# SNIF TOOL: Sniffing for Patterns in Continuous Streams*

Abhishek Mukherji, Elke A. Rundensteiner, David C. Brown, Venkatesh Raghavan
Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA, USA.
{mukherab | rundenst | dcb | venky }@cs.wpi.edu

## ABSTRACT

Continuous time-series sequence matching, specifically, matching a numeric live stream against a set of predefined pattern sequences, is critical for domains ranging from fire spread tracking to network traffic monitoring. While several algorithms exist for similarity matching of *static* time-series data, matching continuous data poses new, largely unsolved challenges including online real-time processing requirements and system resource limitations for handling infinite streams. In this work, we propose a novel live stream matching framework, called *n-Snippet Indices Framework* (in short, SNIF), to tackle these challenges. SNIF employs snippets as the basic unit for matching streaming time-series. The insight is to perform the matching at two levels of granularity: *bag matching* of subsets of snippets of the live stream against prefixes of the patterns, and *order checking* for maintaining successive candidate snippet bag matches. We design a two-level index structure, called SNIF index, that supports these two modes of matching. We propose a family of online two-level prefix matching algorithms that trade off between result accuracy and response time. The effectiveness of SNIF to detect patterns has been thoroughly tested through experiments using real datasets from the domains of fire monitoring and sensor motes. In this paper, we also present a study of SNIF's performance, accuracy and tolerance to noise compared against those of the state-of-the-art Continuous Query with Prediction (CQP) approach.

## Categories and Subject Descriptors

H.2.8 [**Database Application**]: Data mining

## General Terms

Design, Performance, Reliability

*This work was partly supported by National Science Foundation under grants IIS 0414567, SGER 0633930 and CRI 0551584.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*CIKM'08,* October 26–30, 2008, Napa Valley, California, USA.
Copyright 2008 ACM 978-1-59593-991-3/08/10 ...$5.00.

## Keywords

Streaming Time-series, Similarity Queries, Prefix Matching

## 1. INTRODUCTION

### 1.1 Time-Series Processing over Streams

The recent technological advances in sensor networks and mobile devices produce high volume data streams. Applications such as environmental monitoring of hazardous waste and poisonous attack clouds, network traffic monitoring and tracking web click stream require on-the-fly matching of streaming time-series sequences to a set of known patterns. Continuous time-series matching poses several research challenges [8, 9, 12, 16, 25].

A *live stream* is defined as a potentially infinite series of relational records. A *time-series* is a sequence of real numbers representing values from some given domain at specific points in time, also called data sequences. A live stream composed of time-series data is called a *streaming time-series* [8, 9].

According to [3], the processing of queries over streaming time-series is more complex than traditional *static* time-series for the following reasons. First, the elements in a live stream must be processed online due to the real-time requirements of the applications. The data tends to be continuously appended to the end of the live stream. Thus, to keep up with the high input rates, the most recent elements typically must be processed before the next elements arrive. In contrast, in static time-series stored in a database, there is no limit on the processing time. 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 finite, algorithms for processing them can access the whole sequences either sequentially or by preprocessing them into some indexed form for faster access. Third, any portion of the streaming time-series obtained previously cannot be assumed to be available at a later time. Since streaming time-series have infinite lengths, the data obtained in the far past must either be explicitly stored by the system in a compressed form or otherwise simply discarded. In contrast, in traditional static time-series database, the entire time-series can be retrieved at any time. Thus multiple passes can be performed.

To further complicate the problem, continuous similarity matching over a live stream may need to decide on matching intermittently while only partially knowing the live stream. Also the data sources may be noisy and gaps between the live stream and the pattern sequence may arise.

## 1.2 Our Proposed Solution: SNIF

In this paper, we propose *n-Snippet Indices Framework* (in short SNIF) for efficiently matching a streaming time-series against a set of numeric pattern sequences. As the live stream is infinite we work with chunks of data from the live stream. Based on the notion of n-Grams [6, 18, 14], originally introduced for textual information retrieval, we introduce the concepts of n-snippets and m-snippetCollections as the foundation for our matching framework. The insight is to match small snippets of the live stream against prefixes of the patterns. The longer the pattern prefixes are identified to be similar to the live stream, the better the confirmation of the match becomes.

In our framework, the live stream matching is performed using two layers of matching, namely, *bag matching* for quick matching of sets of snippets while allowing partial disorder and *order checking* for maintaining the sequence of the match at this more coarse-grain level. The *bag matching* step performs approximate matching of small chunks of the live stream data to quickly discard subsequences of pattern sequences within the live stream that definitely did not match. The *order checking* step is analogous to subsequently stitching the adjacent subsequences to discover which of the pattern sequences match the live stream and incrementally computing how closely each such potential candidate matches. SNIF can perform range queries as well as nearest neighbor searches. We design a two-level index structure, called the SNIF index, that supports these two layers of matching. In an offline step, the pattern sequences are preprocessed and then loaded into this SNIF index. During the online *live stream matching* step streaming time-series is matched on-the-fly against the indexed pattern sequences.

## 1.3 Contributions

The main contributions of this paper are as follows:

1. We abstract the Continuous time-series sequence matching problem into a formally defined *Prefix matching* problem and propose a snippet granularity matching solution that extends well known n-grams [6], originally applied to text matching, to matching numeric time-series data. The approach gains from the efficient query processing capabilities of n-grams, while preserving the essence of approximate similarity measures established over the years for matching of numeric data.

2. We define a two-layered progressive matching technique for numeric time-series data: *Bag Matching* for early filtering of patterns followed by *Order Checking* for refined checking of false positives to precisely match prefixes over *n-snippets*.

3. SNIF combines all the features desired in Continuous time-series matching such as:
   - (a) An efficient two-level index facilitating matching against patterns having different lengths.
   - (b) A flexible framework giving the user the choice to plug in any domain-specific similarity measure at the snippet granularity.
   - (c) Incremental matching storing the tiny portion ($\ll$ pattern length) of the infinite live stream.

4. SNIF is implemented inside the data stream engine CAPE [20]. Our experiments demonstrate that:
   - (a) SNIF is efficient and effective in identifying patterns on real datasets of fire monitoring [24], and Sensor Motes [21] datasets;
   - (b) SNIF outperforms the state-of-the-art CQP [8] in CPU costs as well as match accuracy;
   - (c) SNIF performs robustly under considerable amounts (up to 20%) of randomly introduced noise in the live stream data.

The rest of this paper is organized as follows. Section 2 presents the background and the matching problem definition. We introduce in Section 3 the proposed solution while Section 4 describes the matching framework. Section 5 presents evaluation while Section 6 reviews related work. Section 7 summarizes our work.

# 2. PRELIMINARIES

## 2.1 Assumptions

| Symbols | Definitions |
|---|---|
| $S_{ID}$ | Sequence with its unique identifier ID |
| Len(S) | Length of a sequence S ($S_L$ or $S_P$) |
| 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. |
| $D(S_1, S_2)$ | Distance between sequences $S_1$ and $S_2$ based on a chosen distance measure. |

**Table 1: List of notation used**

Without loss of generality, we assume that all time-series are one-dimensional, i.e., each entry is of the form S[t], where S[t] is the data value at time t. We also assume that all time-series are sampled at *equidistant* time intervals. If originally they were sampled at unequal intervals, then we interpolate the values to make them equally intervaled. We further assume that the first sample is taken at time 0. A time-series S is finite if it extends only up to a finite length $L \geq 0$ (Len(S) = L), denoted as $\langle$ S[0], S[1], S[2],..., S[L] $\rangle$. A time-series S is infinite if no such L exists.

## 2.2 Problem Definition

A *pattern sequence* $S_P$ is a *finite* sequence of time-series data, such as a sequence of sensor readings that records the characteristic behavior during a phenomenon (such as a fire event). Given a library of such pattern sequences, similarity queries find those pattern sequences from a library that are most similar to a query sequence $S_Q$, given by the user. Many similarity measures are possible, for instance weighted Euclidean distance [15], time warping [13], and wavelets [4].

*Definition 1.* Given a library $Lib_P$ of N pattern sequences, $\langle S_{P0}, S_{P1}, \ldots, S_{PN-1} \rangle$, each having the same length L, a pattern $S_{Pi}$ is said to be the Nearest Neighbor of the query sequence $S_Q$ if for all other $S_{Pj}$, $j \neq i$, $D(S_Q, S_{Pi}) < D(S_Q, S_{Pj})$. Similarly, the k−Nearest Neighbors (NN) is a set of size k NN of the top k patterns in $Lib_P$ ranked by their distance from $S_Q$.

The above assumes that both the query sequence and the pattern sequences are finite and static. In a streaming environment, a live stream replaces the fixed finite query sequence $S_Q$. The pattern sequences $S_{Pi}$ also need not be of same lengths. In the static sequence matching scenario

$S_Q$ and $S_P$ are available in full to be compared against each other, while in a live streaming scenario the query sequences $S_Q$ need to be extracted out of the ever growing live stream $S_L$.

A *live stream*, denoted as $S_L$, is an infinite time-series data sequence to which new data entries are continuously appended at every time unit. $S_L$ consists of a sequence of data values collected starting at time 0 until the current time $t_c$, denoted as $S_L[0{:}t]$ = $S_L[0]$, $S_L[1]$, ..., $S_L[t_c]$. A subsequence of $S_L$ of length l ending at a time $t_s$ is denoted by $S_L[t_s\text{-}l\text{+}1{:}\ t_s]$. At any time $t_s$, the distance between the live stream $S_L$ and a pattern sequence $S_{Pi}$, having length $L_i$, is denoted as $D(S_L[t_s\text{-}L_i\text{+}1{:}t_s], S_{Pi})$. The definition of a similarity query changes to accomodate the dynamic nature of the live stream.

*Definition 2.* Given a current time $t_c \geq 0$, a pattern $S_{Pi}$ is the *Nearest Neighbor of $S_L$* at $t_c$ if for all other patterns $S_{Pj}$, $j \neq i$, $D(S_L[t_c\text{-}L_i\text{+}1{:}t_c], S_{Pi}) < D(S_L[t_c\text{-}L_j\text{+}1{:}t_c], S_{Pj})$. Similarly, to find the *k-Nearest Neighbors of $S_L$*, at time $t_c$ all the patterns are ranked by their distances from $S_L$ and the top k are output.

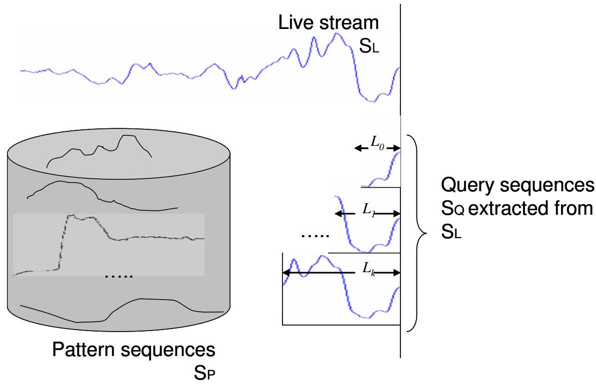



**Figure 1: Query sequences over live stream**

Real-time continuous time-series matching technique must aim at facilitating early detection of the patterns within the live stream. For critical applications such as the detection of fire patterns, the ability to find a complete pattern of the overall fire event is a little too late, i.e., the fire would have come and burned down. Notice in Figure 1 that for the different length values $L_i$ of patterns, several suffixes of the live stream of the form $S_L[t_c\text{-}L_i\text{+}1{:}t_c]$ are used as the query sequence, $t_c$ being the current timestamp. The suffixes of $S_L$ are continuously matched against the prefixes of the patterns $S_P$ aiming for early detection of the match. Thus we match several query sequences against the library of patterns now. Rather than requiring to collect the full pattern length of data from the live stream, we propose to incrementally match the live stream against the *prefixes* of the patterns. We call this *prefix matching* (Figure 2). The longer the prefix of the pattern matched with the portion of the live stream the better the confirmation of the match.

*Definition 3.* Given a live stream $S_L$ matched against a library *Lib* of patterns $S_{Pi}$, *continuous time-series similarity query* using *prefix matching* is accomplished by maintaining, for each pattern $S_{Pi}$ of Length $L_i$ in the *Lib*, $S_Q$ suffixes $\varsigma$ extracted from $S_L$ of the form $\langle S_L[t_c{:}t_c], S_L[t_c\text{-}1{:}\ t_c], \ldots, S_L[t_c\text{-}L_i\text{+}1{:}t_c]\rangle$ and prefixes $\varrho$ of each pattern $S_{Pi}$ of the form $\langle S_{Pi}[0{:}0], S_{Pi}[0{:}1], \ldots, S_{Pi}[0{:}L_i\text{-}1]\rangle$ such that $D(\varsigma_j, \varrho_j)$

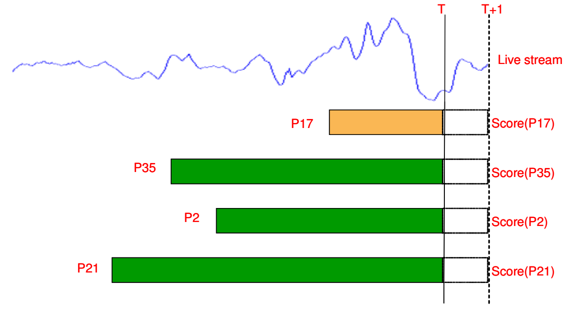



**Figure 2: Prefix Matching: matching prefixes of the patterns against the suffixes of the live stream**

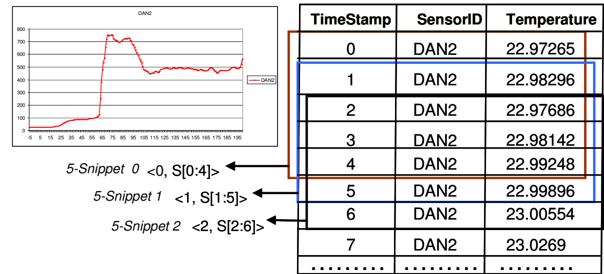
$= D(S_L[t_c\text{-}j\text{+}1{:}t_c], S_{Pi}[0{:}j\text{-}1]) \leq \Delta$threshold, where j is the length of the matched prefix $\varsigma$ and suffix $\varrho$. Similarly, Nearest Neighbor and k-NN for current time $t_c$ can be obtained as the prefix of the pattern matched closely with the suffix of $S_L$ until the current time $t_c$.

# 3. SEQUENCE MATCHING USING BAG AND ORDER SEMANTICS

In this section we first introduce the concepts of n-Snippets and m-SnippetCollections that form the building blocks for our match framework. Then we describe alternative matching approaches using n-Snippets and m-SnippetCollections.

## 3.1 Snippet-based Similarity Measure

An *n-snippet* is our basic unit for matching. We continuously extract snippets from a sequence by collecting groups of n consecutive data values. Two adjacent snippets of size n overlap by n-1 datapoints. Figure 3 represents a sequence of temperature readings from sensor DAN2 taken from the EDaFS [24] dataset.



| 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 |
| ......... | ......... | ......... |



**Figure 3: Forming n-snippets from a sequence**

*Definition 4.* An *n-snippet*, or henceforth also called *snippet*, is a subsequence of *n* consecutive data values in a sequence S, represented as a tuple ⟨ p , S[p : p+n-1] ⟩ starting at the $p^{th}$ position of S.

In our framework, a snippet need not hold the n datapoints in their entirety. The snippet can be a reduced representation of the n datapoints capturing the essential content of the snippet sequence. Possible representations may be fourier coefficients, wavelets or data statistics such as average, slope, and standard deviation. While any representation can be plugged in, the best choice greatly depends on the domain and the dataset as demonstrated in [22].

A similarity measure that is a good distinguisher between alternate patterns in the domain is the most suitable snippet

representation. Moreover, the representation should ideally eliminate noise and be inexpensive to compute. As there is much overlap between consecutive snippets, an incrementally computable measure is desirable. Moreover, the combination of measures used to represent a snippet may be a better confirmation of the match.

For our work, we chose the pair of the average and the standard deviation of the n datapoints to form the snippet representation. Both metrics are inexpensive to compute even on live streams. A moving average smoothes the data, thus eliminating some noise. We observe empirically that this pair of metrics forms a significant distinguisher between the pattern sequences in the datasets we examined. The standard deviation by itself is not a strong candidate since the same standard deviation value can occur at totally different temperature bandwidths. A combination of the two metrics (average & standard deviation) as the similarity criteria is a more reliable match measure than either taken alone. This reduces distance computation costs as we need to match only the two data statistics instead of n data values (assuming $n \geq 2$).

To summarize, an n-snippet of sequence S, starting at $p^{th}$ position, is henceforth represented as $\langle$p, (Avg(S[p : p+n-1]), Stdev(S[p : p+n-1]))$\rangle$. For simplicity, we henceforth use Euclidean distance over the *normalized* average & stdev pair (Formula 1) to compare two snippets, though *weighted* Euclidean distance or any other similarity measure could be plugged in too.

$$\Delta(\text{Snip}_A,\text{Snip}_B) = \sqrt{(\text{Avg}_A - \text{Avg}_B)^2 + (\text{Stdev}_A - \text{Stdev}_B)^2} \quad (1)$$

*Definition 5.* A snippet pair ($\text{Snip}_A$, $\text{Snip}_B$) is said to have matched if $\Delta(\text{Snip}_A,\text{Snip}_B) \leq$ a user defined tolerance ΔAvgStdev.

## 3.2 Two Alternatives for Snippet-based Matching of Sequences

We propose two strategies for snippet-based matching of sequences, namely, *bag matching* and *order checking*. We will note that these alternatives work irrespective of the choice of snippet representation.

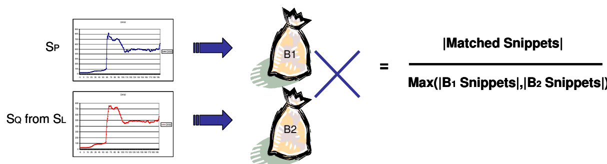



**Figure 4: Bag matching of snippets over sequences**

*Definition 6. Bag matching* is a sequence matching strategy where the set of snippets of $S_Q$ *snip-set($S_Q$)*, is matched against the set of snippets of $S_P$ *snip-set($S_P$)*. The match score $S_{BM}(S_Q, S_P)$ is computed using Formula 2. The sequence pair ($S_Q$, $S_P$) is said to have matched with score $S_{BM}(S_Q, S_P)$ if $S_{BM}(S_Q,S_P)$ is within a certain user specified threshold $S_{Col}$Threshold for bag match.

$$S_{BM}(S_Q, S_P) = \frac{|\text{Matched n-snippets between } S_Q \text{ and } S_P|}{\max(|\text{n-snippets in } S_Q|,|\text{n-snippets in } S_P|)} \quad (2)$$

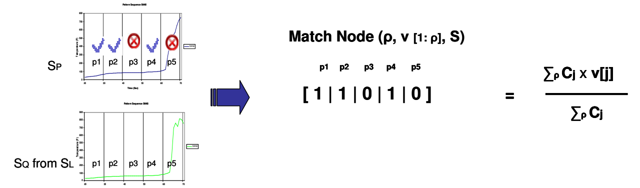



**Figure 5: Order checking of snippets over sequences**

Intuitively, we consider $S_Q$ and $S_P$ *matched* if they have enough snippets in common. See Figure 4 for an example.

In bag matching there is no notion of sequence matching as the order of occurrence of the snippets is not considered. For this, we introduce the notion of *order checking*.

*Definition 7. Order checking* is defined as a sequence matching step where the order of occurrence of terms (here, snippets or collections of snippets) within sequences is checked. The order checking score is maintained in a *match store* for each possible prefix $\varrho_\rho$ of a pattern $S_{Pi}$, the match score can be computed up to any position $\rho$ in the pattern using Formula 3. A *match store* is a triplet $\langle\rho, \nu[1{:}\rho], S^{\rho}_{OC}\rangle$ with:

1. *Match Position* $\rho$ - Position of current snippet up to which $S_{Pi}$ has been matched above the threshold.
2. *Match Vector* $\nu[1{:}\rho]$ - A vector recording $\Delta(\text{Snip}_A,\text{Snip}_B)$ ($\leq$ ΔAvgStdev) but only if the snippets within $S_L$ occur in the original order of $S_{Pi}$; $\text{Snip}_A \in S_Q$ and $\text{Snip}_B \in S_{Pi}$.
3. *Match Score* $S^{\rho}_{OC}$ - Cummulative score of the individual scores stored in the Match Vector $\nu$ up to the Match Position $\rho$.

$$S^{\rho}_{OC}(S_Q,S_{Pi}) = \frac{\sum_{j=1}^{\rho} C_j \times \nu[j]}{\sum_{j=1}^{\rho} C_j} \quad (3)$$

In Formula 3, $C_j$ is the weight associated with snippet at position j, $\nu$[j] is the $\Delta(\text{Snip}_A,\text{Snip}_B)$ score at position j. An auxiliary structure called *match store* maintains the $S_{OC}$ score. As shown in Figure 5, $S_Q$ is said to have matched with $S_P$ if the strict order of the snippets is maintained between them. In other words, the order checking score $S^{\rho}_{OC}(S_Q, S_{Pi})$ being within a certain user-specified threshold is a measure of the extent of match between the ($S_Q$, $S_{Pi}$) pair. However, the strict ordering of terms is a rather stringent condition whereas in practice an approximate match as opposed to an exact match may be more realistic. Our solution takes advantage of the best of both strategies as explained in the following section.

## 3.3 Integrated Bag and Order Matching

On the one hand, we would like to perform a bag match allowing some local disorder among the snippets. On the other hand, we would like to assure that the order within the sequences is mostly maintained. To facilitate this we now introduce another layer of granularity between individual snippets and complete sequences. We call it an *m-SnippetCollection* or simply a collection. Forming collections of m consecutive snippets out of a sequence S divides S into $\lceil(\text{Len}(S)/m+n-1)\rceil$ groups of consecutive snippets. Each snippet collection consists of m+n-1 consecutive data values. Two consecutive collections overlap by just n-1 data values, where n is the snippet size.

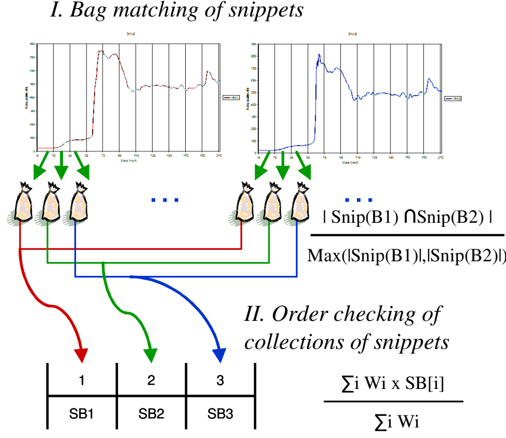



**Figure 6: Two layers of matching**

Instead of conducting either bag matching or order checking at the snippet level, we now propose to integrate these methods into a two-layered matching technique (as illustrated in Figure 6):

1. *Bag matching* across snippets within a collection of m consecutive snippets, and

2. *order checking* across the matched m-snippetCollections of snippets for a sequence.

This arrangement achieves our goal of allowing local disorder while still maintaining overall order of the sequences. The bag match score for each matching collection pair is passed on to its position in the match vector for computation of the order checking score for the sequence.

An order match between a $S_Q$ and a particular pattern $S_{Pi}$ is established only if the consecutive collections of snippets of $S_Q$ are found in exactly the same order as that of collections of snippets in $S_{Pi}$. Similar to order checking over snippets, here also we make use of a *match store* for each $S_{Pi}$ to maintain the match with slight modifications, namely, $\rho$ is now the position of the current *collection* up to which the $S_{Pi}$, The match vector $\nu[1{:}\rho]$ records the *bag matching* scores of the collections in the order they occur in the original $S_{Pi}$.

An example of a match store is $\langle$ 3,$\langle$0.98|1.0|0.89$\rangle$, 0.95 $\rangle$. Here 3 denotes the match position $\rho$. $\langle$0.98|1.0|0.89$\rangle$ denotes the scores of the first 3 consecutive collections of $S_{Pi}$, and 0.95 is $S^3_{OC}$ computed according to Formula 3. The value of $\rho$ for a $S_{Pi}$ can vary from 0 to $\lfloor$Len($S_P$)$\div$(m+n-1)$\rfloor$, i.e., the number of collections the $S_{Pi}$ can be divided into.

*n* determines the degree of smoothing. To preserve the significant patterns yet be able to eliminate noise, n needs to be small compared to the sequence length (n $\ll$ Len(S)). Setting n = 1 corresponds to no smoothing, whereas, setting a large n value may cause over-smoothing. Hence, smoothing over a small number 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 disorder in the snippets while still calling it a match. Ideally we would avoid the choice of extreme values for m. m =1 means *order checking* over individual snippet, whereas, m = Len(S) (i.e., equal to the size of the sequence) will mean *bag matching* of all the snippets in the sequence. 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)).

# 4. LIVE STREAM MATCHING

SNIF performs the matching of the live stream $S_L$ against the set of patterns $S_P$ in two phases:

1. *Off-line Preprocessing Phase*: Each $S_P$ is scanned once and snippets as well as collections are extracted. They are then loaded into a two-level index structure. The index is cleaned by removing approximate duplicates during this index construction process.

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 analogous to snippet extraction from each $S_P$. Each $L_S$ probes the first level of the index to perform bag matching over a collection worth of data. The high-ranked collections then probe the second level of the index to perform order checking to output the potential $S_P$ candidates.

## 4.1 Preprocessing: Designing Indexes for Bag- and Order- based Matching

One of the greatest challenges of the live stream matching problem is that the two layers of our proposed matching technique need to be performed on-the-fly between the live stream $S_L$ and each of the pattern sequences $S_P$. We propose to store the pattern sequences $S_P$ in a hierarchical structure composed of two inverted indices (for each match layer). An inverted index reduces the memory required to store the patterns, allows quick lookup and, as an inherent feature, returns matches ordered by frequency counts.

We first define the terms occurrence list and inverted index. For a term (here snippets or collections), an *occurrence list* consists of the identifier of $S_P$ and the list of offsets where the term occurs within $S_P$. An *inverted index* consists of a map between the terms (here snippets or collections) and their occurrence list. The two indices are shown in Figures 7 and 8.

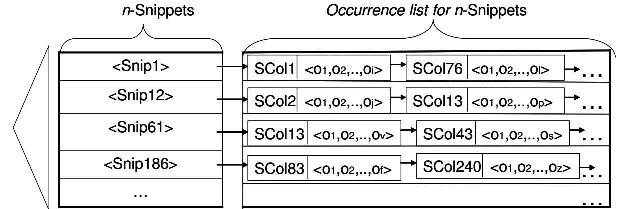



**Figure 7: n-Snippet Index**

The front-end inverted index, also called the snippet index (Figure 7), uses snippets as indexing terms. For each snippet it maintains an occurrence list that contains information about the occurrence of the snippet within the collections. The occurrence list information corresponds to a vector $\langle$ $SCol_{ID}$,$\langle$ $o_1$,$o_2$,..,$o_i$ $\rangle$ $\rangle$, 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. This information is used for bag matching of snippets to report what fraction of a collection has matched.

The back-end inverted index (Figure 8) uses the identifier of the collections as indexing terms. For each collection, it maintains an occurrence list that contains information about the occurrence of the collection within $S_P$. 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.

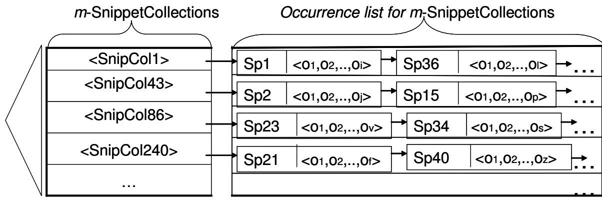



**Figure 8: m-SnippetCollection Index**

The back-end index, also called the m-SnippetCollection index, is used for order checking of collections within $S_P$.

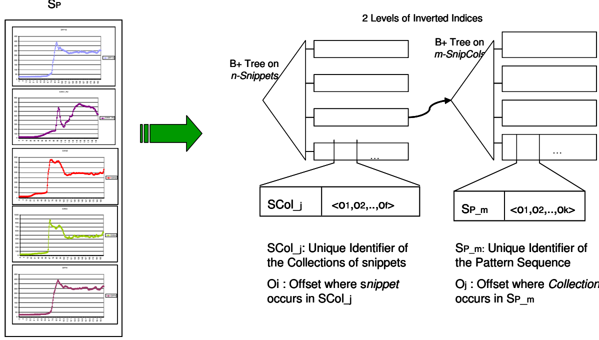



**Figure 9: Building the hierarchical two-level index**

The index structure is supplemented by a $B^+$-tree structure called Avg-StdevSortedTree (ASTree) of the snippets present in the front-end index. It is sorted on the snippet similarity measure values.

The snippets and snippetCollections are extracted from $S_P$ and loaded in the index during preprocessing. Every time we collect *m* number of snippets, they are grouped together and given a unique collection identifier. The snippets are loaded into the snippet inverted index, called front-end index, as during matching it is matched against the snippets of the live stream. In a similar manner, the m-SnippetCollections are loaded into the collection inverted index, called the back-end index, as this is not directly probed by $S_L$.

During live stream matching, first, a range search would be performed on the ASTree to extract similar (based on the similarity measure) snippets. The snippets would result in multiple (likely redundant) front-end index probes. To avoid this, we reduce the index size by clustering the snippets on the similarity measure values using some third party clustering tools [10].

To summarize, the preprocessing phase consists of three tasks:

1. Extracting snippets and collections from each $S_P$.
2. Clustering snippets and associating with a cluster identifier ($C_{ID}$).
3. Loading the two levels of the index.

## 4.2 Live Stream Matching Phase

Using the two-level index structure, the live stream matching is divided into two layers 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 long has the $S_P$ matched.

These two abstract layers of matching make it possible to match the live stream against $S_P$ of different lengths.

### 4.2.1 n-Snippet Index Lookup

As new data is being appended to $S_L$, live snippets $L_S$ are extracted from $S_L$ (see Section 3.1). $L_S$ identifiers cannot directly probe the n-Snippet index. As an intermediate step, the similarity measure values (here, average and standard deviation) of $L_S$ are used to perform a range query over the ASTree. The retrieved identifier of a snippet is then used to probe the n-Snippet index to retrieve the potential collections to which $L_S$ belongs. The matching phase uses auxiliary structures for recording the matches at the two layers. The structure called *Collections-Latest-m-$L_S$*, records each extracted $L_S$ and the list of the collections corresponding to it.

As the input stream is infinite, the question arises for how many such live snippets $L_S$ we need to maintain the candidate collections? As the name suggests, we propose to maintain the collection of *Collections-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 $L_{Si}$ 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 $S_L$. If a time-series data point uses 'b' bits, then (m+n-1)× b bits of memory is required to store the live stream portion used in SNIF matching.

In our case, each of the Collections of $S_P$ and also the *Collections-Latest-m-$L_S$* 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-Latest-m-$L_S$* across the latest m $L_S$. We utilize another auxiliary structure called *FrequencyCount-Latest-m-$L_S$* to record the counts. For each collection in the collection list, the score is the ratio between its frequency count across m $L_S$ and the value m. Finding all m snippets of a collection, called a complete match, corresponds to the count m/m = 1.

The process of bag matching is illustrated using the two snapshots in Figure 10. Say, for our example, m = 26 and n = 5. As each live snippet $L_S$ is extracted from the live stream $S_L$, it probes the front-end index and extracts candidate collections. These collections are listed with the probing $L_S$ in the *Collections-Latest-m-$L_S$*. When we have *m* $L_S$ we can perform the frequency count of each collection listed in *Collections-Latest-m-$L_S$* and either create an entry in the *FrequencyCount-Latest-m-$L_S$* with its count or update an existing entry. The two figures show bag matching steps for live snippets 1 to 26 and 2 to 27. In Figure 10 as we transition from Snapshot 1 to Snapshot 2 , when $L_{S27}$ arrives, $L_{S1}$ gets eliminated. The frequency counts for each collection can be incrementally computed by taking into account the outgoing $L_{S1}$ and the incoming $L_{S27}$.

### 4.2.2 m-SnippetCollection Index Lookup

For the *order checking* step, the back-end index is looked up by the collection identifiers to fetch their corresponding occurrence lists. Out of the collections listed in *FrequencyCount-Latest-m-$L_S$* only the ones with count scores above the user defined threshold $S_{Col}$Threshold are used to look up the

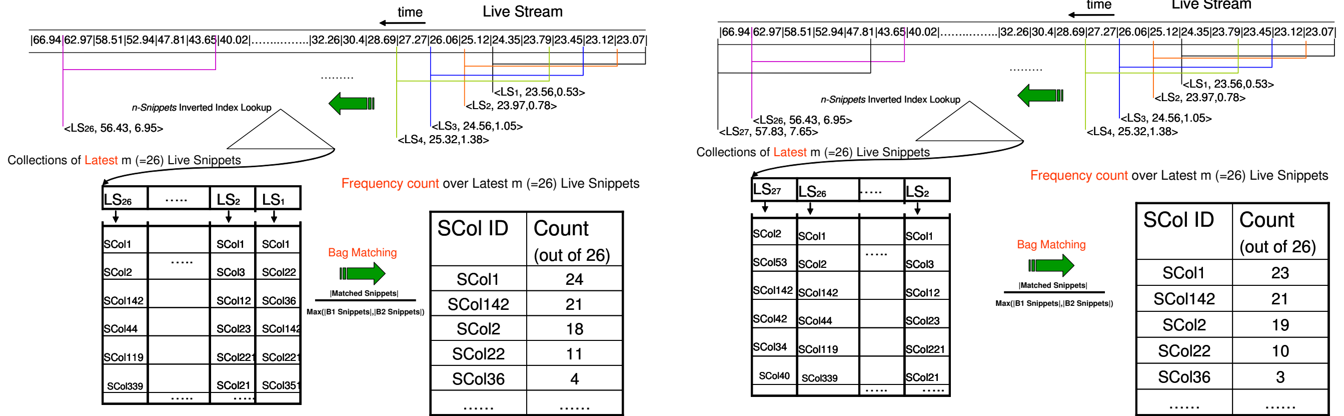



**Figure 10: Bag matching in progress: Snapshot 1, and Snapshot 2**

back-end index. By probing the back-end index the candidates $S_P$ are obtained from the occurrence lists.

Figure 11 illustrates the process of order checking using the back-end index. Say, the $S_{Col}$Threshold = 0.65. Out of the collections in the *FrequencyCount-Latest-m-$L_S$*, the unshaded ones are highly ranked, i.e., having $S_{BM} \geq$ $S_{Col}$Threshold and are used to look up the back-end index. They are $S_{Col1}$, $S_{Col142}$ and $S_{Col2}$. The shaded ones are below the threshold. From $S_P$ and position information of the occurrence list the match for each candidate $S_P$ can be recorded in the $S_P$ match stores. Given the $S_P$Threshold = 0.85, looking at the match stores in Figure 11, the highly ranked are unshaded, namely, $S^3_{P1}$, $S^2_{P6}$, $S^3_{P6}$, and $S^4_{P3}$),i.e. having $S_{OC} \geq S_P$Threshold. Only a high-ranked $S_P$ will be reported as a candidate match.

The individual collection score is the result of bag matching between the collection and the latest m $L_S$. At the next layer, 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 stores are also incrementally evaluated and maintained just like the *FrequencyCount-Latest-m-$L_S$*.

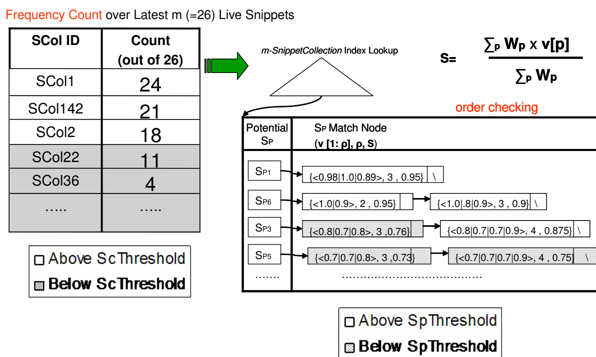



**Figure 11: Order Checking using Collection Index Lookup**

Next we discuss alternative approaches for incremental evaluation of $S_P$ match stores. First, while several $S_P$ matches can be formed, a mechanism is required for pruning adhoc matches and false positives. We propose to use another threshold, we call $S_P^{lower}$Threshold ($< S_P Threshold$) to discard $S_P$ matches that have $S_{OC} < S_P^{lower}$Threshold. Second, many collections, especially adjacent collections within a $S_P$ 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 (best $S_{OC}$ value) or the extent of the match (the $\rho$ value). Alternatively, multiple match vectors can be maintained for each $S_P$. We found that maintaining just the single $\nu[1{:}\rho]$ for a candidate $S_P$ works well for our two 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 the response time and result accuracy. Hence, there needs to be an upper bound on the maximum number of match vectors to be maintained per $S_P$.

We propose 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 stores maintained as a combination of best k and multiple 1 per position.

We list the complete algorithm for the live stream matching step in Algorithm 1.

# 5. EVALUATION

In this section we study the performance, the accuracy and the robustness of SNIF matching framework by a thorough comparative study against the state-of-the-art Continuous Query with Prediction (in short, CQP) proposed by Gao et. Al [9].

**System Implementation.** SNIF and CQP are both implemented in java over an existing continuous stream processing engine, CAPE [20]. For CQP implementation [11] FFT package was used. Experiments were conducted on an Intel(R) Pentium(R) machine with processor speed 1.70 GHz and 2GB memory.

**Algorithm 1** Live Stream Matching Algorithm

**Input:** The 2-level indices, Live stream sequence $S_L$
**Output:** Potential pattern sequences $S_P$ and their respective scores

```
 1: Range search on ASTree.
 2: SC[ ][ ] = φ
 3: counter = 0;
 4: for each n tuples in S_L do
 5:   Form snippets LS_i =< L_Sid, Avg, Stdev >
 6:   counter++
 7:   for each LS_i do
 8:     // find the set of matched pattern snippets
 9:     SC[i] = φ
10:     for PS_j ← RangeSearch(ASTree, LS_i) do
11:       // find a set of snippetCollections
12:       SC[i] = SC[i]∪ FrontEndIndexLookUp(PS_j)
13:   if counter == m then
14:     OList[ ] = φ
15:     for each SC_j ∈ SC[][] do
16:       if FREQUENCY_COUNT(SC_J, SC[ ][ ]) ≥
          S_col Threshold then
17:         // Determine the occurrence list of all patterns
18:         // (and its corresponding offsets) in which SC_j
19:         // exists of the form < S_P1, offset_P1 >, . . . <
            S_Pi, offsets_Pi >
20:         OList[j] ← BackEndIndexLookUp(SC_j)
21:       for each OList[j] ∈ OList[ ] do
22:         for each S_Pi ∈ OList[j] do
23:           MatchStores(S_Pi)
24:           if MatchScore(S_Pi) > S_P Threshold then
25:             return S_P as candidate
26:     Remove SC[0] from SC[][]
```

**Real Data Sets.** The experiments were conducted on 2 different real datasets, namely EDaFS [24] fire dataset and the sensor network motes dataset [21]. The EDaFS dataset contains temperature, smoke CO readings recorded during several live fire tests conducted in NIST fire labs to study smouldering and flaming fires. Motes dataset [21] consists of 4 groups of sensor measurements (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. Streams are heterogeneous streams, i.e., temperature shows a weak daily cycle and a lot of bursts, whereas humidity does not have any regular pattern.

**Pattern Sequences**. We use a collection of pattern sequences of various lengths, namely 200∼300, 500∼600 and 700∼800. Each collection consists of 50∼60 patterns which are extracted from each of the two real datasets. The pattern sequences are maintained in a two-layered index along with its auxiliary tree structure (ASTree).

We have designed three sets of experiments. First, we measure and compare the average CPU execution time of the proposed algorithms versus the state-of-the-art CQP. Figure 12 identifies that the SNIF algorithms execute at least 5 folds faster than CQP for all the three pattern lengths. The increase in pattern length drastically reduces the performance of CQP. Our two-level indexing strategy facilitates our SNIF algorithms to show better performance even for longer pattern lengths.

In the second set of experiments, we compare the accuracy of the different algorithms based on the match score

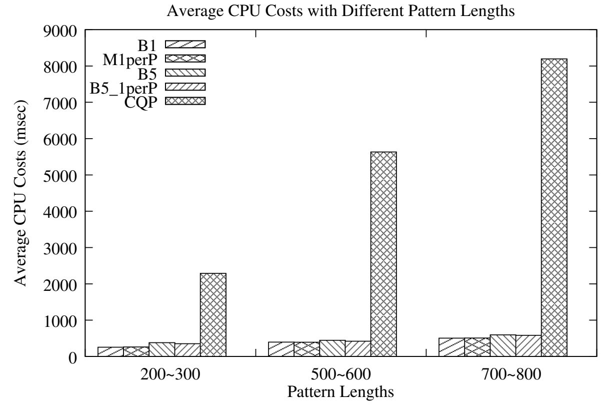


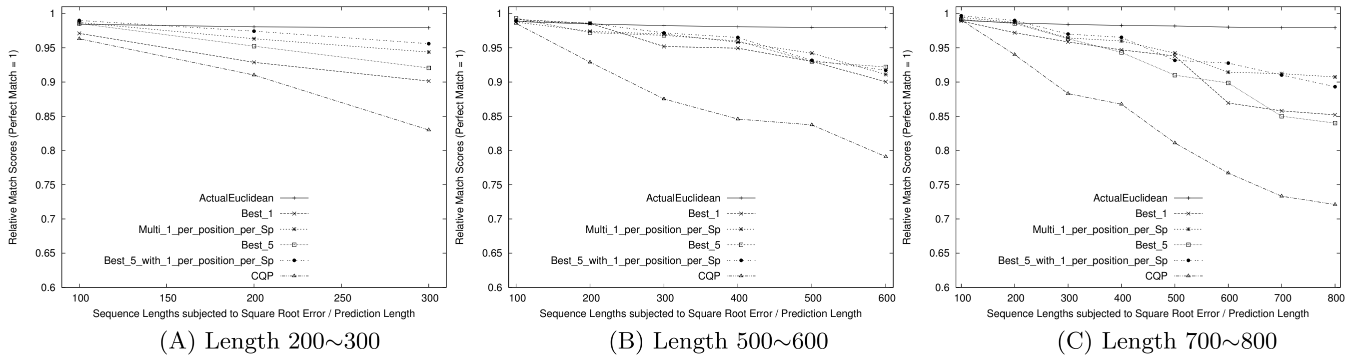

**Figure 12: Comparing Average CPU costs of SNIF algorithms against CQP**

measured on a scale from 0 to 1, with the perfect match score as 1. To faciliate this comparison we introduce **error** in the live stream using the error model, $Err_{sqrt}$, proposed in CQP. In $Err_{sqrt}$, the absolute error increases in the order of $O(\sqrt{k})$, which is defined as $Err_{sqrt} = a * RAND * \sqrt{k}$, where $k$ is the prediction step, $RAND$ is a uniformly distributed random variable having values between -0.5 to +0.5, $a$ is the *error control* which can scale up the prediction error as needed. In our experiments a is set to be 1. The actual Euclidean distances and the distances from the CQP algorithm are normalized and subtracted from 1 to compare against the match scores of the SNIF algorithms. In Figure 13 we clearly see that longer the sequence is subjected to error the less accurate of a match it is. In our experiments, the match score is a function of the algorithm tested and the portion of sequence subjected to error (equivalent to the prediction length for CQP). For all three pattern sets (200∼300, 500∼600, 700∼800), the SNIF algorithms sustain their match scores. While CQP matches deteriorate with longer predictions.

In the third set of experiments, we measure the robustness of our proposed algorithms against **random noise**. We examine three pairs of the snippet size n and collection size m, namely, (n=1, m=10), (n=5, m=10) and (n=5, m=3). For each pair we set the corresponding sizes for the pattern sequences and the live stream. We introduce different amounts of noise in the live stream ranging from 0% up to 20% by randomly dropping data values from them. We observe that for a given live stream, the match score is a function of the noise level and the (snippet size, collection size) pair. We find in Figure 14.A that snippet size set to 1 means no smoothing and the SNIF algorithms lose their accuracy beyond 5% noise level. On the contrary, setting moderate sizes, snippet size = 5 & collection size = 10 (Figure 14.B) truly brings out the robustness of the SNIF algorithms. Moreover, Figure 14.C having short collection sizes while maintaining the snippet size = 5, makes the SNIF algorithm vulnerable to the noise beyond 8 % error. Therefore, moderate snippet size and collection size make SNIF a robust stream matching algorithm.

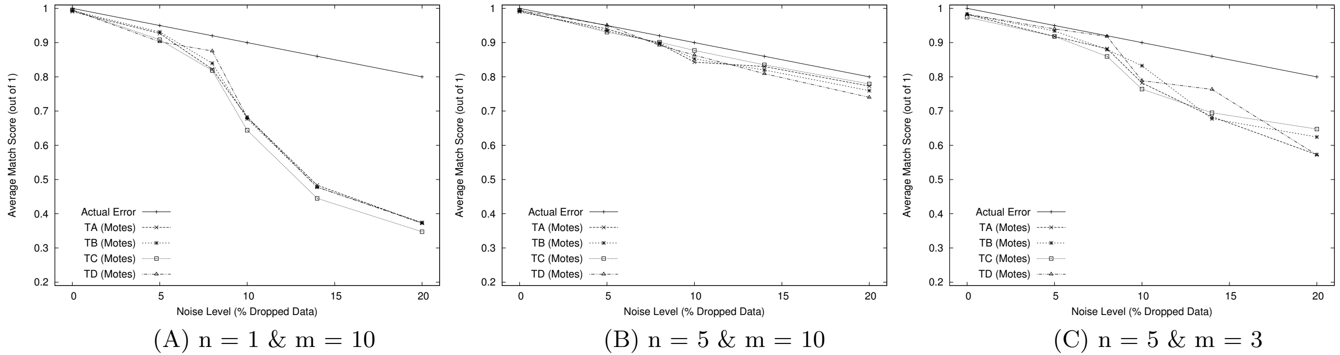
# 6. RELATED WORK

For *static* time-series data, numerous algorithms [1, 2, 7, 23, 19, 5] have been proposed for a variety of similarity matching queries. However, similarity matching over continuous streams still lacks a generic approach that can be used across domains.




(A) Length 200∼300 (B) Length 500∼600 (C) Length 700∼800

**Figure 13: Comparing accuracy of SNIF algorithms against CQP**




(A) n = 1 & m = 10 (B) n = 5 & m = 10 (C) n = 5 & m = 3

**Figure 14: Robustness of the SNIF algorithms to Noise Levels ranging from 0% up to 20% for different combinations of the Snippet Size (n) and the Collection Size (m).**

Gao and Wang [8] propose a prediction-based matching technique called Continuous Query with Prediction (in short, CQP). CQP monitors the streams to search patterns that are relevant and solves Nearest Neighbor and h-Near Neighbor queries (h being the distance tolerance) for whole matching. The already arrived data is used to predict future subsequences. They pre-compute the distances between the query sequence and the predicted subsequences employing Fast Fourier Transform (FFT). When the actual data arrives, prediction error and predicted distances is used to filter false positives. Their experiments show performance gains over a naive FFT approach with reasonable prediction errors. However, as admitted by the authors, in our experimental study we did not find the approach useful under large prediction errors such as when the live sensor streams have abrupt changes and high noise levels. CQP faces obvious overheads of performing on-the-fly FFT computations and adjusting the prediction error before every next batch processing. Moreover, their experimental results are based on synthetic data, hence CQPś applicability to real data is unknown. In our work, we compare our solution to this CQP approach and demonstrate superiority of our approach in performance and match accuracy.

Gao et. Al [9] propose another sequence matching method using prefetching which solves the k-Nearest Neighbor queries. Prefetching transforms the query sequences, extracted from the live stream, into lower-dimensional points, and stores them to disk in a multi-dimensional index. For new arriving data points, k-nearest query sequences are searched from the database. The arrived data values are used to predict k-NN candidates for the near future. The multidimensional index and the amortized disk reads handle a large number of query sequences. The prefetching solves k-NN for only fixed length patterns and it relies on a fixed tolerance of the pattern sequences, which is less applicable in real scenarios.

Kontaki et. Al [16] propose the IDC-index for streaming time-series matching in which DFT computations are performed incrementally over the streaming sequences. 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. To avoid frequent on-the-fly updates to the R*-tree, they propose a deferred update policy.

Han et. Al [12] present techniques for ranked subsequence matching using time warping, that finds top-k matches. They introduce minimum-distance matching-window pair (mdmwp) as a lower bound between the pattern subsequences and a query sequence. Based on the mdmwp-distance, they develop a ranked subsequence matching algorithm to prune false positives. Their method uses DTW [13] which suffers 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.

Overall, the state-of-the-art stream sequence matching algorithms have been extensions to the well-established *static* sequence matching techniques. Most reuse the dimensionality reduction as used for static time-series which are expensive to compute on-the-fly. These limitations motivate us to explore new avenues. One such matching technique, namely, n-Gram matching using inverted-index is yet to be explored for sequence time-series matching.

n-gram [6, 17, 18] based inverted index is a very popular and efficient information retrieval technique. Significant

work has been done in enhancement of n-grams [18, 14]. Kim et. Al [14], propose a two-level n-gram inverted index. Their proposed index has shown significant improvement in the query performance while preserving the advantages of the n-gram inverted index. However, the scope of n-gram is restricted to text matching. In this work, we extend n-grams to apply to numeric time-series data, calling it n-Snippets and develop approximate matching methods for prefix matching over the live streaming data.

# 7. CONCLUSION

In this paper, we abstract the continuous time-series sequence matching problem into a *prefix matching* problem and propose a generic snippet-based framework. We call it the n-Snippet Indices Framework (in short, SNIF). We introduce the concepts of n-snippets and m-snippetCollections for numeric data. We also propose to apply two abstract layers 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.

We demonstrate the efficiency and effectiveness of SNIF in matching live streams to sets of patterns having different lengths. We also compare SNIF to the state-of-the-art CQP [8] approach and demonstrate superiority of our approach in performance and match accuracy. We further demonstrate the robustness of SNIF to various noise levels. We successfully test the framework over two distinct datasets.

# 8. REFERENCES


[1] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.

[2] R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *VLDB*, pages 490–501, 1995.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[4] K.-P. Chan, A. W.-C. Fu, and C. 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.

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

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

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

[8] L. Gao and X. S. Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *SIGMOD Conference*, pages 370–381, 2002.

[9] L. Gao, Z. Yao, and X. S. Wang. Evaluating continuous nearest neighbor queries for streaming time series via pre-fetching. In *CIKM*, pages 485–492, 2002.

[10] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng.*, 15(3):515–528, 2003.

[11] T. Hammond, C. F. Gerald, and P. O. Wheatley. *Algorithm used from Applied Numerical Analysis*. Addison Wesley, 1994.

[12] W.-S. Han, J. Lee, Y.-S. Moon, and H. Jiang. Ranked subsequence matching in time-series databases. In *VLDB*, pages 423–434, 2007.

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

[14] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[15] S.-W. Kim, D.-H. Park, and H.-G. Lee. Efficient processing of subsequence matching with the euclidean metric in time-series databases. *Inf. Process. Lett.*, 90(5):253–260, 2004.

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

[17] J. H. Lee and J. S. Ahn. Using n-grams for korean text retrieval. In *SIGIR*, pages 216–224, 1996.

[18] E. Miller, D. Shen, J. Liu, and C. Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *Journal of Digital Information*, 2000.

[19] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.

[20] E. A. Rundensteiner, L. Ding, T. M. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[21] J. Sun, S. Papadimitriou, and C. Faloutsos. Distributed pattern discovery in multiple streams.

[22] A. S. Varde, E. A. Rundensteiner, C. Ruiz, M. Maniruzzaman, and R. D. S. Jr. Learnmet: learning domain-specific distance metrics for plots of scientific functions. *Multimedia Tools Appl.*, 35(1):29–53, 2007.

[23] C. Wang and X. S. Wang. Supporting content-based searches on time series via approximation. In *SSDBM*, pages 69–81, 2000.

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

[25] H. Wu, B. Salzberg, G. C. Sharp, S. B. Jiang, H. Shirato, and D. 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.