. To form a snippet of length 5, we collect 5 consecutive data values and compute the similarity measure (average & standard deviation) for this snippet. A typical snippet will look like $\langle$ t_i, Avg(S[t_i:t_{i+4}]), Stdev(S[t_i:t_{i+4}])$\rangle$, where the t_i value is t_2, the timestamp of the middle element. t_i also denotes the position of the snippet in the sequence S. Even starting or end element′s timestamp will also work equally as long as it is consistent for all extracted snippets.

For the choice of window type we chose sliding windows for extracting successive snippets. However disjoint windows could be used. Snippets of disjoint windows are beyond the scope of this discussion. We extract snippets from a sequence using the 1-sliding technique (defined in Table 3.2). Suppose the sequence is a time-series of temperature readings from sensor DAN2 (taken from the EDaFS [WVM04] dataset) as shown in the Figure 3.1.

TimeStamp	SensorID	Temperature
0	DAN2	22.97265
1	DAN2	22.98296
2	DAN2	22.97686
3	DAN2	22.98142
4	DAN2	22.99248
5	DAN2	22.99896
6	DAN2	23.00554
7	DAN2	23.0269
8	DAN2	23.04316
9	DAN2	23.07767
10	DAN2	23.12692
11	DAN2	23.17159
12	DAN2	23.25956
13	DAN2	23.45956
14	DAN2	23.79367
15	DAN2	24.35931
16	DAN2	25.12741
17	DAN2	26.06227
18	DAN2	27.27287
19	DAN2	28.6983
20	DAN2	30.40258
21	DAN2	32.26899

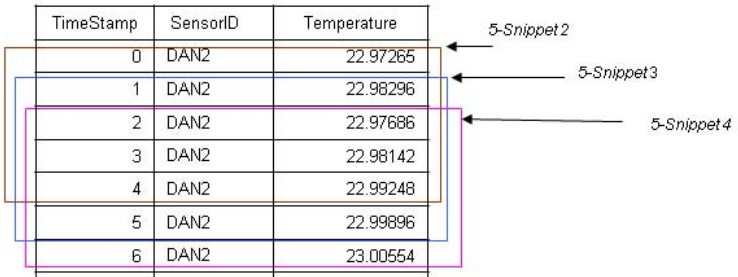

Figure 3.1: Forming n-Snippets from a sequence

Since, we 1-slide, the consecutive snippets are ⟨ 2, 22.98, 0.008 ⟩, ⟨ 4, 22.99, 0.0106 ⟩, ⟨ 4, 23.001, 0.01517 ⟩, and so on. There would be Len(S)n+1 snippets formed from a sequence of length Len(S). For example, from a sequence of 22 data points (as in Figure 3.1), 18 (i.e., 22-5+1) snippets can be formed.

By analyzing various datasets such as the EDaFS fire dataset [WVM04], the Chlorine monitoring dataset [EPA] and the Network Motes dataset [SPF] we found that lots of snippets are common across the S_P. Hence, we consider bag matching (defined in Table 3.2) as one possible design choice for matching snippets. However, bag matching does not serve sequence matching well since similar trends of statistical behavior such as rise or fall of a curve overtime would thus not be discernable from overall fluctuations over time. In other words, we cannot say that a rise in temperature for 5 seconds, then a fall for another 5 seconds and then again a rise for 5 seconds will be similar to a fall for 5 seconds and then a rise for 10 seconds. So instead we may opt to match and report if the live snippets, as they are formed out of the S_L, are matching in the exact order as snippets occurring in the S_P. We call this order checking. Order checking over snippets however seems like a very strict constraint. Consider the example of the following consecutive snippets of length 5 (corresponding to the above dataset):

⟨ 10, 23.362, 0.244 ⟩, ⟨ 11, 23.608, 0.4319 ⟩, ⟨ 12, 23.999, 0.6757 ⟩, ⟨ 13, 24.56, 0.939 ⟩, ⟨ 14, 25.323, 1.2372 ⟩, ⟨ 15, 26.304, 1.5424 ⟩, ⟨ 16, 27.512, 1.8765 ⟩, ⟨ 17, 28.941, 2.2063 ⟩.

The snippets show a trend of gradual increase in both the average value and the standard deviation. As we see the data points we see that from timestamp 10 to timestamp 17, the slope transforms from plane to steep rising. We will be matching not for the exact average and standard deviation values, but do range search (defined in section 2.1). But if there is a slight alteration like swapping between snippet 11 and 12, which might occur due to noise in S_L, it should still be considered as a match. Hence, as a good framework should support approximate matching, we allow some flexibility by preparing for some level of disorder between the occurrences of snippets in a sequence. Another alternative is to use the concept of collections of snippets (defined in Table 3.2) as explained in the following section.

To compare two snippets against each other we can measure the Euclidean Distance between them. The Euclidean distance can be computed using just the average and the standard deviation values that summarize the data values of the snippets,

rather than using each data value itself. The distance comparison between any arbitrary snippet pair (Snip_A Snip_B) is shown in equation 3.1. However, weighted Euclidean Distance is also one choice if we wish to give one measure (out of average and standard deviation) more importance over the other.

$$\Delta(\text{Snip}_A,\text{Snip}_B) = \sqrt{(\text{Average}_A - \text{Average}_B)^2 + (\text{Stdev}_A - \text{Stdev}_B)^2} \tag{3.1}$$

3.4 Snippet index and matching

As an alternative to order checking at the snippet level we propose to have two levels of matching: *Bag matching* across snippets within a collection of m snippets and *order checking* across the collections of snippets for a sequence. An m-SnippetCollection is a collection of m consecutive snippets extracted from a sequence (either a S_P or the S_L). Forming collections of m consecutive snippets out of sequences divides the sequence into $\lceil$(Len(S)/m+n-1)$\rceil$ groups of consecutive snippets. Alternatively, one can say that each snippet collection consists of m+n-1 consecutive data values. Like the term snippet we will use the term c̈ollectionöf snippets to denote m-SnippetCollection, where m will simply denote the number of snippets within a collection. In other words, due to the way snippets are extracted, two consecutive collections of snippet just overlap by n-1 data values. The purpose of introducing the collection of m snippets is, on one hand, to allow some margin of disorder in finding the snippets of a sequence and, on the other hand, matching the order of the occurrence of collections. We will now call it a match only if the consecutive collections of snippets of the S_L are found in exactly the same order as the successive collections of snippets occur in S_P. Another benefit of introducing collections of m snippets is to get rid of the redundancy caused by a non-trivial Multivariate Dependency [KWLL05].

Now two inverted indices are formed and used for matching (for each abstract match level). The front-end index or the snippet index, where the occurrence list for each snippet now contains the identifier of the collection of m snippets and offsets where the snippet occurs in the collection. It is used for bag matching of snippets to report fractions of m-SnippetCollections matched. The back-end index or the m-SnippetCollection index is the one used for the order checking (defined in Table 3.2) of the collection within the set of S_P. Due to the evident merits of the snippet indices in 2 levels, given in the above paragraph, we choose two indices instead of

the single snippet index.

3.4.1 n versus m Ratio

Here we discuss the effect of considering different values for n and m over the functionality of the framework.

n determines the degree of smoothing. To preserve the significant patterns of the sequence yet be able to eliminate noise, n needs to be a much smaller value compared to the sequence length ($n \ll Len(S)$). Setting $n = 1$ corresponds to using the original sequence. On the other hand, setting n to a larger value may cause over-smoothing. Hence, smoothing over a medium size of data values in our case $3 \leq n \leq 8$ has been found to be a good choice.

m is the degree of allowed randomness in the snippets while still calling it a match. Ideally we would avoid the choice of extreme values for m. m =1 will correspond to *order checking* over every individual snippet. However, ironic though, $m = Len(S)$ (i.e., equal to the size of the sequence) will also mean matching just at the snippet level, but *bag matching* of all the snippets in the sequence. In that case we will be order checking just 1 collection of m snippets. Hence, for almost all domains we will keep low value of m (say $3 \leq m \leq 30$) compared to sequence sizes ($m \ll Len(S)$).

3.4.2 The 2-Level Indices

The snippets are extracted during preprocessing from the S_P as explained in Section 3.3. Simultaneous to the extraction, collections of snippets are formed as well. Say we consider m as the size of the collection. So every time we collect m number of snippets, they are grouped together and given a unique identifier. The snippets are loaded into the snippet inverted index (we call it the front-end index, as during matching it is matched against the snippets of the live stream). Similarly, the m-SnippetCollections are loaded into the m-SnippetCollection Index (we call it the back-end index, as this is not matched against the live stream, it is referenced for the match measurement though).

The front-end inverted index (Figure 3.2) uses snippets as indexing terms. For each snippet there is an occurrence list that contains information about the occurrence of the snippet within a collection. The occurrence list information corresponds

to a vector $\langle$ SCol$_{ID}$,$\rangle$ o$_1$,o$_2$,..,o$_i$ $\rangle$ $\rangle$ i.e. the identifier of the Snippet Collection in which the snippet exists along with each of the offsets o$_i$ within the Collection where the snippet occurs.

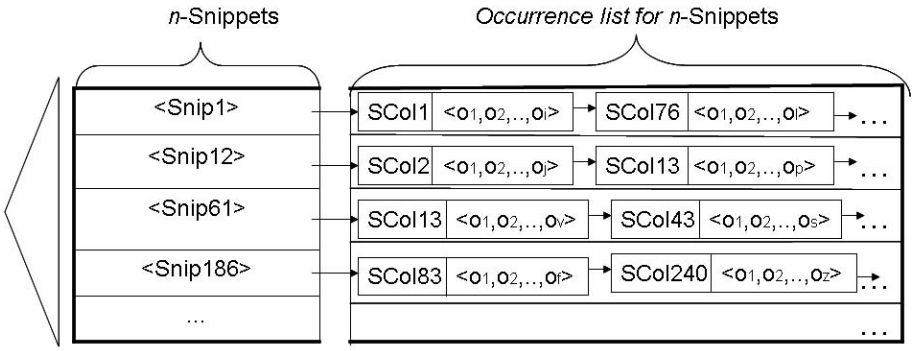

Figure 3.2: n-Snippet Index

The back-end inverted index (Figure 3.3) uses the identifier of collections as indexing terms. For each collection there is an occurrence list that contains information about occurrence of the collection within the S$_P$. The occurrence list information is $\langle$ S$_{ID}$, $\langle$ o$_1$,o$_2$,..,o$_i$ $\rangle$ $\rangle$ where S$_{ID}$ is the identifier of the S$_P$ in which the collection exists along with each of the offsets within the S$_P$ where the collection occurs.

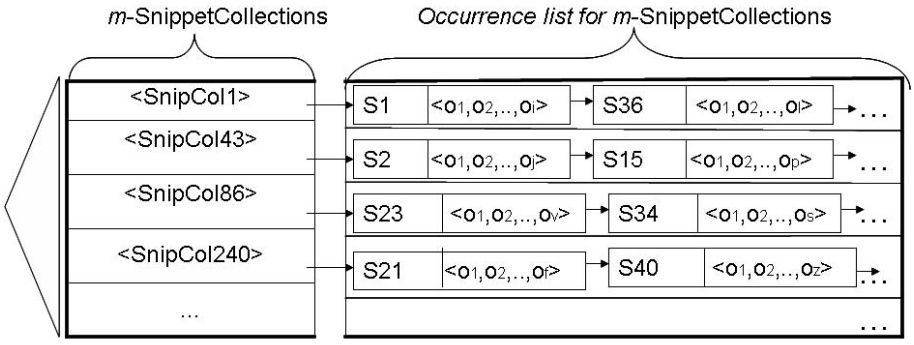

Figure 3.3: m-SnippetCollection Index

We make use of the 2-Level inverted indices for live stream matching. Details of this are explained in the approach section (Chapter 4.3).

Chapter 4

Approach

In this section we describe our matching framework in detail. We also provide explanations of the key heuristics used in the algorithm. We begin with a quick overview of the approach. Then we explain our framework for live stream matching that uses the two level indices.

4.1 Overview

We propose the *n-Snippet Indices Framework in 2-Levels* (in short SNIF Tool). The matching of the live stream S_L against the set of patterns S_P is performed in the following two phases:

1. *Off-line Preprocessing Phase*: Each S_P is scanned through once and snippets, as well as collections, are extracted from them. This step is performed simultaneously with index building. We concurrently clean the index by removing approximate duplicates during this index construction process. This helps to reduce the index size, thus enhancing the performance during the live stream matching step.

2. *On-line Live Stream Matching Phase*: As new data values continuously arrive at S_L, live snippets (L_S) are incrementally extracted from it in a way identical to snippet extraction from each S_P. The L_S is then used to probe the front-end index to record the portions of the respective collections found so far. The high ranked collections then probe the back-end index to perform order

checking to output the potential S_P candidates.

Before we discuss each of the two steps of the matching framework in more detail below we define some parameters used in the framework in Table 4.1.

Parameters	Definitions
S_{Col}Threshold	Corresponds to the lower bound on the collections match score. Only a collection having score greater than or equal to this value is used to probe the back-end index.
S_PThreshold	Corresponds to the lower bound on the S_P match score. Only the S_P with a match score greater than or equal to this value is output as candidate matches
Delta-AvgStdev	The tolerance (ϵ) for the range query over a snippet.
AllowedMissingCollections	Used in the back-end index matching step. This is an additional parameter to allow gaps of collections in the order checking step. For now we have set this value to 0 to allow no gaps.

Table 4.1: Parameters used in the Match Framework

4.2 Preprocessing Phase

The index building (shown in Figure 4.1) and the index cleaning steps are performed offline. The rationale behind this plan of having offline steps prior to the live stream matching phase is to have minimum possible computation during the live matching to reduce the response time.

The preprocessing phase consists of two tasks:

1. extracting snippets and collections from each S_P. (refer to Section 3.3 for details)

2. building the 2 levels of indices. (refer to Section 3.4 for details)

In addition to building the indices, we consider the design of auxiliary structures to help the tuning of the indices to provide possibly more efficient indices for lookup to the matching algorithm.

One issue to be addressed offline is the lookup identifier problem. Since the *front-end* index is a hash-based index using the snippet identifier as shown in Figure

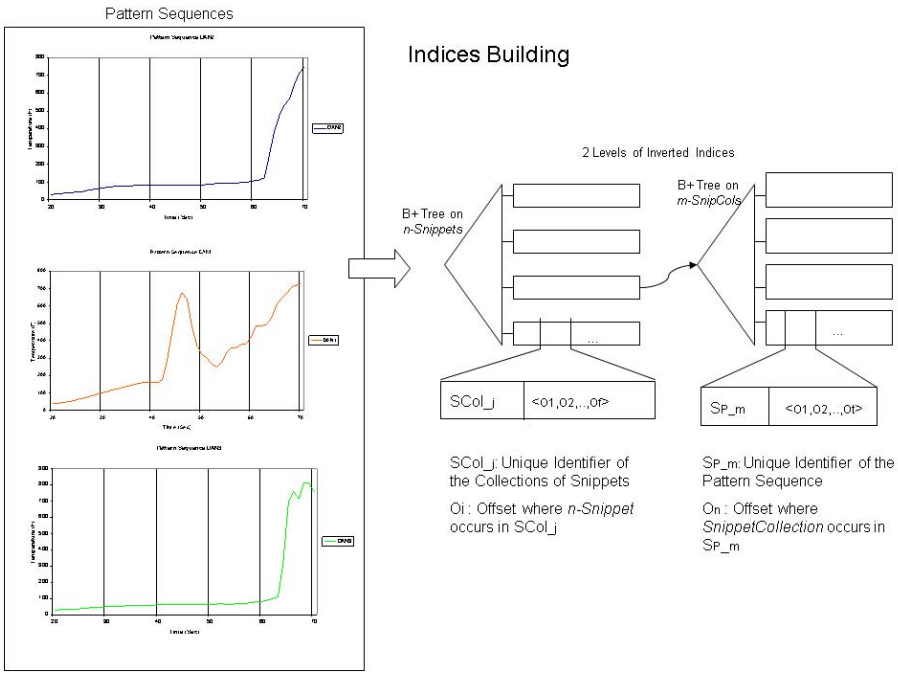

Figure 4.1: Building the Indices

3.2. Hence, the live snippet (L_S) identifier cannot be used directly to lookup that index. We propose to have a tree structure called Avg-StdevSortedTree (ASTree) (illustrated in figure 4.2) of the snippets present in the front-end index which, as the name suggests, is sorted on the average and standard deviation values. This sorted tree can be created as a B-tree. Prior to the front-end index lookup, the live snippet L_S performs a range search on the ASTree to extract similar (in average and standard deviation) snippets (there will possibly be multiple). These snippets, obtained from the ASTree, are then used to lookup the front-end index.

However, while creating the above sorted tree we observed that there are large numbers of similar snippets. This is attributed to the fact that the data values fall within a certain common bandwidth (say, temperature readings ranging from 20 to 900).

Since we perform range search on the sorted tree, there are always multiple snippets extracted, which eventually result in multiple (likely redundant) front-end index looks up. Moreover, this means that the front-end index is loaded with a large number of similar snippets. Hence, there is a potentially great scope for index size reduction. From the point of view of the matching step, such index reduction will significantly boost the efficiency since the search space is reduced. Our proposed so-

Algorithm 1 Algorithm for building the indices

Input:

1. The set of S_P,
2. The length m of collections,
3. The length n of snippets.

Output: The 2-level n-Snippet Indices

1: Extraction of collections:
for each S_P in the set
Suppose that a S_P is a sequence of time-series data values d_0, d_1, d_2, d_N;
where N = Len(S_P).
Extract collections starting from the data value d_i of sizes (m+n-1), where ($0 \leq i \leq \lfloor$((N-n+1)/m $\rfloor$) and record the offsets of the collections within S_P.
If the length of the last collection is less than m, pad the sequence with d_N value to form the last m-SnippetCollection.

2: Construction of the back-end inverted index:
for each m-SnippetCollection obtained in Step 1
Suppose that a collection $SCol_A$ occurs in a pattern sequence S_P at offsets o_0,o_1,...,o_f ;
append an occurrence $\langle$ S_{ID}, o_0,o_1,...,o_f] $\rangle$ to the occurrence list of $SCol_A$.

3: Extraction of snippet:
For each collection say $SCol_B$
Extract snippets starting at the data value d_i, where ($0 \leq i \leq$ L-n) and record the offsets of the snippet within $SCol_B$.

4: Index Cleanup/Clustering of snippets:
Process all the extracted snippets through the clustering algorithm, keeping average and standard deviation values as the dimensions to cluster on.
Obtain the clusters in the form of $\langle$ C_{ID} | set of snippets $\rangle$

5: Construction of the front-end inverted index:
for each Cluster obtained in Step 4
Suppose that a cluster C_E consists of set of snippets [$Snip_A$, $Snip_B$, $Snip_C$, ... $Snip_G$] ;
for each snippet contained in the cluster C_E
Suppose that a snippet $Snip_X$ occurs in a collection $SCol_C$ at offsets o_0, o_1 ,..., o_f ;
Append the occurrence $\langle$ $SCol_C$, [o_0,o_0,...,o_f] $\rangle$ to the occurrence list of C_E.

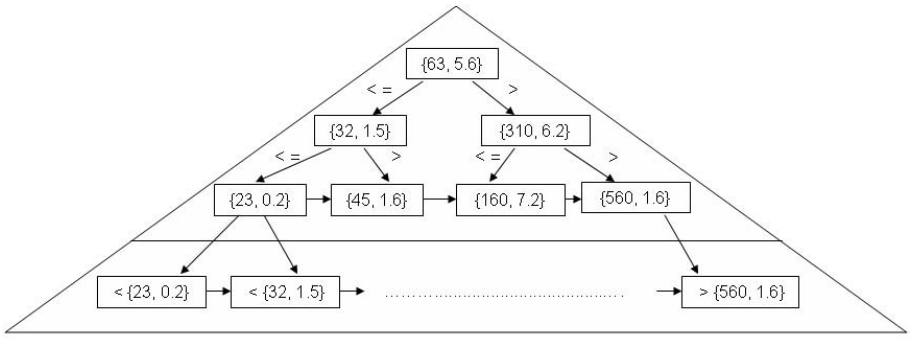

Figure 4.2: Avg Stdev Sorted Tree

lution for index size reduction is to cluster the snippets on the average and standard deviation values using some third party clustering tools [GMM+03]. This clustering task takes place offline, either between or after the initial two preprocessing tasks. We propose slight modifications to the auxiliary ASTree and the front-end index to make this work, as explained below.

Now, after the snippets are extracted and clustered. Each snippet is associated with a cluster identifier (C_{ID}) depending on the range of average and standard deviation values it belongs to. The ASTree now contains the average and standard deviation ranges mapping to the C_{ID} instead of the set of possible snippets. Also the front-end index now contains C_{ID} (which represents all the similar snippets within the range) as the term. Therefore, the occurrence list of each C_{ID} consists of the individual occurrence lists of all the snippets that belong to that cluster. This optimization cuts down the index size and reduces multiple index lookups to a single lookup.

4.3 Live Stream Matching Phase

We propose the following framework for the live stream matching. Using the 2-level indices, the live stream matching is divided into two levels of matching:

1. Snippet Index lookup for *bag matching* of snippets to determine which and

how much of a given collection is matched, and

2. Collection Index lookup for *order checking* of the collections to determine which S_P and how much of it is matched.

These two abstract levels of matching make the matching against S_P of different lengths possible. A S_P of length Len(S) consists of $\lceil$(Len(S)/m+n-1)$\rceil$ collections. We explain each of the above two levels of matching in the following sections.

4.3.1 n-Snippet Index Lookup

As new data is being appended to S_L, live snippets L_S are extracted from S_L (refer to Section 3.3 for details of the snippet extraction process). Each L_S probes the n-Snippet (front-end) index through the ASTree. The n-Snippet index uses the identifiers of snippets present in the S_P (or C_{ID} from the index cleaning step) for indexing (explained in Section 4.2). Therefore, L_S identifiers cannot directly probe the n-Snippet index. As an intermediate step, the average and standard deviation values of the L_S are used to perform a range query (refer to Section 2.1) over the ASTree (illustrated in figure 4.3). The identifier of a snippet or C_{ID} thus obtained by probing the ASTree is used to further probe the n-Snippet index. The list of the collections obtained from probing the index are the potential collections to which the L_S belongs. The matching phase uses several auxiliary structures for recording the matches at the two level. One such structure, that we call *Collections of Latest m* L_S, is used to record each extracted L_S and the list of the collections corresponding to it.

As the memory is limited, one obvious question may be: for how many such L_S do we need to maintain the candidate collections? As the name suggests, we propose to maintain the *Collections of Latest m* L_S for the m current L_S, i.e., equal to the count of snippets in each collection of a S_P. For example, we discard the L_{Si} and its corresponding list of collections as L_{Si+m} is extracted, and so on, for any general i^{th} L_S. This in turn means that we need to store just the latest m+n-1 data points of S_L for our matching technique. For a wide range of m and n values, m+n-1 numbers is certainly a feasible amount of memory. One issue to address here is whether approximately matching a single snippet of the collection of S_P should be considered as the occurrence of the collection? For order checking (defined in Table 3.2) of collections of an S_P, there may be several options of reporting if a collection is

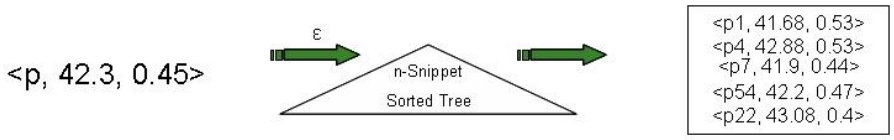

Figure 4.3: Performing a range query: a snippet probing the ASTree

matched. One such option may be to say that the collection has either OCCURRED or NOT-OCCURRED (1 or 0 respectively). But when do we say that? Approximate matching of a single snippet corresponding to the collection is not proof enough of the occurrence of the whole collection since the latter is comprised of m snippets. Requiring all m snippets of the collection to match to report that it has matched is too strict a requirement. In any case, a collection′s occurrence depends directly on the fraction of its snippets in the observed set of the latest m L_S. The set of the latest m L_S may be considered as the live collection ($Coll_{Live}$). We propose to report a collection match when a significant portion of its snippets, exists above the S_{Col}Threshold (defined in Table 4.1) in the $Coll_{Live}$.

$$S_{BM}(Coll_{Live}, \text{Coll of } S_P) = \frac{|\text{Matched n-Snippets between } Coll_{Live} \text{ and Coll of } S_P|}{\max(|\text{n-Snippets in } Coll_{Live}|, |\text{n-Snippets in Coll of } S_P|)} \quad (4.1)$$

Moreover, another question is: what do we report as output of the bag matching (defined in Table 3.2) step to the order checking step? A binary OCCURRED value? Or should we be distinguishing 15 out of 26 versus 26 out of 26 (considering m = 26)? Here we propose to report the fraction from the bag matching (refer to the Formula 4.1) as the score for how much of the collection is matched, where S_{BM} stands for score of bag matching. In our case, each of the Collections of S_P and also

the Coll_{Live} are of size m. To compute the fraction of bag matching, we maintain the frequency count of each collection existing in the list of *Collections of Latest m* L_S across the latest m L_S. We utilize another auxiliary structure called *Frequency Count of Latest m* L_S to record the counts. For each collection appearing in the collection list, the score is the ratio between its frequency count across m L_S and the value m, since finding all m snippets of a collection will be called a complete match.

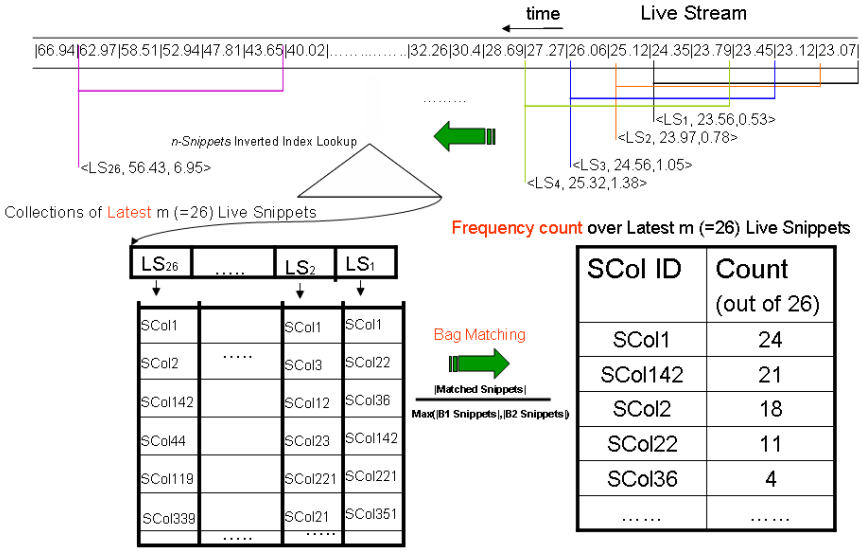

Figure 4.4: Snapshot 1: Bag matching in progress

Next we explain the whole process of bag matching as illustrated in the two Figures 4.4 and 4.5. Say, for our example, m = 26 and n = 5. As each live snippet L_S is extracted from the live stream S_L, the corresponding collections found by probing the front-end index are stored in the *Collections of Latest m* L_S. When we have m L_S we can perform the frequency count and store it in the *Frequency Count of Latest m* L_S. The two figures show bag matching steps for live snippets 1 to 26 and 2 to 27. As we do transition from the Figure 4.4 to the Figure 4.5, when L_{S27} arrives, L_{S1} gets eliminated. The frequency counts for each collection can be incrementally computed just by taking into account the outgoing L_S and the incoming L_S. The *Frequency Count of Latest m* L_S can be used to transfer the bag matching score to the order checking step as shown in the next section.

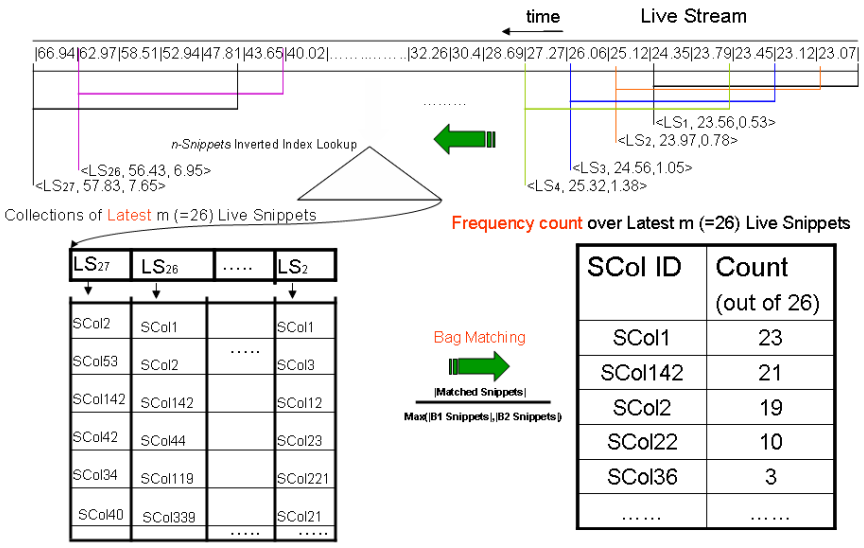

Figure 4.5: Snapshot 2: Bag matching in progress

4.3.2 m-SnippetCollection Index Lookup

From the bag matching step the candidate collections along with their count scores are obtained in *Frequency Count of Latest m* L_S. For the *order checking* step (defined in Table 3.2), the back-end index is probed (defined in Table 3.2) by the collection identifiers to fetch their corresponding occurrence lists (defined in Table 3.2). However, out of the collections listed in *Frequency Count of Latest m* L_S only the ones with count scores above the S_{Col}Threshold (defined in Table 4.1) are used to probe the back-end index. By probing the back-end index the candidate S_P are obtained from the occurrence lists. We introduce another auxiliary structure: S_P *match node* that we use in this step. A S_P *match node* looks like $\langle \rho, \nu[1{:}\rho], \phi \rangle$ comprising of 3 components:

1. *Match Position* ρ - The position of the current collection up to which the S_P has been matched.

2. *Match Vector* ν*[1:*ρ*]* - A vector recording the bag matching scores of the collections in the order they occur in the original S_P.

3. *Match Score* S_{OC} - Cummulative score for the Match Vector ν as a function of the scores of the individual collections averaged up to the Match Position ρ (as given in the Formula 4.2), where S_{OC} stands for score of order checking.

$$S_{OC}(S_P,\rho) = \frac{\sum_{k=1}^{\rho} C_k \times \nu[k]}{\sum_{k=1}^{\rho} C_k} \tag{4.2}$$

C_k stands for a weight for each matched position k. For our score computation we kept $C_k = 1$ for all k positions. However, other C_k values could be used for weighted averages. For example, to give matched S_P scores according to the length of match, one could keep $C_k = k$. Thus, weighing increases with the increase in the match position. The exploration of exponential or linear weights for ordering is kept as future work. *S_P Match Nodes* are maintained for each candidate S_P.

Figure 4.6 illustrates the process of order checking using the back-end Index. Those collections in the *Frequency Count of Latest m L_S*, that are marked with 'green' are the high ranked collections ($S_{BM} \geq S_{Col}$Threshold) and are used to probe the back-end index. The 'red' ones have low scores. From the S_P and position information of the occurrence list the match for each candidate S_P can be recorded in the S_P match nodes. As shown in Figure 4.6, similarly the S_P match nodes are 'green' if their order checking score S_{OC} meet the S_PThreshold (defined in Table 4.1), otherwise they are marked in 'red'. Only a high ranked S_P will be reported as a candidate match.

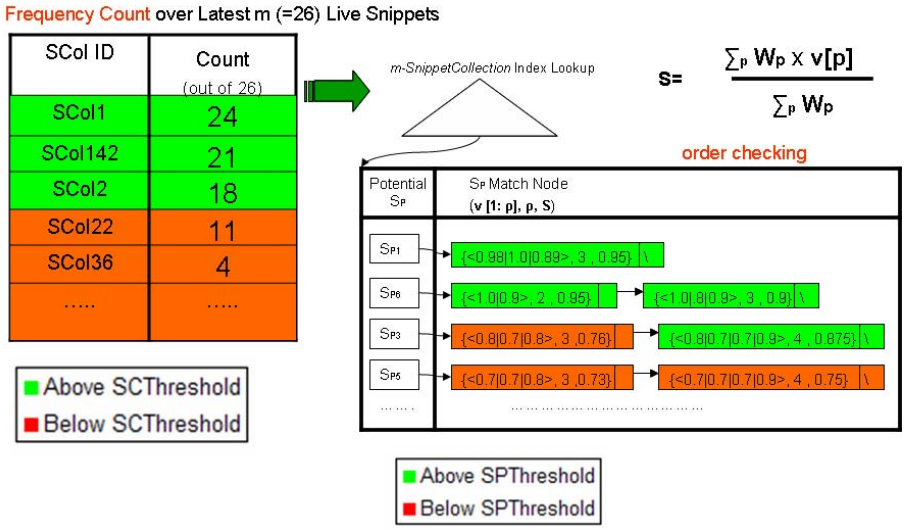

Figure 4.6: Order Checking using Collection Index Lookup

An example of a S_P match node is $\langle$ 3,$\langle$0.98|1.0|0.89$\rangle$, 0.95 $\rangle$. Here 3 denotes the match position ρ. The $\langle$0.98|1.0|0.89$\rangle$ is the ν[1:3], denoting the scores of the first 3 consecutive collections for the S_P, and 0.95 is the S_{OC} computed according

to Formula 4.2. The value of ρ for a S_P can vary from 0 to $\lfloor \text{Len}(S_P) \div (m+n-1) \rfloor$, i.e., the number of the collections into which the S_P is divided in the preprocessing step.

The individual collection score is the result of bag matching between the collection and the latest m L_S. Here the order of occurrence is not a concern, hence it is a bag matching. However, at the next level, while reporting a match for a S_P, the order of the collections is checked. A score for a collection is added to a $\nu[1{:}\rho]$ of a S_P only if that collection is at a position $p \leq (\rho + 1)$ and with a score above the S_{Col}Threshold. These S_P match nodes are also incrementally evaluated and maintained just like the Frequency Count of Latest m L_S.

Next we discuss some issues related to incremental evaluation of the S_P Match Nodes. Firstly, while several S_P matches can be formed, there needs to be a mechanism for discarding the matches that become less promising over time. We propose to discard S_P matches where the match vector remains unchanged for more than Latest m L_S as we do not allow gaps or noise of more than m L_S worth of live data. This conforms to our goal of prefix matching. We could relax this restriction and allow gaps between matched collections. This can be achieved by using the parameter *AllowedMissingCollections* (defined in Table 4.1). For now the parameter AllowedMissingCollections is set to zero.

Secondly, many collections, especially adjacent collections within a S_P, can share similar snippets. For this reason multiple collections within a S_P may have high scores in the latest m L_S. There are several options how best to address this. One may maintain just a single $\nu[1{:}\rho]$ for a S_P based on either the best match score (the S_{OC} value) or the extent of the match (the ρ value). Alternatively, multiple match vectors can be maintained for that S_P. We found that maintaining just the single $\nu[1{:}\rho]$ for a candidate S_P is very efficient and works well for our three experimental datasets. However, clearly this is a heuristic and, in general, multiple $\nu[1{:}\rho]$ allows any one of those to become the best choice later. There is a trade off between response time and result accuracy; hence there needs to be an upper bound on the maximum number of match vectors to be maintained per S_P.

Another design decision to make is that, due to multiple match nodes (ν) per S_P, there can be a case where two or more nodes ν for a S_P may be at the same position ρ. For example, for a S_{Pi} let $\nu_1[1{:}3] = \langle\ 1.0|0.7|0.9\ \rangle$ and $\nu_2[1{:}3] = \langle 1.0|1.0|0.9 \rangle$ be two match vectors both matched up to $\rho = 3$. Here $S_{OC1} = 0.86$ and $S_{OC2} = 0.93$

respectively. Since ν_2[1:3] has a perfect match (1.0) in the 1^{st} collection position as well as in the 2^{nd} collection position, the match is as good as it can get with respect to the first two positions. Hence, ν_1[1:3] has the lower score and can be discarded. The reason being, any change to position 2, will not make it better than ν_2[1:3]. Both the νs can get effected if a S_{BM} is reported for either the position 4 collection or at position 3 itself. In both the cases the ν_2[1:3] will always remain the one with the higher score. Hence in such a situation ν_1[1:3] can be safely discarded. Hence, maintaining multiple S_P Match Nodes (ν) per S_P but, within them, just a single per match position ρ of collection, forms another of our heuristics for maintaining selective multiple match nodes ν per S_P.

We propose to have four variations of the live stream matching according to the number of match vectors maintained per S_P. The variations allow the user the capability to choose between the two conflicting characteristics of result accuracy versus response time. The variations are:

1. *Best 1*- Only a single match vector is maintained per S_P based on the match score.

2. *Multiple 1 per position*- Multiple matches for a S_P but only 1 per position of collection in the S_P

3. *Best k*- The Top k match vectors are maintained per S_P based on the match score.

4. *Best k with 1 per position*- Multiple match nodes maintained as a combination of Best k and Multiple 1 per position.

We present the complete algorithm for the live stream matching step in 2. In the next section we discuss the performance evaluation experiments done using our approach.

Algorithm 2 Live Stream Matching Algorithm

Input:

1. The 2-level indices,
2. The continuous live stream sequence S_L.

Output: Potential pattern sequences S_P and S_{OC}.

1: Range search on ASTree.
As data values continuously arrive at the live stream;
Form snippets L_S using 1-Sliding technique;
Suppose L_S is of the form $\langle$ L_{Sid}, Avg, Stdev $\rangle$
For each L_S extracted
Perform range search on the ASTree using the average & stdev value pair of L_S,
Extract the single C_{ID} corresponding to the range (average-stdev) of L_S.

2: Front-end index lookup.
For each C_{ID} extracted in Step 1,
Look up the front-index and collect the list of collections corresponding to the C_{ID}.
Report the L_S and the corresponding list of collections to *Collections of Latest m L_S*.

3: Incrementally maintain Latest m L_S in *Collections of Latest m L_S*.
If the number of L_S reported in *Collections of Latest m L_S* exceeds m
Remove the earliest reported L_S entry.
Note the collections and their frequency count over the latest m L_S in *Frequency Count of Latest m L_S*

4: Incrementally Maintain *Frequency Count of Latest m L_S*
Suppose *Collections of Latest m L_S* is of the form:
$\langle$ L_{S1}, (List of collections) $\rangle$, $\langle$ L_{S2}, (List of collections) $\rangle$, , so on till m consecutive L_S.
For each collection reported in *Collections of Latest m L_S*
Report the collections count across the latest m L_S into *Frequency Count of Latest m L_S* as $\langle$ $SCol_{ID}$, Count $\rangle$.
For each collection,
part of the removed (earliest) L_S in Step 3,
Reduce the corresponding count of the collection in *Frequency Count of Latest m L_S*.

5: Back-end index lookup.
From the *Frequency Count of Latest m L_S*,
select the collections whose S_{BM} exceeds the S_{Col}Threshold.
For each such high ranked collection,
Lookup the back-end index
Collect the occurrence lists of the form $\langle$ S_{P1}, offsets $\rangle$, $\langle$ S_{P2}, offsets $\rangle$ ... $\langle$ S_{Pi}, offsets $\rangle$

6: Incrementally Maintain the S_P Match Nodes $\nu[1{:}\rho]$ as per the chosen heuristics.
From the occurrence lists of collection, $\nu[1{:}\rho]$ can be incrementally computed and maintained.
Report S_P as a candidate only if S_{OC} exceeds S_PThreshold

Chapter 5

Experimental Evaluation

In this section we study the performance and accuracy of our live stream matching framework using an experimental study. The effectiveness of SNIF to detect patterns has been thoroughly tested through extensive experimental evaluations. We use the continuous query engine called CAPE [RDS+] as our platform. Experiments were performed on a dedicated laptop computer (Dell Inspiron 600m with 2GB RAM and Intel(R) Pentium(R) M processor 1.70 GHz).

5.1 Experimental Setup

5.1.1 Datasets

We used 3 different real datasets to perform our experiments, namely, the EDaFS fire dataset [WVM04], the sensor network motes dataset [SPF] and the chlorine monitoring dataset [EPA]. Each of these datasets consists of 40∼50 different sequences.

As we can see from the plot of sample sequences from each of the datasets (refer to the Figures 5.1, 5.2,5.3), the datasets are from three different domains and are significantly distinct from each other. The EDaFS dataset (see Figure 5.1) contains temperature, smoke, CO readings recorded during several fire tests. Specifically, the temperature ranges vary from room temperature up to 700∼900 F depending on the fire type. There are mainly data pertaining to two fire types, namely, the smouldering fire and the flaming fire.

The Motes dataset (see Figure 5.2) consists of 4 groups of sensor measurements

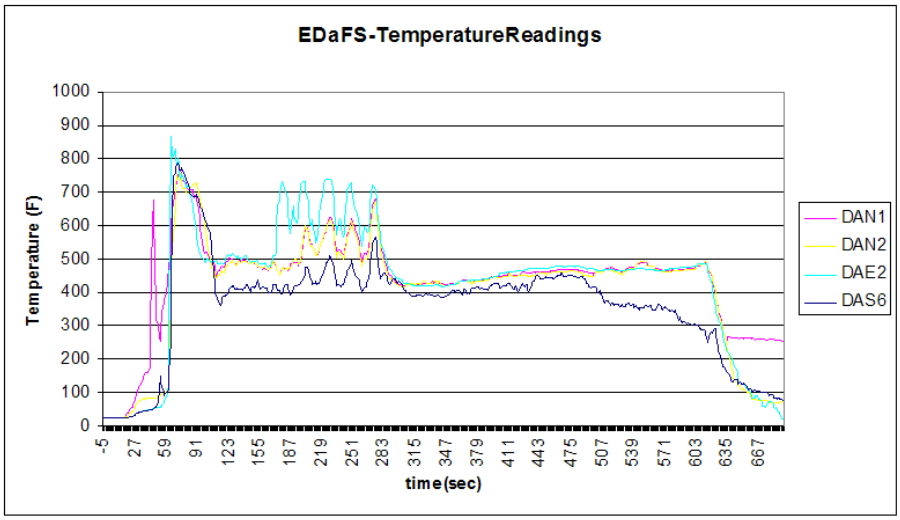

Figure 5.1: Sample sequences from the EDaFS dataset

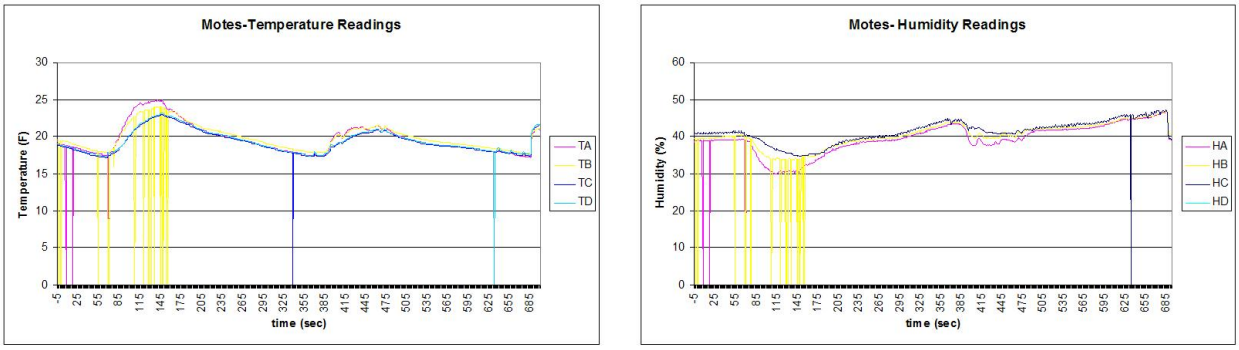

Figure 5.2: Sample sequences from the Motes dataset 1) Temperature 2) Humidity

(i.e., light intensity, humidity, temperature, battery voltages) collected using 48 Berkeley Mote sensors at different locations in a lab, over a period of a month. This is an example of heterogeneous streams. Temperature shows a weak daily cycle and a lot of bursts. Humidity does not have any regular pattern.

The Chlorine dataset (see figure 5.3) was generated by EPANET [EPA] that accurately simulates the hydraulic and chemical phenomena within drinking water distribution systems. There are two characteristic features of the data. A clear global periodic pattern (daily cycle, dominating residential demand pattern) and a slight time shift across different junctions. Thus, most streams exhibit the same sinusoidal-like pattern, except with gradual phase shifts as we go further away from

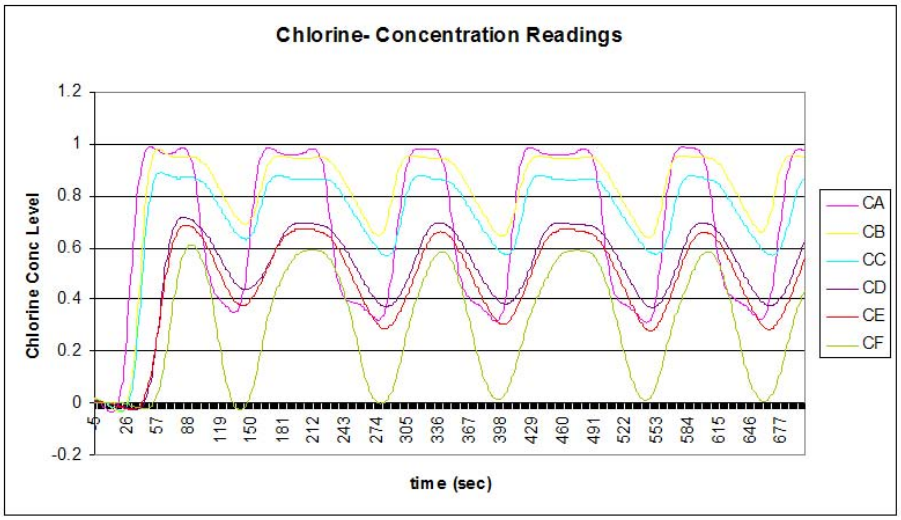

Figure 5.3: Sample sequences from the Chlorine dataset

the reservoir.

5.1.2 Forming Pattern Sequences and Live Streams

The pattern sequences are extracted from the data sequences present in the datasets. Specifically, each of the sequences are trimmed to eliminate the regular or common portions that do not include any phenomenon. Overall, we extract sequences of length ranging between 200 datapoints and 900 datapoints for different set of experiments. The index building code scans through each data sequence and loads it into two indices. The auxiliary tree structure ASTree is also created alongside.

The live streams are generated from the sequences of the dataset as well. To form long (seemingly infinite) live streams, a sequence is repetitively appended to itself several times.

5.1.3 Experimental Plan

Our experiments are designed to evaluate the following two factors:

1. *Performance*- We compare the CPU costs for the four proposed variations of the matching algorithm.

2. *Robustness*- We examine the change in the match score S_{OC} of a live stream as we increase the noise level in it.

We also compute the robustness and accuracy of the matching technique by introducing different amounts of noise (missing data values by removing values from the sequences) in the live sequences. Specifically, we vary the amount of missing data from 0% up to 20%.

Table 5.1 lists the different parameters of the system with their variations or their constant settings in the different experiments.

Parameters	Range of values
S_{Col}Threshold	Varies between 0.5 and 0.8
S_PThreshold	Varies between 0.5 and 0.8
DeltaAvgStdev	Varies between 0.1 and 1.0.
AllowedMissingCollections	For now it is set 0 to allow no collection gaps.
$_P$ Lengths	Three sets of S_P with lengths ranging 200∼300, 500∼600 and 800∼900 are maintained as S_P(s)
Missing Data %	S_L(s) are formed by varying the missing data % between 5% and 20%.

Table 5.1: Parameters varied during Experiments on Live Stream Matching

5.2 Performance Evaluation

We evaluate the performance of the following four variations of the algorithm (discussed in Section 4.3.2) based on the SNIF framework:

1. Best 1 Match Node (ν) per S_P
2. Multiple ν but 1 per matching position (ρ) per S_P
3. Best k ν per S_P
4. Best k ν per S_P with just 1 per ρ per S_P.

For the performance evaluation, the total CPU time for matching, and the average CPU time per processing cycle are the performance measurement on the y-axis. The variables are the desired S_{Col}Threshold, the desired S_PThreshold and the index size determined by the length of sequences stored.

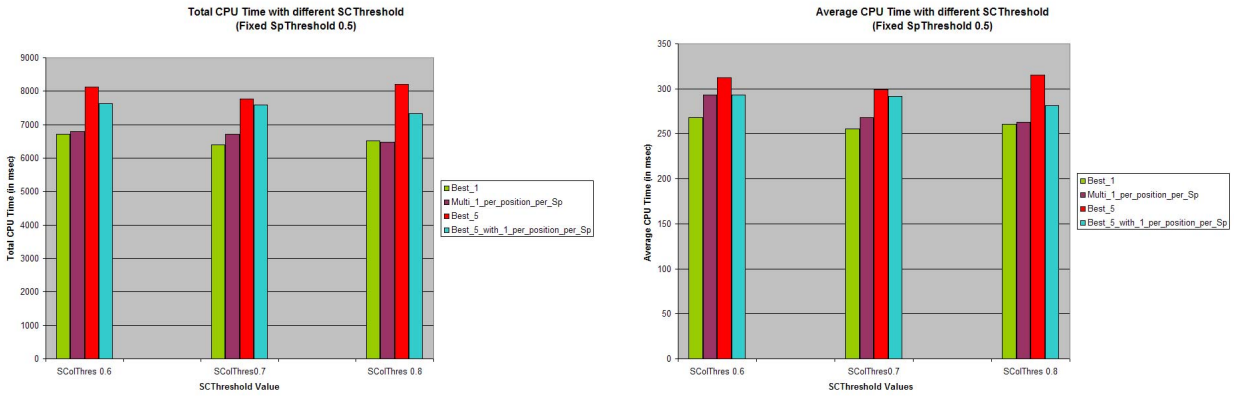

Figure 5.4: CPU costs for different S_{Col}Threshold and fixed S_PThreshold: 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle.

For the experiments 1, 2 and 3, we use the EDaFS fire test dataset.

Experiment 1: We vary the desired S_{Col}Threshold keeping the S_PThreshold fixed (refer to Figure 5.4).

Experiment 2: Similar to experiment 1, we also study the processing time by using two distinct S_PThreshold values while keeping a fixed value for the S_{Col}Threshold (as shown in Figure 5.5)

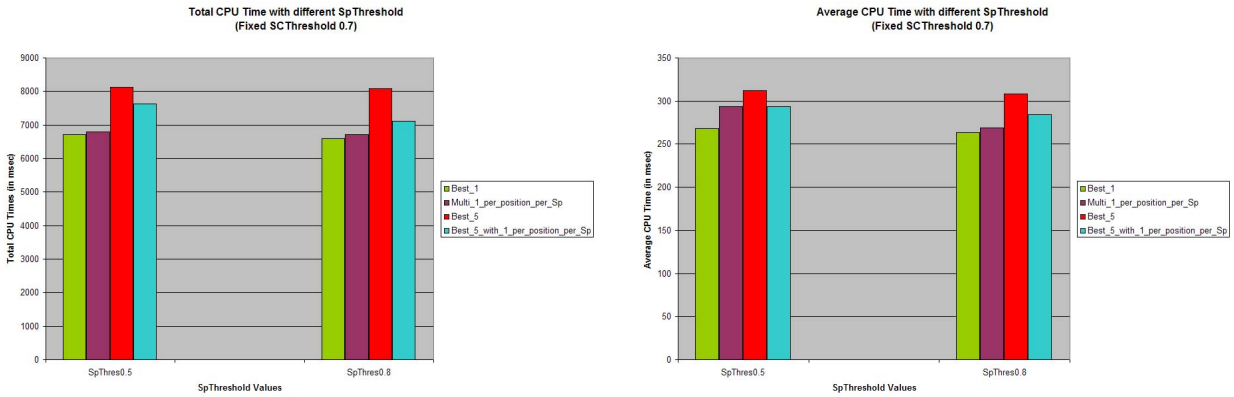

Figure 5.5: CPU costs for different S_PThreshold and fixed S_{Col}Threshold: 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle.

Observations: In both experiments 1 and 2, we observe that the CPU cost tends to be increasing in the following order: Best 1 < Multiple but 1 per ρ per S_P < Best k (=5) with 1 per ρ per S_P < Best k (=5). However, there is not much difference in the processing costs of the four variations except for about 100∼200 milliseconds. This is attributed to two factors: a) the index lookup is quick, b) the search quickly narrows down to very small set of candidate S_P and their corresponding match nodes

ν (refer to Figure 5.6). Hence, the CPU costs are associated to maintaining and processing very few S_P and νs.

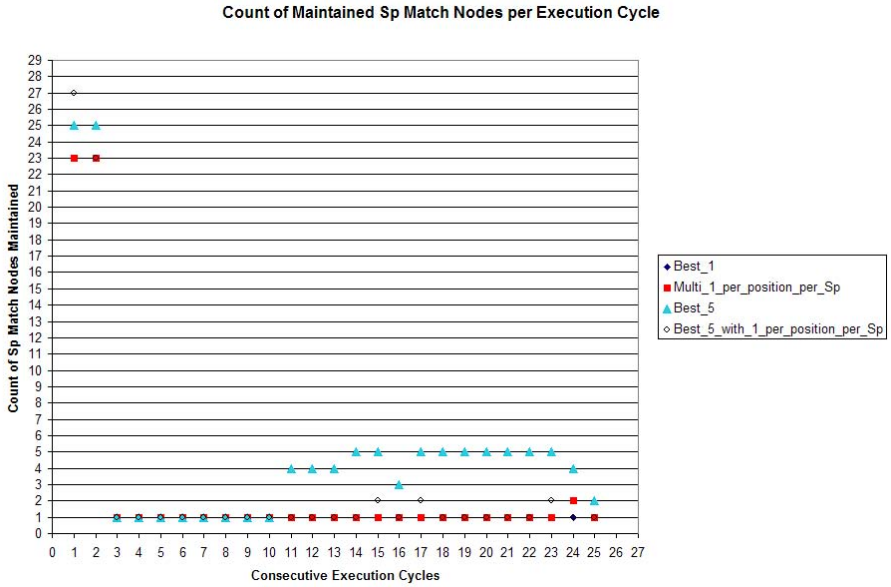

Figure 5.6: Count of Match Nodes (νs) maintained through the runs of the four algorithms

Experiment 3: Further, we evaluate the effect of the size of the index on the processing cost of the matching algorithms. The longer the patterns and the larger the number of patterns, the larger the index size becomes. This will eventually effect the processing cost. We chose to extract three sets of pattern series having length ranges of 200∼300, 500∼600, and 800∼900, respectively.

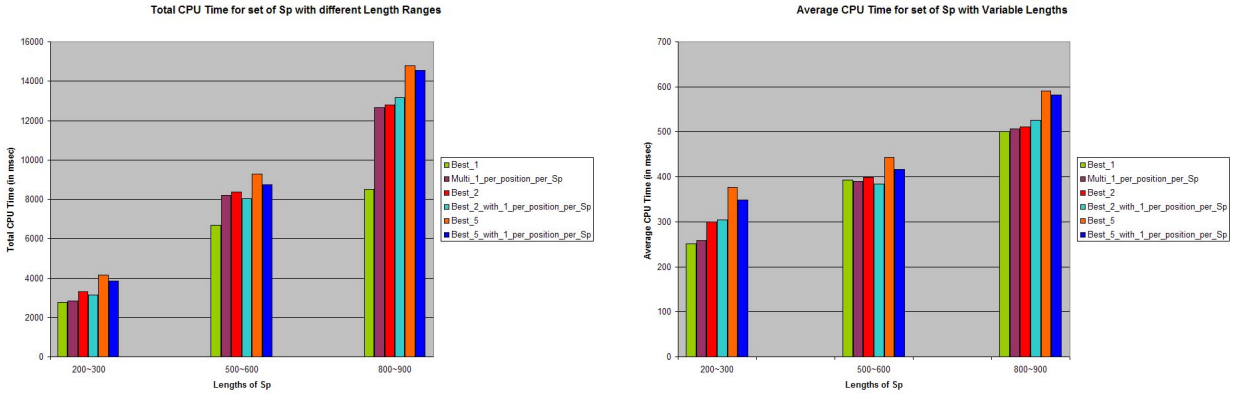

Figure 5.7: CPU costs for different pattern sizes 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle.

In Figure 5.7, we compare the total and average CPU costs over sets of patterns having different lengths. We examine 6 variations of the algorithm, namely, Best 1, Multiple but 1 per ρ per S_P, Best 2, Best 2 with 1 per ρ per S_P, Best 5, and Best 5 with 1 per ρ per S_P.

Observations: We observe that the longer the pattern lengths, the larger the index size and the higher the CPU costs become. Also the trend of increasing CPU times is similar to experiments 1 and 2, i.e., Best 1 $<$ Multiple but 1 per ρ per S_P $<$ Best 5 with 1 per ρ per S_P $<$ Best 5. However, for Best 2 and Best 2 with 1 per ρ per S_P, we observe that their CPU costs are quite similar to Best 1 and Multiple but 1 per ρ per S_P. Also for pattern lengths 500$\sim$600, Best 2 $<$ Best 2 with 1 per ρ per S_P. We find that for both the Best 2 variations, the number of νs maintained per S_P is not much different from those in Best 1, hence the similar CPU costs. Moreover, after maintaining the Best 2 νs per S_P, the check for 1 ν per ρ per S_P is consuming some unnecessary CPU as it does not have much scope for reducing the number of νs.

Experiment 4: Here we evaluate the performance of the Chlorine dataset based on the total CPU costs and the average CPU costs. We vary the desired S_{Col}Threshold keeping the S_PThreshold fixed (refer to Figure 5.8)

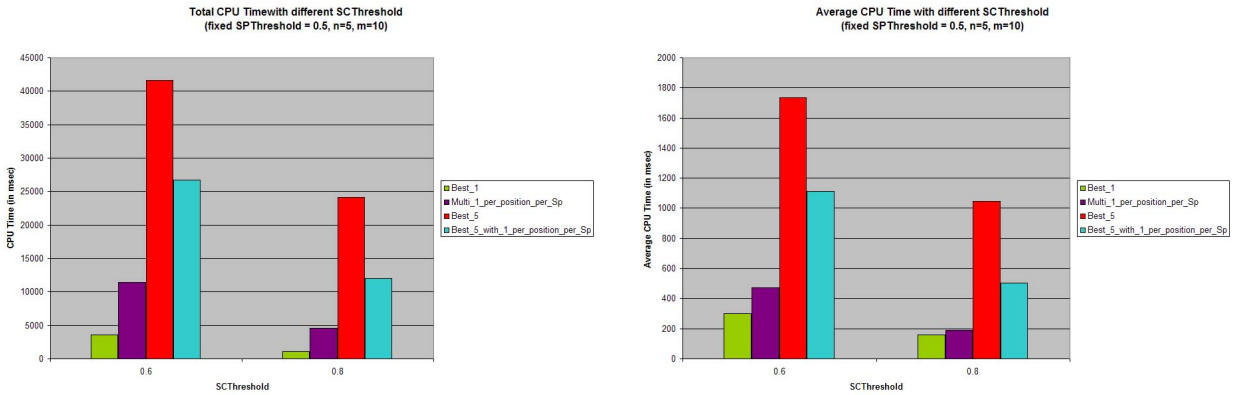

Figure 5.8: CPU costs for different S_{Col}Threshold and fixed S_PThreshold: 1⟩ Total CPU costs 2⟩ Average CPU costs per processing cycle.

Observations: The chlorine dataset has many similar patterns and those vary only within a very small data range (-0.2 to +1.0). Hence several candidate S_P and multiple νs per S_P are being maintained. The CPU cost tends to be in the following increasing order: Best 1 $<$ Multiple but 1 per ρ per S_P $<$ Best k (=5) with 1 per ρ per S_P $<$ Best k (=5). Moreover, the CPU costs for the four variations are distinctly different. This is attributed to the small data range of sequences and

the cyclic nature of the patterns. The number of νs maintained is much larger in variations maintaining multiple νs and the CPU cost is highly dependent on the number of νs maintained. We also observe that by increasing the S_{Col}Threshold, the CPU costs are decreased as less νs have scores higher than S_{Col}Threshold 0.8 as compared to S_{Col}Threshold 0.6.

5.3 Robustness

The robustness criteria of the match algorithms is the effect of the noise levels (missing data values) in the live stream on the match scores. Experiment 1 is on the EDaFS dataset whereas experiments 2 and 3 are on the Motes and the Chlorine datasets respectively.

Experiment 1: We examine various live sequences and the change in their scores as we increase their noise level. For the results refer to Figure 5.9. We show 6 of the live sequences in the figure, namely, DAN2, DAN4, DAE1, DAS6, DFN4 and DFF11.

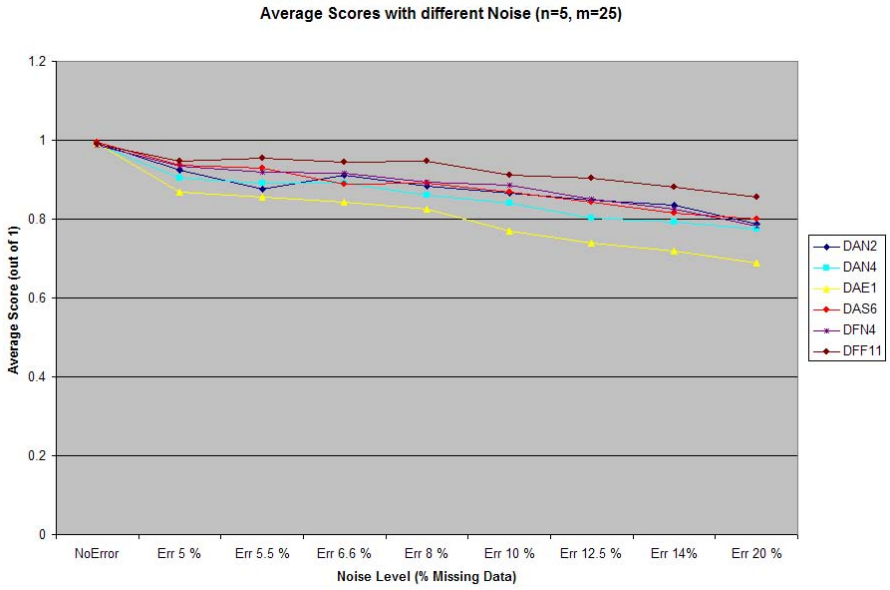

Figure 5.9: EDaFS dataset- Average scores of different live sequences with increasing noise.

Observations: Due to the high amount of smoothing (snippet size n =5) and the large margin of disorder of snippets allowed within each collection (collection size

= 25) the scores of the live sequences decrease gradually, when they are matched against the library of pattern sequences.

Experiment 2: We examine various live sequences of the Motes dataset. We observe the change in their scores as we increase their noise level. Here we also examine how varying the snippet size (n) and the collection size (m) will affect the scores. Snippet size is the degree of smoothing of snippets and collection size determines the allowed disorder amongst the snippets within a collection for bag matching. In this experiment we vary the snippet size keeping the collection size same (refer to Figure 5.10). We show 5 of the live sequences in the figure, namely, TA, TB, TC, TD, TE.

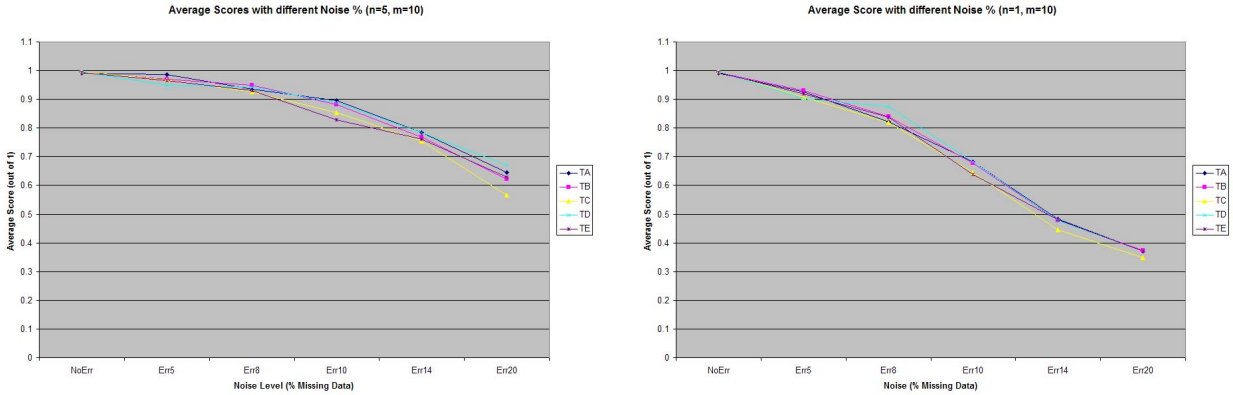

Figure 5.10: Motes dataset- Average scores of different live sequences with increasing noise. a) n = 5 and m = 10 and b) n = 1 and m = 10

Observations: In the two graphs of Figure 5.10, we compare the change in average scores of 5.10 a) n = 5 and m =10 (smoothed snippets with adequate collection size) against that of 5.10 b) n = 1 and m =10 (snippets with no smoothing but with adequate collection size). We observe that the average scores of the live sequences in case b decrease much more rapidly as we increase the noise level as compared to case a. The unsmoothed snippets (in case b) versus the smoothed snippets (in case a) cause this difference. However, within case b we find that the adequate collection size allowing disorder amongst the snippets prevents the scores to fall rapidly up to 8 percent missing data despite unsmoothed snippets.

Experiment 3: We examine various live sequences of the Chlorine dataset. As in experiment 2 here also we observe the change in the scores as we increase the noise level in the live sequences. In this experiment we vary the collection size keeping the snippet size the constant. Refer to Figure 5.11 for results. We show 5 of the

live sequences in the figure, namely, CA, CB, CC, CD, CE.

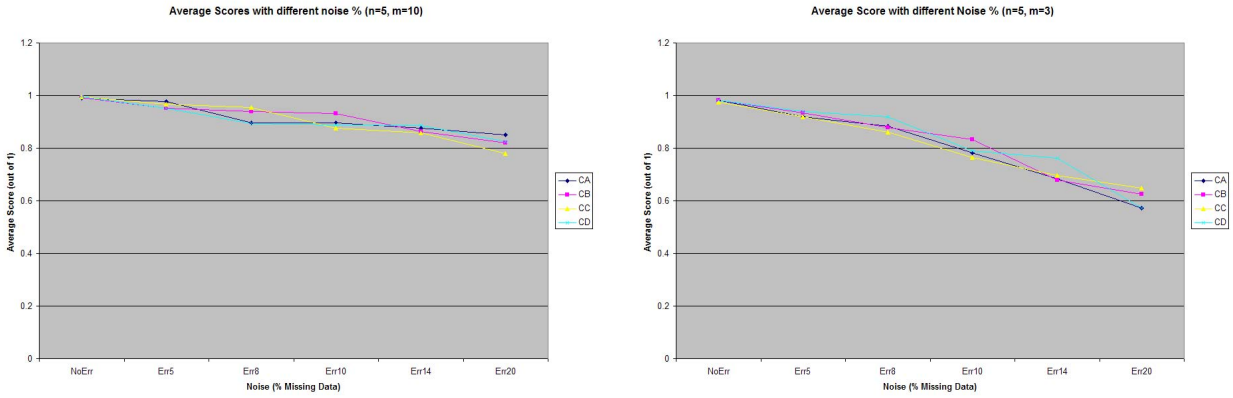

Figure 5.11: Chlorine dataset- Average scores of different live sequences with increasing noise. a) n = 5 and m = 10 and b) n = 5 and m = 3

Observations: In the two graphs of Figure 5.11, we compare the change in average scores of 5.11 a) n = 5 and m = 10 (smoothed snippets with adequate collection size) and 5.11 b) n = 5 and m = 3 (smoothed snippets with small collection size). The small data range of the sequences results in gradual decrease of the average scores when more noise is introduced in the live sequences. However, the adequate collection size (allowed disorder) makes the match algorithm more robust than the small collection size.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we propose a generic framework for sequence matching over streaming data. We call it the n-Snippet Indices Framework (in short, SNIF). We introduce the concepts of snippets and collections for numeric data. We also propose to apply two abstract levels of matching, namely, bag matching and order checking.

The framework addresses challenges of the streaming environment, namely, noise elimination, incremental evaluation, and efficient CPU utilization. Our framework stores very small portions of the live stream S_L: i.e., it maintains the live data up to just the latest m snippets. More precisely, maintaining m snippets means maintaining just the latest m+n-1 data values.

Experiments demonstrate the efficiency and effectiveness of the SNIF Tool for sets of patterns having different lengths (300, 600, 900). We show that the framework is capable of real-time response. We also demonstrate how the framework is tolerant to the different noise levels in the live streams ranging up to 20 percent of missing data.

6.2 Future Work

During this thesis work we have addressed several challenges. However, many additional research challenges have been identified for possible future work. A few of them are listed below:

1. Evaluation of the performance and robustness of the framework using other

similarity measures such as Dynamic Time Warping, Discrete Wavelet Transform and Fast Fourier Transform.

2. Use of *disjoint* snippets rather than using 1-Sliding to extract snippets from a sequence. There are two aspects to be explored namely, a) disjoint pattern snippets against sliding live snippets, and b) sliding pattern snippets against disjoint live snippets.

3. Monitor several live streams from adjacent / related sensors and predict some phenomenon based on the correlation of the matches of those live streams. This is the next abstract level of a monitoring system which can obtain results from the matching framework and make decisions based on observations from several streams.

4. Support for multiple continuous similarity queries is another aspect to explore. The focus would be to enhance the performance of the framework to be able to handle several similarity queries while having limited resources.

Bibliography

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.

[ALSS95] Rakesh Agrawal, King-Ip Lin, Harpreet S. Sawhney, and Kyuseok Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *VLDB*, pages 490–501, 1995.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[BS03] Stephen D. Bay and Mark Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD*, pages 29–38, 2003.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. Modern information retrieval. *ACM Press*, 1999.

[CFY03] Kin-Pong Chan, Ada Wai-Chee Fu, and Clement T. Yu. Haar wavelets for efficient similarity search of time-series: With and without time warping. *IEEE Trans. Knowl. Data Eng.*, 15(3):686–705, 2003.

[CN04] Lei Chen and Raymond T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[Coh97] Jonathan D. Cohen. Recursive hashing functions for n-grams. *ACM Trans. Inf. Syst.*, 15(3):291–320, 1997.

[EPA] www.epa.gov.ord.nrmrl.wswrd.epanet.html.

[FRM94] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings 1994 ACM SIGMOD Conference, Mineapolis, MN*, pages 419–429, 1994.

[GMM+03] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng.*, 15(3):515–528, 2003.

[GW02] Like Gao and Xiaoyang Sean Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *SIGMOD Conference*, pages 370–381, 2002.

[GYW02] Like Gao, Zhengrong Yao, and Xiaoyang Sean Wang. Evaluating continuous nearest neighbor queries for streaming time series via pre-fetching. In *CIKM*, pages 485–492, 2002.

[HLMJ07] Wook-Shin Han, Jinsoo Lee, Yang-Sae Moon, and Haifeng Jiang. Ranked subsequence matching in time-series databases. In *VLDB*, pages 423–434, 2007.

[HS95] Gísli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *SSD*, pages 83–95, 1995.

[KCMP01] Eamonn J. Keogh, Kaushik Chakrabarti, Sharad Mehrotra, and Michael J. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD Conference*, 2001.

[KP99] Eamonn J. Keogh and Michael J. Pazzani. Scaling up dynamic time warping to massive dataset. In *PKDD*, pages 1–11, 1999.

[KPL04] Sang-Wook Kim, Dae-Hyun Park, and Heon-Gil Lee. Efficient processing of subsequence matching with the euclidean metric in time-series databases. *Inf. Process. Lett.*, 90(5):253–260, 2004.

[KPM07] Maria Kontaki, Apostolos N. Papadopoulos, and Yannis Manolopoulos. Adaptive similarity search in streaming time series with sliding windows. *Data Knowl. Eng.*, 63(2):478–502, 2007.

[KWLL05] Min-Soo Kim, Kyu-Young Whang, Jae-Gil Lee, and Min-Jae Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[LA96] Joon Ho Lee and Jeong Soo Ahn. Using n-grams for korean text retrieval. In *SIGIR*, pages 216–224, 1996.

[LPK06] Seung-Hwan Lim, Hee-Jin Park, and Sang-Wook Kim. Using multiple indexes for efficient subsequence matching in time-series databases. In *DASFAA*, pages 65–79, 2006.

[MSLN00] Ethan Miller, Dan Shen, Junli Liu, and Charles Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *Journal of Digital Information*, 2000.

[MWH02] Yang-Sae Moon, Kyu-Young Whang, and Wook-Shin Han. General match: a subsequence matching method in time-series databases based on generalized windows. In *SIGMOD Conference*, pages 382–393, 2002.

[MWL01] Yang-Sae Moon, Kyu-Young Whang, and Woong-Kee Loh. Duality-based subsequence matching in time-series databases. In *ICDE*, pages 263–272, 2001.

[RDS+] Elke A. Rundensteiner, Luping Ding, Timothy Sutherland, Yali Zhu, Brad Pielech, and Nishant Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.

[SPF] Jimeng Sun, Spiros Papadimitriou, and Christos Faloutsos. Distributed pattern discovery in multiple streams. note.

[WMB99] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 1999.

[WSS+05] Huanmei Wu, Betty Salzberg, Gregory C Sharp, Steve B Jiang, Hiroki Shirato, and David Kaeli. Subsequence matching on structured time

series data. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 682–693, New York, NY, USA, 2005. ACM.

[WSZ04] Huanmei Wu, Betty Salzberg, and Donghui Zhang. Online event-driven subsequence matching over financial data streams. In *SIGMOD Conference*, pages 23–34, 2004.

[WVM04] J. W. Woycheese, R. Venkatesh, and K. Mihyun. Experiment database for fire science, database architecture 0.9, August 2004.

[WW00] Changzhou Wang and Xiaoyang Sean Wang. Supporting content-based searches on time series via approximation. In *SSDBM*, pages 69–81, 2000.

[WW03] Teddy Siu Fung Wong and Man Hon Wong. Efficient subsequence matching for sequences databases under time warping. *ideas*, 00:139, 2003.