Multivariate Correlation Discovery in Streaming Data

d'Hondt, Jens

Award date:
2021

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science
Database Research Group

Multivariate Correlation Discovery in Streaming Data

Master Thesis

Jens d'Hondt

Assessment Committee:
dr. ir. O. Papapetrou
dr. ir. V. Menkovski
prof. dr. ir. G.H.L. Fletcher

Supervisor:
dr. ir. O. Papapetrou

Eindhoven, September 2021

Abstract

Correlation analysis plays an important role in large-scale event monitoring applications over streaming data. Most works to date focus on detection of strong pairwise correlations - pairs of streams with high Pearson correlation over sliding windows. A generalization of this problem involves discovery of strong multivariate correlations over streams, i.e., finding *sets* of streams (in the order of 3 to 5 streams) exhibiting high interdependence as a whole. Although several applications of this problem are known, an efficient solution is yet to be proposed, owing to the problem's immense search space. In this work we propose the first-ever streaming algorithm for multivariate correlation discovery. Our algorithm handles both synchronous and asynchronous streams, works with two different query types, and supports addition of user constraints. We also propose an extension to the algorithm which is able to react to anomalous behavior of the input streams, safeguarding short- and long-term performance. Extensive experimental evaluation with 5 datasets shows that our algorithms are able to handle streams with millisecond-level arrival rates, outperforming alternative approaches by multiple orders of magnitude.

Preface

This thesis marks the end of my academic career as a student. Despite including some less eventful times related to a certain pandemic, the last five years have been a rollercoaster of exciting experiences and newly-formed friendships. Beginning my journey at the school of Industrial Engineering has helped me develop my interests, which sparked my passion for Computer Science and matured me as a person. Working on this thesis has only strengthened this passion, though it would not have been possible without the support of the following people.

First of all, I would like to thank Odysseas Papapetrou for his guidance over this past year. You have been an incredible supervisor; not only for this thesis, but also for my internship at BMW. Your commitment to help and advice your students has been truly inspiring, and, combined with the absence of offline contact, has had me question your existence as a real person, and not some highly advanced AI. I hope we will keep in contact in the future in whatever endeavors we will both get involved with.

Additionally, I would like to thank Vlado Menkovski and George Fletcher for participating in the assessment committee of this thesis.

Special thanks to Koen Minartz, with whom I have shared my entire academic career, including this graduation process. You have always kept me on my toes, improving the quality of our work, and making the whole process a lot more enjoyable.

I would also like to thank my brother Tim for proof-reading this work, but more importantly for being a great brother and friend.

Thanks to my friends Miguel and Emaar, who have hosted me at tough times during this process, helping me regain my motivation and my (faded) summer tan.

Finally, a big thanks to my parents for their unconditional support throughout these years, my girlfriend Rosella for her endless love and her ability to keep me sane, and my friends in Eindhoven, Hulst, and Munich for making my student years unforgettable.

Jens

Contents

List of Figures

List of Tables

Glossary

basic window A consecutive subsequence of timepoints over which a system maintains a digest, e.g., a few minutes. 5

breaking cluster combinations The act of splitting the largest cluster of an indecisive cluster combinations into its sub-clusters to create new cluster combinations. 23

changing state Phenomenon of cluster combinations moving in or out of the result set, or becoming indecisive. 20

correlation pattern Combination of maximum set sizes (i.e, cardinality) of queried highly-correlated vector combinations. 9

digest Aggregation of the values observed in one basic window. 5

dominant bound A DCCs **mc**-bound that is closest to the correlation threshold τ. 20

epoch The period in which a batch of updates is processed. Usually the period between two timepoints. 8

extrema pair clusters Clusters that share an extrema pair in different decisive cluster combinations. 23

extrema pairs Pairs of streams/vectors that make up a cluster combinations' **mc**-bound. 20

forward-filling Imputation technique where missing values are filled with the most recent observed value. 7

global update Process of sliding the windows of all considered streams simultaneously, either by observing an actual new value or imputing one. 7

local update Process of only sliding the windows of streams that received a new value from one timepoint to another. 7

materialization A set of vectors that can be formed by picking picking precisely one vector from each cluster in a cluster combinations. 16

negative decisive cluster combinations Cluster combinations that have both **mc**-bounds below the correlation threshold τ. 16

one-shot Algorithmic technique that computes the query answer in one go, without using any information obtained prior to the run. 3

positive decisive cluster combinations Cluster combinations that have both **mc**-bounds above the correlation threshold τ. 16

significance of updates The extend in which new values in a stream violate existing decisive combinations. 23, 32

singleton clusters Clusters containing only one stream/vector. 16

sliding window A consecutive subsequence of basic windows over which the user queries statistics, e.g., an hour. 5

synchronous streams Streams that always receive new values at identical timepoints. 6

timepoint The smallest unit of time over which a system collects data, e.g., second. 5

warm-up period Time period in which we gather data on the performance of CD and CDStream in order to train the regressor used by CDHybrid to select the optimal algorithm. 33

List of Symbols

δ	Minimum jump factor
$\mathcal{R}$	Query result set
$\mathcal{S}$	A set of stream time series
$\mathcal{V}$	A set of discretized stream time series
$\mathcal{X}$	A set of sets (caligraphic capital)
$\mathbf{mc}$	Multiple correlation measure
τ	Correlation threshold
$\hat{\mathbf{v}}$	A z-normalized vector (bold-faced with hat)
$\mathbf{v}$	A vector (bold-faced)
B	Batch-size (either number of arrivals or time-period in which arrivals are received
b	Buffer size in top-k queries
b_{bw}	Basic window size
C	A set (upper-case)
k	Number of vector combinations with the highest multiple correlation coefficient
L	The index of the running basic window
$l_{\max}$	Maximum size of left-hand side of correlation pattern
n	The number of considered streams
$r_{\max}$	Maximum size of right-hand side of correlation pattern
$s_i[x : y]$	The observed values of stream i between timepoint x and y
S_i	The discretized version of stream s_i (sequence of digests)
s_i	A stream with index i
t	The most recent timepoint
w	Sliding window size

Chapter 1

Introduction

The following chapter provides a high-level motivation for the current study including a context description of the project setting in which the research took place. Further, the academic contributions of the work are presented and an outline is provided for the contents of the remainder of the thesis.

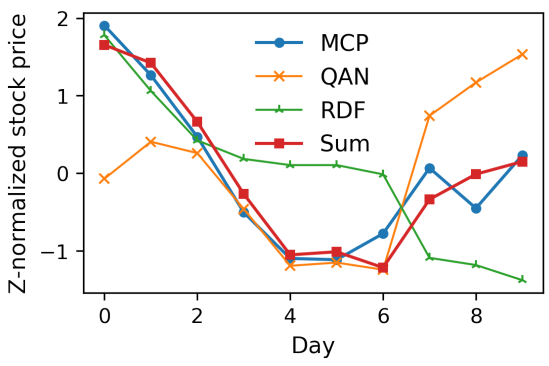
1.1 Motivation

Correlation analysis plays an important role in the toolbox of data analysts, for understanding the data and extracting insights. For example, in neuroscience, a strong correlation between activity levels in two regions of the brain indicates that these regions are strongly interconnected [14]. In finance, correlation plays a crucial role in portfolio diversification in order to limit risk while striving for maximum expected returns [28]. In genetics, correlations have helped scientists better understand hereditary syndromes and their causes. A prime example is the Spark project for discovering gene properties related to the manifestation of the autism spectrum disorder [10], which led to a list of genes and their correlated symptoms [11]. In database management design, correlations – as a generalization of functional dependencies – found use for optimizing access paths in databases [45].

Multivariate, or high-order correlations, are a generalization of pairwise correlations aimed at measuring relations between arbitrarily-sized sets of variables, represented either as high-dimensional vectors or as time series.[1]

In the last years it was repeatedly shown that multiple correlations come with great potential for discovering novel insights from the data, and for better understanding natural phenomena. To illustrate, detection of ternary correlations in fMRI time series improved our understanding of how different brain regions work in cohort for executing different tasks [2, 3]. For instance, the activity of the left middle frontal region was found to have a high correlation with the average activity of the right superior frontal and left inferior frontal regions while the brain was processing audiovisual stimulus. This insight suggests that the left middle frontal has an integrative role of assimilating information from the other two regions, which was not possible to find by looking only at pairwise correlations. In climatology a ternary correlation led to the characterization of a new weather phenomenon and to improved climate models [25]. In the fields of genomics and medicine, researchers found through multivariate correlation analysis that presence of multiple RASopathy genes contributed to an elevated risk of autism spectrum disorders (ASDs) due to a phenomenon called epistasis [12]. This phenomenon involves the dependence of the effect of a gene mutation on the presence or absence of mutations in other genes. In other words, multiple genes interact with each other which impacts the expression of a disease, while each gene individually only has weak correlation with the disease trait [40, 48, 24, 30].

[1]In the remainder of this thesis we will generally refer to the specific case of time series, as it is most relevant to the context of streaming data.



(a)

	MCP	**QAN**	**RDF**	**Sum**
MCP	1	0.46	0.53	0.96
QAN	0.46	1	-0.47	0.52
RDF	0.53	-0.47	1	0.52
Sum	0.96	0.52	0.52	1

(b)

Figure 1.1: (a) Normalized daily closing prices for stocks traded at the Australian Securities Exchange, (b) Correlation matrix of the prices. Source: [29]

Several multivariate correlation measures have been proposed throughout the last years. Agrawal et al. [2] defined tripoles – sets of three vectors – where the sum of the first two vectors has a high linear correlation with the third one. Three years later, Agrawal et al. was introduced the multipoles measure, which measures the degree of linear dependence of a vector set of arbitrary cardinality [3]. Similarly, Canonical Correlation Analysis (CCA) aims at finding linear combinations of two sets of vectors such that the Pearson correlation of the two resulting vectors (i.e., the aggregates) is maximized [17]. There also exist non-linear multivariate correlation measures that are frequently used. For example, Total Correlation is an information theoretic measure that expresses the amount of information shared between variables [44], serving as the basis for several other correlation measures and algorithms [48, 35, 34].

The fundamental challenge in finding strong multivariate correlations is the immense growth in the amount of combinations of time series that need to be examined (i.e., search space), as the number of time series in the data increases. As the search space consists of possible *combinations* of objects, it logically grows at a binomial rate. To illustrate, a small data set of 100 vectors already implies a search space of 1 million candidates when searching for ternary high correlations, whereas finding quaternary high correlations on 1000 vectors involves 1 trillion combinations. Unfortunately, apriori-like pruning techniques cannot be applied to the general case of multivariate correlations. Consider, for example, the three time series presented in Figure 1.1(a), which represent closing prices of three stocks from the Australian securities exchange. Here, we see that all pairwise correlations are relatively low, whereas the artificial time series created by summing QAN and RDF is strongly correlated to MCP. This indicates that pairwise correlation values do not provide sufficient information to reason about whether these vectors can participate in a higher-order correlation. At the same time, an exhaustive algorithm that iterates over all possible combinations implies combinatorial complexity, which does not scale well to large datasets. Therefore, smart algorithms are needed that can drastically prune the search space to reduce computational complexity.

Past algorithms attempted this through one of the following approaches: (a) by focusing on definitions of multivariate correlations that enable apriori-like filtering [3, 34, 48], (b) by constraining query results (e.g., only consider time series combinations with low pairwise correlations), which are only applicable to specific application scenarios [2, 3, 48], or, (c) by developing approximation algorithms, often without any provable guarantees on the completeness of the query answer [2, 3]. Consequently, even though these algorithms are still very useful for their particular use cases, they are not relevant for general use.

Furthermore, literature on efficient multivariate correlation detection has only focused on *one-shot* computations over static data; data that is made available at the beginning of the algorithm and does not change during its runtime. Such techniques are inadequate for real-time event monitoring applications that involve analysis of *data streams* [13]. Typical examples include flash-trading models (where early discovery of irregularities in the market can help traders spot investment opportunities [49, 33]), weather sensor networks (where measurements must be monitored and analyzed for detection of anomalous events such as storms and floods), and network monitoring systems (where usage information must be tracked to timely identify weak spots and DoS attacks). In such contexts, continuous maintenance of query answers is required instead of one-shot responses to occasional queries [16, 41].

This study takes a more general approach to mining multivariate correlations as well as extending the problem to a streaming setting. We present a streaming extension to a *one-shot* algorithm for mining high multivariate correlations called *Correlation Detective* (abbreviated as CD) [29]. In contrast to other work on multivariate correlation discovery, CD considers correlation measures that are, by nature, not suitable for apriori-like pruning. It prunes the search space by identifying clusters of vectors, and handling all vectors within the cluster as a single entity. Further, CD prioritises discovery of the less complex multivariate correlations – the ones that contain the smallest number of vectors – to foster more intuitive and interpretable results. The streaming extension (named CDStream) naturally inherits all of these features. Different algorithmic variants of CDStream are considered: an exact threshold variant that monitors and returns all correlations higher than a threshold τ, and an exact top-k variant that returns the top-k highest correlations. Additionally, we present an extension to CDStream called CDHybrid, which adaptively chooses between CD and CDStream for updating the query answer based on the properties of the input stream.

We evaluate our algorithms on five datasets for a wide range of query parameters, and compare them to CD and current state-of-the-art one-shot algorithms. Our evaluation demonstrates that CDStream outperforms CD by at least an order of magnitude in unexceptional situations, which in turn outperforms the state-of-the-art by approximately the same amount. Also, we show that the CDHybrid effectively orchestrates CD and CDStream with marginal switching cost, offering a more context-robust solution.

1.2 Context of the Research

This work was part of an overarching research initiative on multivariate correlation discovery algorithms broken up into a collection master thesis projects (running in parallel). The current study, focused on designing an algorithm suitable for streaming data, was linked to the research that led to the development of CD [29]. Although the work on CD served as the basis for CDStream, the main artifact of this study, it is important to note that the main author (J.E. d'Hondt) does not claim (co-)ownership of the work on CD as part of his master thesis. He merely supported the research on CD through regular discussions, challenging the ideas of the main researcher (Koen Minartz). Moreover, the choice for CD as the initialization algorithm of CDStream was an independent design decision. It was not set as a requirement prior to the study. The author could have gone a totally different road if it led to a potentially better result.

A joined version of the work on CD and the current is under submission for the 48th International Conference on Very Large Data Bases [43]. The technical report of this submission can be found in our github [2].

1.3 Outline

This work is structured as follows. In Chapter 2, preliminary knowledge is covered which is essential to understand the problem definition, discussed in the same chapter. The problem definition includes discussion of the possible query types and constraints considered, along with a description of the streaming model that will be employed. Chapter 3 discusses related work on bivariate and multivariate correlations, including an overview of the current state-of-the-art algorithms for both static and streaming data. Furthermore, the chapter includes an in-depth

[2]https://github.com/JdHondt/CorrelationDetective

description of the Correlation Detective algorithm, which will serve as the basis for this work. Next, Chapter 4 presents the proposed extension to CD through illustrating the prior's shortcomings in a streaming context. Then, Chapter 5 describes how a combination of the two algorithms can be created which adapts to sudden events and changes in that context in order to improve robustness of the solution. In Chapter 6, evaluation and comparison of the proposed algorithms is covered. Finally, Chapter 7 introduces the final conclusions of this study and evaluates its limitations.

Chapter 2

Preliminaries and Problem Formulation

We start with a discussion of the multivariate correlation measure that we will be considering in this work; the Multiple Correlation measure. We then discuss the methods for handling streaming data, particularly how meaningful correlations can be calculated over data that is updated *continuously* and *asynchronously* among time series in a *non-distributed* manner. Finally we formalize the research problem.

2.1 Multiple Correlation Measure

This study focuses on a multivariate correlation measure named multiple correlation. Given two sets of vectors X and Y, multiple correlation is defined as follows:

$$\mathbf{mc}(X, Y) = \rho\left(\frac{\sum_{\mathbf{x}\in X}\hat{\mathbf{x}}}{|X|}, \frac{\sum_{\mathbf{y}\in Y}\hat{\mathbf{y}}}{|Y|}\right) \tag{2.1}$$

where ρ denotes the Pearson correlation coefficient and $\hat{\mathbf{x}}$ denotes $\mathbf{x}$ after z-normalization, i.e., $\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_{\mathbf{x}}}{\sigma_{\mathbf{x}}}$ with the average and standard-deviation of $\mathbf{x}$ as $\mu_{\mathbf{x}}$ and $\sigma_{\mathbf{x}}$, respectively. Intuitively, $\mathbf{mc}$ takes the element-wise means of the z-normalized vectors in X and Y, and computes the Pearson correlation of the result. The aggregation method of choice here is averaging, but note that both the definition and our work can be easily extended to other linear aggregation methods such as weighted average.

2.2 Handling Streaming Data

Following [49], our approach begins by distinguishing three time periods, from smallest to largest:

- *timepoint* - the smallest unit of time over which a system collects data, e.g., second.
- *basic window* - a consecutive subsequence of timepoints over which a system maintains an aggregated value called a *digest*, e.g., a few minutes. The size of the basic window is assumed to be fixed and pre-defined in this work.
- *sliding window* - a consecutive subsequence of timepoints over which the user queries statistics, e.g., an hour. For example, the user might ask, "which pairs of stocks were correlated with a value of over 0.9 for the last hour?". The size of the sliding window is assumed to be fixed and pre-defined in this work.

The necessity of breaking-up a data stream into intermediate time intervals that we call basic windows will become apparent in the following subsections.

Time Series Data Streams

We consider data arriving as a time-ordered series of tuples (i, timepoint, value), with i being the index of the stream. We do not assume *synchronous streams*, i.e., it is possible that only a subset of the streams receive a new value during a periodic time interval, e.g., a second. A stream with maximum arrival rate will have a new value available at every timepoint. Arrival rates do not have to be fixed for each stream. For example, it is possible for a single stream to receive a new value 1 second after receiving the preceding value, while subsequently receiving no new values for a period of 10 seconds. If a stream receives multiple values during a timepoint, then a summary value will be assigned to that timepoint (e.g., the value received last). Let $s[i]$ denote the value of stream s at timepoint i. If s did not receive a value at timepoint i, $s[i]$ will be ignored when computing statistics over a period including timepoint i. $s[i : j]$ denotes the values of stream s received between timepoints i and j inclusive. s_i denotes a stream with index i. We use t to refer to the latest timepoint at a certain moment, i.e., now.

Temporal Spans

While data streams are assumed to be virtually infinite, people are generally only interested in statistics on data gathered over a certain temporal span, e.g., over the last few days. We define the most common temporal spans over which statistics of stream time series are calculated.

- **Landmark windows:** Statistics are computed based on the values received between a fixed timepoint called *landmark* and the present. For example, when computing the moving average of a stream time series s from a landmark k up to and including timepoint t, one would compute $\frac{1}{t-k+1}\sum_{i=k}^{t} s[i]$.
- **Sliding windows:** This temporal span can be considered a landmark window with a landmark that shifts with time. Given the length of the window w and the current timepoint t, statistics will be computed over the subsequence $s[t-w+1 : t]$. For example, the moving average over a period w is computed as $avg(s[t-w+1 : t]) = \frac{1}{w}\sum_{i=t-w+1}^{t} s[i]$. This window is most used for financial applications as it allows to move with short-term trends.
- **Decaying window model:** Can be considered a weighted sliding window model with decreasing weights for older data points. For example, in contrast to the computation of $avg(s[t-w+1 : t])$ with a normal sliding window, the moving average in a damped model can be computed as

$$avg_{new} = avg_{old} * \alpha + s[t] * (1-\alpha), \alpha \in [0, 1]$$

In this work, we will be using the sliding window model, as it is the most general and widely-used alternative. For conciseness we denote $s_i[t-w+1 : t]$ as $\mathbf{v}_i^t$ (bold-faced $\mathbf{v}$ referencing a vector of fixed dimensions).

2.2.1 Correlations over asynchronous streams

We do not assume streams to be synchronous. This decision is not obvious. In fact, the vast majority of work on monitoring (pairwise) correlations over stream time series requires that streams are synchronized or artificially synchronizes streams using imputation techniques (e.g., interpolation) [13, 20, 22, 48, 49]. The reasons for this assumption are often non-explicit or unclear. However, the decision does become more sensible when considering the complications that come with handling asynchronous streams, to be discussed shortly.

In reality, streams are typically not synchronized. For example, security prices are updated based on the agreed value from one successful trade to the next (i.e., ticks). These updates are irregular within a stream, thus also resulting in asynchronization between streams of different securities. Also, consider analyzing sensor data from sensors spread globally, potentially having different manufacturers or measuring different data. Then, it is likely those sensors will have different measurement intervals which will also result in asynchronization. If one wishes to perform linear correlation analysis on this data, he/she will have to find a way to 'line up' data points such that points gathered at (approximately) the same time can be compared. Again, the most common solution for this is imputation of missing values. When applying this to the case of sliding windows, this means that all windows will be slid when moving to a new timepoint, either observing an actual new value or an imputed one. We refer to the process of sliding all windows simultaneously as a *global update*. Likewise, we refer to the process of sliding only a single window as a *local update*.

Performing global updates on naturally asynchronous streams with sliding windows has the danger of leading to spurious correlations. This is because imputed values give a false sense of similarity between time series, especially between time series that have imputed values at the same timepoints. For example, consider a stream s_1 with values $\langle \dots, 1, 10, 20 \rangle$ for the last 3 seconds, and stream s_2 with values $\langle \dots, 1, 1, 1 \rangle$. Clearly, when using a sliding window of size $w = 3$, the correlation between s_1 and s_2 will be around zero as s_2 does not report the same relative increase in values as s_2. Though, if both streams do not receive updates for the next 3 seconds, and we impute values using *forward-filling*, their correlation will grow to 1. While correlations may indicate high similarity, this conclusion is solely based on the absence of data in the two streams, while that absence might have been caused by totally different factors. The likelihood of spurious correlations only increases with the level of dissimilarity between arrival rates of streams (i.e., level of asynchronization).

Employing a local update model for monitoring correlations over asynchronous streams, however, is controversial as it will lead to time-misalignments between windows, effectively computing lagged correlations instead of synchronized correlations. This will deteriorate the meaningfulness of correlations as the time lag of correlations will not be constant over time or fixed between streams. Logically, this also holds for multivariate correlations as a correlation $\mathbf{mc}(\mathbf{v}_a, (\mathbf{v}_b, \mathbf{v}_c))$ at timepoint t will not mean the same the correlation at $t + 1$ if only $\mathbf{v}_c$ receives an update. As we look to develop a general method that works in most situations, we require a streaming model that allows processing of both synchronous and asynchronous updates while preserving the meaningfulness and reliability of the results.

Similar to the work of Zhu [49], we will do this by discretizing the time series into sequences of aligned basic windows, maintaining digests (i.e., summary values) of the values received within those windows (see Fig. 2.1). Then, correlations can be computed over the digests instead of over the raw (imputed) values. Note that this means that sliding windows will also be taken over basic windows instead of timepoints. Digests of the running basic window will be updated as data arrives using an aggregation method appropriate for the data context. When a running basic window ends, all windows will be slid and a new basic window is started with an appropriate starting value (e.g., latest price for stock data). A good value of the size of the basic window b_{bw} would be the smallest average arrival rate among streams, such that on average every basic window will contain at least one original value, which limits the amount of aggregation and global updates over time. Note that this method is effectively an online approach to resampling time series. The discretized version of stream s_i will be denoted as S_i, with the sliding window (of basic windows digests) $S_i[L - w + 1 : L]$ denoted as $\mathbf{V}_i^L$ (using L to indicate the index of the running basic window). The symbol w will still be used to refer to the sliding window size, even though it now contains at most b_{bw} original values.

2.2.2 Batching Models

Batch processing of incoming data is often perceived as the counterpart of stream processing. In batch processing, data is collected over a relatively large period of time (e.g., a day) and processed in bulk at the end of the period. This method is most useful for streams of data that are not time-sensitive and/or require complex processing. Stream processing, on the other hand, involves processing data piece-by-piece which is useful in cases where processing is fast and results are needed immediately. Streaming algorithms are by definition the solutions that fall into the latter category. However, streaming algorithms often still involve some form of batching with

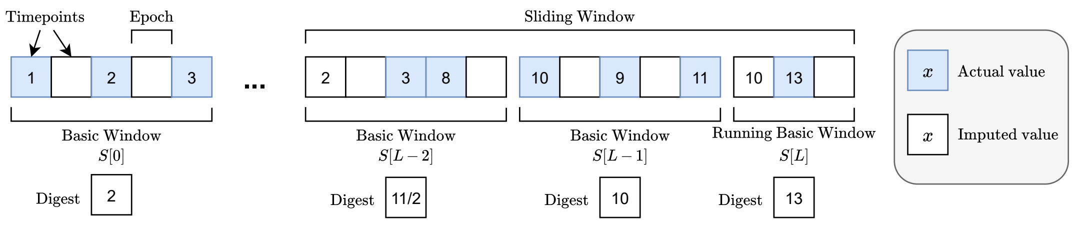


Figure 2.1: Example of discretized stream with $w = 3$, $b_{bw} = 5$, average aggregation and last-average imputation

their arrivals, albeit on a much smaller scale. This is to benefit from some efficiency gain that comes from handling a group of updates (e.g., avoiding double work), or because it is necessary considering the context (e.g., correlation monitoring over synchronized streams). Batching is also favourable in the current case as it may open doors to additional optimizations, and updates may cancel each other out leaving correlations unchanged. This work considers two main ways of batching updates;

- **Time-based batching:** Batches involving all data arrived throughout a sequence of one or more timepoints (e.g., 5 seconds). This implies the query result is updated at a fixed time-interval.

- **Arrival-based batching:** Batches involving a fixed amount of arrivals. This implies the query result is updated based on the rate updates come in.

The batch-size (either a time interval or a number of arrivals) determines the update rate of the result set. Thus, it should be carefully set in consideration of both context and algorithm capacity. Note that basic windows always span a fixed amount of time, also if an arrival-based batching model is employed. In such a case, if the time of the basic window has been passed, the algorithm will move the sliding window as soon as the running batch has been finished.

In this work, we will mainly be using a *time-based* batching model as this guarantees that updates will always contribute to basic window they arrived in. In contrast, using an arrival-based model enables the algorithm to place updates in later basic windows if they were received during processing. For example, consider the situation that the algorithm receives a batch of updates that takes a significant amount of time, resulting in incoming data queuing up during processing. Then, when the algorithm is finally done, it first needs to move the sliding window as an interval of b_{bw} has expired. Subsequently, the data arrived during processing will be added to the new basic window while it was received earlier. If this happens, correlations will no longer be in line with our formulated definitions and will break down the meaningfulness of our results.

We will also be doing experiments employing the *arrival-based* model to show the solution's sensitivity to batching models and the batch-size.

2.3 Problem Definition

Consider a set of asynchronous streams $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ and their discretized variants $\mathcal{V} = \{\mathbf{v}_i \in \mathbb{R}^w\}_{i=1}^n$ with windows of size w, receiving new data in a non-distributed manner. Then, the goal of this work is to find the result set R_t containing all subsets of $\mathcal{V}$ that have a high **mc** coefficient, satisfy the additional constraints, and contain data arrived up to timepoint t. User queries need not to have a latency larger than the time required to fill one batch (i.e, time period for time-based batching, number of arrivals for arrival-based batching). We call the period in which a batch of updates is processed an *epoch* (see Fig. 2.1).

Users may be interested in all subsets that have a **mc** coefficient higher than some threshold

$\tau \in [-1, 1]$, or merely the subsets with the highest **mc** coefficients. [1] Therefore, we consider two query types:

Query 1: Threshold query For a user-chosen correlation correlation threshold τ, and parameters $l_{\max}, r_{\max} \in \mathbb{N}^+$, find all pairs of sets $(X \subset \mathcal{V}, Y \subset \mathcal{V})$, for which $\mathbf{mc}(X, Y) \geq \tau$, $X \cap Y = \emptyset$, $|X| \leq l_{\max}$ and $|Y| \leq r_{\max}$.

Query 2: Top-k query For a user-chosen integer parameter k, and parameters $l_{\max}, r_{\max} \in \mathbb{N}^+$, find the k pairs of sets $(X \subset \mathcal{V}, Y \subset \mathcal{V})$ that have the highest values $\mathbf{mc}(X, Y)$, such that $X \cap Y = \emptyset$, $|X| \leq l_{\max}$, and $|Y| \leq r_{\max}$.

The combination of $l_{\max}$ and $r_{\max}$ controls the desired complexity of the answers. Smaller $l_{\max} + r_{\max}$ values yield results that are easier to understand, and more useful to the data analyst.

Complementary to the two query types, users may also want to specify a set of additional constraints. Typically, these constraints relate to the targeted diversity of the answers. We will consider two different constraints:

Irreducibility constraint For each (X, Y) in the result set, there exists no (X', Y') in the result set such that $X' \subseteq X$, $Y' \subseteq Y$, and $(X', Y') \neq (X, Y)$. Intuitively, if $\mathbf{mc}(X', Y') \geq \tau$, then no supersets of X' and Y' should be considered together. This constraint prioritizes smaller answers, which are more easily explainable.

Minimum jump constraint: For each (X, Y) in the result set, there exists no (X', Y') such that $X' \subseteq X$, $Y' \subseteq Y$, $(X', Y') \neq (X, Y)$, and $\mathbf{mc}(X, Y) - \mathbf{mc}(X', Y') < \delta$. This constraint, which was first proposed in [2], prioritizes the solutions where each time series in X and Y contributes at least δ to the increase of the correlation.

The minimum jump constraint applies to both query types. The irreducibility constraint, on the other hand, is useful only for threshold queries. This is because the irreducibility constraint is ill-defined for top-k queries. For example, consider vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ packed into sets $X = \{\mathbf{a}\}$, $Y = \{\mathbf{b}, \mathbf{c}\}$, $X' = \{\mathbf{a}\}$, $Y' = \{\mathbf{b}\}$, with correlations $\mathbf{mc}(X, Y) = 0.9$, and $\mathbf{mc}(X', Y') = 0.8$. Then, if one queries the top-1 vector combinations with an irreducibility constraint, it is unclear which vector combination should be in the result set; top-k states that the highest correlations should be in $\mathcal{R}$ (i.e., (X, Y)), while irreducibility states that the smaller combinations have priority (i.e., $\mathcal{R} = \{(X', Y')\}$). To avoid such conflicts we refrain from considering such query definitions.

For conciseness, we will denote the combination of the set sizes as $\mathbf{mc}_{\text{size}}(l_{\max}, r_{\max})$. We will call this a *correlation pattern*. For example, $\mathbf{mc}_{\text{size}}(2, 1)$ will identify the combinations of sets of time series with one or two time series on the left-hand side and one on the right-hand side, with high **mc** correlation. Finally, we will denote a particular combination of time series by displaying the time series, grouped by parentheses. For example, $(\mathbf{v}_1, (\mathbf{v}_2, \mathbf{v}_3))$ denotes a set combination of correlation pattern $\mathbf{mc}_{\text{size}}(1, 2)$, where time series $\mathbf{v}_2$ and $\mathbf{v}_3$ are aggregated together.

[1] Note that for some applications it is also interesting to identify subsets with multivariate correlations *lower* than some threshold, or within some range of values (e.g., negative correlations between stock prices are relevant for portfolio diversification). While we do not focus on such thresholds in this work, the theory and methods that are presented can straightforwardly be extended to the above result set definition (further discussed in Chapter 7). These extensions were left out of the scope of this study in view of time-constraints.

Chapter 3

Related Work

Correlation analysis is commonly used in Exploratory Data Analysis, and several algorithms exist for finding highly correlated sets of time series, considering both static and streaming data. The following sections present the most prominent algorithmic techniques in bivariate and multivariate correlation discovery through discussing the most influential work in the field. We start with the discussion of relevant work on simple correlation discovery for streaming time series. As also noted in the previous chapter, all presented work assume streams are synchronized. Next, we focus on the current state-of-the-art for multivariate correlation discovery. These algorithms currently only support static data. Lastly, the unpublished Correlation Detective algorithm is described in appropriate detail, as it will serve as the basis for the proposed algorithm discussed in later chapters.

3.1 Streaming algorithms for pairwise correlation discovery

The following section describes relevant work on simple correlation discovery for streaming time series. We grouped the work based on the techniques used to prune the respective search space, and focus on the rationale behind those techniques to infer whether they can be applied to our context.

Discrete Fourier Transformations In 2002, Zhu and Shasha presented StatStream, a non-distributed streaming algorithm supporting maintenance of threshold queries for pairwise correlations [49]. One of the core contributions of StatStream is a monotonic one-to-one mapping of Pearson correlation to Euclidean distance, which opens doors to dimensionality reduction techniques. Specifically, given that Euclidean distance is preserved during Discrete Fourier Transformation (DFT) of vectors, they use DFT to reduce the dimensionality of vectors such that these can be indexed in a low-dimensional grid, where all highly correlated pairs end up in neighboring cells. As such, high correlations are identified by finding all neighboring pairs in the grid. At every (global) update, DFT approximations are updated incrementally and re-indexed in a clear grid. StatStream also supports lagged correlations and autocorrelations by keeping pointers to outdated windows of streams in the grid for a certain sequence of timepoints. The method's evaluation shows that the approximation of correlations using DFTs yields significantly better results compared to an exact computation of correlations using the raw data; the computation is faster, allows more streams to be handled, and has no false negatives.

Later, Mueen et al. extended the methods of Zhu by resorting to dynamic programming to reduce the number of comparisons and proposing methods for optimizing I/O costs through graph partitioning [32]. In contrast to Zhu et al., Mueen considers ad-hoc queries, where all data is stored on disk and a target set of streams and target sliding window size are set by the user during the query time. They argue that this implies that no pre-computed index can be created for fast correlation computation because of the large overhead and inefficiency that would come with supporting ad-hoc definition of sliding windows and other hyperparameters. Also, on-disk storage of time series leads to high I/O costs because, due to limited memory, data may need to be read from disk to memory multiple times. Their method solves these problems by approximating correlations using the first few DFT coefficients of time series, and using those results to create batches of time series that have approximate high correlations within the batch, and approximate low correlations with time series from other batches. Then, exact correlations are computed per batch, which minimizes I/O cost as only the time series within that batch have to be read into memory as correlations with other time series are proven to be low. Experiments show that this their algorithm is 17 times faster than that of Zhu et al., and other state-of-the-art exact solutions.

Locality Sensitive Hashing (LSH) LSH is an algorithmic technique where similar input objects (in this case windows of time series) are hashed into the same buckets with high probability [21]. The amount of buckets is heavily subordinate to the universe of input items. This way, it can be seen as a dimensionality reduction method which preserves relative distances between items, similar to the method of Zhu and Shasha. Driven by the lack of work on approximate similarity search algorithms for stream time series, Lian et al. propose an algorithm using weighted locality sensitive hashing (WLSH) to answer approximate range queries over time series [22]. An example of a similarity search query is asking "give me all streams similar to stream i". These queries are relevant to our context as correlation coefficients are also considered a measure of similarity, meaning that querying similar streams could effectively be mapped to querying streams that have high correlation with another target stream. Their method involves transforming windowed time series into a bit vector that is subsequently hashed by randomly selecting k positions in the bit vector with *weighted probability*. This is done such that vectors of high approximate similarity (small L_1 distance) have the same key with adjustable probability. Updates are handled by incrementally updating hash functions which limits down-time of the query system.

Sketches Sparked by the work of Zhu, Cole et al. argue that dimensionality reduction techniques like DFT and Wavelet transforms only work for time series with a fundamental degree of regularity [7]. They state that these techniques "do not work for time series in which the energy is spread over many frequency components, thus resembling white noise". A prototypical example of such time series are stock market returns; changes in price from one timepoint to the next, divided by the initial price. As the energy is spread over many frequency components for such time series, correlations cannot be reliably estimated based on the first few Fourier/Wavelet coefficients, which makes up the core of the techniques of Zhu and Mueen [32, 49].

To cope with this issue, Cole proposes a sketch-based approach to obtain data reduction and uses it to efficiently query highly correlated pairs. Sketches of time series are generated by taking the inner product of each time series window, with a set of structured random vectors. Then the Johnson Lindenstrass lemma [18] is used to make inferences on the correlations of the original data based on the sketches. Again, sketches are updated incrementally by using running dot products. Note that this method is very similar to LSH for Euclidean or Cosine distance without the notion of buckets. Experiments show that the sketches-based approximations of correlations are significantly more accurate than DFT-based approximations for uncooperative time series. On the other hand, when time series closely resemble a random walk, as for stock price data, DFT-approaches give significantly better precision levels at the same recall as compared with the sketch method. In terms of performance, neither the sketch-based approach and Fourier-based approaches are consistently faster than the other.

The main takeaway of this work is that we ought to be cautious with the usage of Fourier-based transformations when working with uncooperative time series. As the financial domain is considered to be one of our main application domains, it is likely that we will face such scenarios in our context.

Hierarchical Boolean Representation (HBR) Introduced by Zhang et al., HBRs are essentially bit vectors of size w which can be used to quickly assess if pairs of a time series are potentially highly correlated through bit operations [47]. In their work Zhang et al. initialize two HBRs for each time series; (1) a macro variant where each bit indicates if the original value with the same index is above the mean value over the window, (2) a micro variant where bits indicate if the original value was larger than the preceding value. The authors prove that these representations can be used to bound correlations with certain precision. Consequently, non-promising pairs are pruned in a two-step approach by computing bounds with the macro HBR following by bounding with the micro HBR which has tighter bounds. Upon updates, both HBRs are updated incrementally as long as the arrival value does not exceed some slack. Elsewise, representations have to be reconstructed fully.

The author's proof that macro and micro HBRs can be used to bound correlations is build upon the assumption that the time series follow a normal distribution. This assumption is arguably crude, especially given the datasets used in the method's evaluation; an unprocessed stock dataset with daily open prices of companies listed on a Chinese stock exchange, spanning a period of 5000 days, and an artificial dataset consisting 1000 generated random walk sequences. Stock prices are known to follow not normal distributions but log-normal distributions; and only after application of a detrending procedure [4]. One would expect to see at least a minor trend present in the used data, especially considering that it spans a period of over 13 years, and that Chinese markets have shown considerable price increases over the last few decades. Still, results show that HBRs are very effective in pruning uncorrelated pairs of time series early, resulting in a performance competitive with other state-of-the-art algorithms. A possible explanation for these results is that the distribution assumption of the author's is only used to estimate the precision of correlation approximations; it is not instrumental to the proof that correlations can be bounded with HBRs. Consequently, we can conclude that assumption violations on the data distribution will only lead to worse approximations, and thus a lower pruning power. Results also tell us that the realized precision is adequate for the respective use case. Though, we should still consider these caveats when looking to apply these techniques to our context.

Piecewise Aggregate Approximations (PAA) This technique exists in literature under many different names but all works essentially come down to the same concept; to down-sample time series to a sequence of basic windows with digests which serves as an approximate representation of the original data. Tangent to their paper on LSH, Lian et al. applied a hierarchical version of this method for answering pattern matching queries over stream time series [23]. Examples of such queries are "Get me all stocks that recently had a two bottom or head-shoulder pattern in their price", or more specific to this context; "Get me all all streams that recently had the same pattern as the one in $\mathbf{v}_i$". In their method they use what they call multi-scale segment means which are effectively the means of basic windows of different sizes, going as low as $b_{bw} = 1$. Then, they employ a multi-step filtering approach to prune the search space by bounding the distance between a target pattern and a sequence of segment means. If the bounds are both on one side of the threshold, one can make a decision based on the approximate representation. If the bounds are not conclusive, a finer grain of segment means are used. Bounds are provided for all L_p-norm distances, making the method very flexible. Updates are handled by incrementally updating the multi-scale segment means.
Other notable work applying a similar technique is the work of Keogh and Lin named Symbolic Aggregate Approximation (SAX) [26]. Here, Keogh proposes a symbolic representation of time series specifically designed for data mining algorithms. The proposed approximation is particularly useful for data mining algorithms as it allowed both dimensionality reduction and lower bounding of L_p norms (similar to Lian and Zhu). This is favourable as data mining algorithms often suffer from high dimensionality and it allows the algorithms to manipulate the representation while preserving relative distances. It does this by transforming time series into a sequence of PAAs. Next, it discretizes the segments by fitting a Gaussian curve over the original values, dividing it into equiprobable regions and using it to map segment values to regions indexed by letters (hence; *symbolic* representations). They further show that this representation can be used to lower bound L_p-norm distances, which is favourable for data mining tasks such as clustering or even classification. The authors later refute those claims in a paper titled "Clustering of Streaming Time Series is Meaningless" [27]. In the paper, the authors argue that clusters extracted from streaming time series are forced to obey the constraint that the weighted sum of cluster averages must be constant over time, which is pathologically unlikely to be satisfied by any dataset. Because of this,

clusters extracted by any clustering algorithm are essentially random; invalidating the contributions of dozens of previously published papers. Despite this argument, the authors continue to show the usefulness of stream time series clustering for various applications in later work [5, 9, 42]. SAX was later applied to the case of correlation by generating symbolic representations over z-normalized vectors, performing a nearest neighbor search with the Euclidean distance measure [36].

A clear pattern can be observed when analyzing the above techniques. Namely, they are generally extensions to a fast one-shot approach with the addition of incrementally updating the compressed representations or relevant auxiliary data structures. After an update, they again consider all candidates ($\mathcal{O}(n^2)$) without re-using any information related to previous results for pruning. This approach makes sense when considering bivariate correlations, where vector dimensionality is a limiting factor, making a case for dimensionality reduction techniques.[1] However, for the case of multivariate correlations, the number of possible combinations increases at a binomial rate with the values of $l_{\max}$ and $r_{\max}$, making the combinatorial complexity the limiting factor. Thus, while dimensionality reduction might speed-up the computation of multiple correlation coefficients, it will only marginally influence the total computation time.

Furthermore, techniques that map vectors to a compressed space where neighborism implies correlatedness (e.g., DFT and LSH) will also not work for multivariate correlations as highly correlated sets do not require correlated pairs within them. In other words, two vectors may have a low pairwise correlation with a third vector, whereas their aggregate may have a high correlation (See, e.g., example of Fig. 1.1).

From this we conclude that the discussed techniques for bivariate correlations are inapplicable for the multivariate case. Nevertheless, the overarching method of incrementally updating the artifacts of a fast one-shot algorithm could be useful if such an algorithm can be found.

3.2 Multivariate correlation algorithms

The following section focuses on the current state-of-the-art for multivariate correlation detection. The family of multivariate correlation measures is small. Still, we limit ourselves to discussing the state-of-the-art algorithms for correlation measures that lie close to our formalized problem. The work is grouped based on the employed correlation measure. Note that all work to date considers one-shot approaches to discovering correlations without discussing possible streaming extensions. We are thus limited to multivariate correlation algorithms on static data.

Multiple Correlation Prior work on multivariate correlations often rely on additional constraints for their pruning power. To illustrate, Agrawal et al. investigate the problem of finding highly-correlated tripoles [2]. A tripole is effectively equivalent to the multiple correlation measure of pattern $\mathbf{mc}_{\text{size}}(2, 1)$. The authors propose two algorithms for finding strongly correlated tripoles that comply to the minimum jump constraint (as defined in Section 2). One of the algorithms, called CONTRaComplete, prunes the search space by deriving bounds on the correlation of two vectors, such that it is impossible for the pair to exist in a triplet and comply to the minimum jump constraint if their pairwise correlation is not within these bounds. Logically, CONTRaComplete initially computes all pairwise correlations, and filters out all pairs that do not comply to the bound. The algorithm subsequently constructs triplets from the pairs that passed the filter, and evaluates their correlation. As the bounds are based on the definition of the minimum jump constraint, this constraint is naturally indispensable in all queries.

The second algorithm, named CONTRaFast, introduces an additional filtering step to remove redundant tripoles from being enumerated, that provides further reduction of the computational effort for the cost of completeness of the result set. The algorithm additionally prunes vector pairs in the filtered set that have a high correlation with any other pair in the filtered set. Formally,

[1]One might argue that pruning candidates is still relevant for bivariate correlation detection as it limits the amount of correlations that need to be computed. However, candidate pruning often involves creating complex indices that come with considerable overhead. The cost of computing correlations for these candidates will likely not exceed the overhead that comes from potentially pruning some of them.

given a pair (T_A, T_B) that complies to the constraint defined in CONTRaComplete, if there exist another pair in the filtered set (T_X, T_Y) such that $(T_A, T_X) \geq \kappa \vee (T_B, T_Y) \geq \kappa$. In such cases, both vector pairs are disregarded. Experiments show that CONTRaFast reports 99% of all interesting tripoles within 100 minutes on a dataset with sensor data from with 10 000 different sensors. The authors further discuss the fysical interpretation of an interesting tripole, and demonstrate how they can lead to novel insights through several case studies.

While the work on tripoles show promising applications of multivariate correlations, the techniques are limited to correlation patterns with a cardinality of three, cannot guarantee completeness, and rely heavily on the minimum jump constraint. Since we look to provide more freedom to the user through allowing correlation patterns of arbitrary cardinalities and optional usage of additional constraints, the algorithms cannot be used as a reference.

Multipoles As a follow-up on tripoles, Agrawal et al. extended the measure to arbitrary correlation patterns by introducing a novel multivariate correlation measure, the multipoles measure. The metric measures the linear dependence of an input set of vectors $\mathbf{X}$. Specifically, let $\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_n$ denote n z-normalized input (column) vectors, and $\mathbf{X} = [\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_n]$ the matrix formed by concatenating the vectors. Then:

$$\textsc{Multipoles}(X) = 1 - \min_{\mathbf{v} \in \mathrm{R}^n, ||\mathbf{v}||_2 = 1} \mathrm{var}(\mathbf{X} \cdot \mathbf{v})$$

The measure takes its maximum value 1 when there exists perfect linear dependence, i.e., there exists a vector $\mathbf{v}$ with norm 1, such that $\mathbf{X} \cdot \mathbf{v}$ has zero variance. Notice that multipoles and multiple correlation are not equivalent, and the one is not a generalization of the other. By definition, multipoles assumes optimal weights (vector $\mathbf{v}$ is such that the variance is minimized), whereas for the case of multiple correlation, the aggregation function for the vectors (e.g., averaging) is determined at the definition of the measure and is independent of the vectors. Furthermore, multipoles expresses the degree of linear dependence within a single set of vectors, whereas for multiple correlation, two distinct, non-overlapping vector sets are considered. Agrawal et al. propose in their paper two approximate algorithms for finding vector sets with high multipole values. Both algorithms use clique enumeration to efficiently explore the search space of possible vector combinations. Their performance relies on a parameter that trades off completeness of the result set for performance. Again, the minimum jump constraint is instrumental to reduce computational effort. The proposed algorithms yield significantly more complete results compared to baseline methods that use l_1-regularization based techniques or methods from structure learning. Still, both algorithms do not come with completeness guarantees.

Total Correlation Total correlation is a non-linear information-theoretic metric that expresses how much information is shared between variables [17]. Nguyen et al. [34] proposed a solution to finding strongly correlated groups of columns in a database considering a correlation measure closely related to Total Correlation. Their key concept is the initial evaluation of pairwise correlations, which can then be used to lower bound the total correlation of a group of vector. Next, their algorithm focuses on identifying quasi-cliques with high pairwise correlations, which thus also yield a high total correlation value. Despite the ability to discover interesting sets of vectors, the algorithm fails to identify the groups with low pairwise correlations which do have a high total correlation as a whole. These are arguably the most interesting cases as these relations will not have been detected by bivariate correlation analysis and will thus uncover potentially novel insights. In line with this deficiency, the method also does not provide completeness guarantees.

Subset Regression In the supervised learning context, subset regression is related to multivariate correlation mining. The goal of this feature selection problem is to select the best p predictors out of n candidate features [8]. Heuristics like forward selection, backward elimination or genetic algorithms are commonly used to find good solutions [31, 15]. Our problem differs from the above in two aspects. First, we aim to find interesting patterns in the data, instead of finding the best predictors for a given dependent variable. Second, instead of finding only the single highest correlated set of vectors, our goal is to find a *diverse set* of results. Showing multiple diverse results to the domain expert helps her to further assess the results on qualitative aspects and to gain more insights.

We can conclude from this discussion that none of the current state-of-the-art methods are applicable to the problem definition of this study. The proposed algorithms cannot be extended to our streaming setting as they all apply to slightly different problems; they either include approximation algorithms (e.g., the reported work Tripoles/Multipoles/Total Correlation detection) or involve a dissimilar definition of the desired answer (e.g., Subset Regression). Fortunately, affiliate research on the Correlation Detective algorithm showed promising results while this research was conducted. In contrast other multivariate correlation algorithms, this algorithm could serve as the basis for a solution as it applies to an identical problem definition (excluding the streaming setting) and outperforms the rest by at least an order of magnitude [29]. Though, we should be weary of the potential dangers of clustering streaming time series proposed by Lin and Keogh [27] when extending CD to a streaming context, as it relies heavily on the hierarchical clustering of time series.

3.3 Correlation Detective: a one-shot approach to Multiple Correlation discovery

The following section describes a recently developed one-shot approach to discovering strong multivariate correlations that looks out to be more general and efficient than the current state-of-the-art described in Section 3.2. As the algorithm assumes the input data to be *static*, we do not have to consider incoming data and windows, making the discussion overly complicated. Instead, we will be referring to the data as a set of static vectors $\mathcal{V}$ to preserve generalization and aid interpretability.

As discussed before, the main challenge in detecting strongly correlated vector sets stems from the combinatorial explosion of the number of possible combinations that need to be examined. Specifically, in a dataset of n vectors, there exist at least $\mathcal{O}\left(\sum_{i=2}^{p}\binom{n}{k}\right)$ possible combinations, where $p = l_{\max} + r_{\max}$ denotes the maximum number of vectors that can participate in a correlation. This means that, even if each possible combination can be checked in constant time, the enumeration of all combinations will still take significant time. CD combats this complexity through the observation that vectors containing data from similar contexts (e.g., stock prices of STMicroelectronics and ASML) often exhibit relatively high correlations between them (recall Fig. 1.1). CD exploits these correlations by clustering the respective vectors together, making decisions on the collective instead of the individual to drastically prune the search space. More specifically, instead of enumerating over all possible combinations of vectors complying to a correlation pattern, CD clusters vectors based on their pairwise ρ, and enumerates over the combinations of the cluster centroids, which are generally several orders of magnitude less. For each of these combinations, CD computes upper and lower bounds on the correlations of all vector combinations in the Cartesian product of the clusters. Based on these bounds, CD decides whether or not the combination of clusters (i.e., all combinations of vectors derived from these clusters) should already be added to the result set, can safely be discarded, or, finally, if the clusters should be split further into smaller subclusters, for deriving tighter bounds. This effectively reduces the number of combinations that need to be considered, making CD an order of magnitude faster than the competitive algorithms.

In the remainder of this section, we will discuss in detail how CD handles the query types and additional constraints proposed in Section 2.3. For conciseness we will limit ourselves to the elements relevant to its streaming extension. A more in-depth description of the algorithm (including its individual evaluation) can be found in the technical report [29]. We will start with a brief description of the initialization phase, which includes data pre-processing and the clustering algorithm. In the later subsections we will describe how CD answers threshold and top-k queries respectively.

Initialization phase. As a pre-processing step, CD z-normalizes all vectors in order to obtain a common scale for all signals. This is essential in situations where signal values differ significantly.

For example, consider analyzing stock data which includes the prices of penny stocks as well as stocks like Amazon. Then, when considering the **mc** value of a pattern including a stock of 1 euro and one of 1000 euro on the same side of the pattern, we lose all information of the cheap stock when taking the element-wise average of the two stocks. Consequently, we are effectively computing a bivariate correlation instead of a multivariate correlation value. Next, CD hierarchically clusters the data based on pairwise correlations using an algorithm inspired by the K-Means++ algorithm [46]. In contrast to typical K-Means clustering which stops at convergence, CD makes sure that the full cluster-tree is created, meaning that the leaves only consist of clusters with a single vector (called *singleton clusters*). The pairwise correlations computed during the clustering stage are cached, such that they can be reused at later stages of the algorithm.

3.3.1 Threshold queries

In line with the problem definition in Section 2.3, CD receives as input a correlation pattern (e.g., $\mathbf{mc}_{\text{size}}(2,1)$, restricting the correlations to types $(\mathbf{x},\mathbf{y})$ and $((\mathbf{x},\mathbf{y}),\mathbf{z})$), a correlation threshold τ, and the cluster tree produced in the initialization phase. It will then start from the children of the root cluster, forming all possible combinations of the correlation pattern with these. In the example of Fig. 3.1, which will be used as a running example throughout the thesis, if the desired correlation pattern is $\mathbf{mc}_{\text{size}}(2,1)$, the following *combinations of clusters* will be examined in order of increasing pattern length:

$$\forall_{C_x,C_y\in\{C_1,C_2,C_3\}}(C_x,C_y) \cup \forall_{C_x,C_y,C_z\in\{C_1,C_2,C_3\}}((C_x,C_y),C_z)$$

A combination of clusters compactly represents the combinations created by the Cartesian product of the vectors inside the clusters, further called the *materialization* of a cluster combination. For example, assuming that $|C_x| = 4$ and $|C_y| = 3$, the cluster combination (C_x, C_y) represents a set of 12 vector combinations. For each cluster combination, CD computes lower and upper bounds on the **mc** value of the materializations of these clusters, denoted with LB and UB respectively (Alg. 1, line 1). Next, CD compares these bounds with the threshold τ (lines 2,4,6). If $UB < \tau$, the combination is *decisive negative* - any materialization cannot yield a correlation higher than the threshold τ. Therefore, this combination does not need to be examined further. We will further refer to these cluster combinations as *negative decisive cluster combinations*, or DCC^-. If $LB \geq \tau$, the combination is *decisive positive*, guaranteeing that all materializations will have a **mc** value of at least τ. Therefore, all materializations are inserted in the result. We will further refer to these cluster combinations as *positive decisive cluster combinations*, or DCC^+. Finally, when $LB < \tau$ and $UB \geq \tau$, the combination is *indecisive*. In this case, CD (lines 7-11) identifies the cluster with the largest radius $C_{\max}$, and recursively checks all combinations where $C_{\max}$ is replaced by one of its sub-clusters from the clustering tree. In the running example, assume that CD examined an indecisive combination of clusters C_1, C_2, C_3, and C_2 is $C_{\max}$. Then, CD will drill down to the three children of C_2, namely C_6, C_7, C_8, and examine their combinations with C_1 and C_3. This process continues recursively until each combination is decisive.

Derivation of bounds for the multiple correlation measure

In the following, we discuss how CD derives bounds on the **mc** value of combinations of clusters of arbitrary correlation patterns. As you will see, these bounds depend solely on pairwise correlations between the contents of clusters participating in the correlation pattern (e.g., $\rho(\mathbf{v},\mathbf{w})$ with $\mathbf{v} \in C_1$, $\mathbf{w} \in C_2$ participating in $\mathbf{mc}((C_1,C_2),C_3)$. The original work on CD describes two ways of approaching these pairwise correlations; (1) by bounding the pairwise correlations between two clusters using trigonometric functions (referred to as *theoretical bounds*), (2) by computing the exact pairwise correlations of the clusters' contents (referred to as *empirical bounds*) [29]. Evaluation of CD showed that the usage of empirical bounds lead to better performance, despite the cost of computing many pairwise correlations. Therefore, in this work we will only consider the usage of empirical bounds in CD to bound **mc** values.

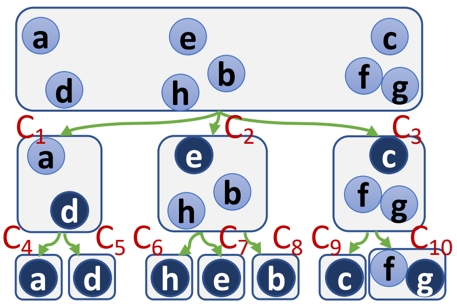


Figure 3.1: Running example in 2 dimensions: the centroids of each cluster are depicted with darker background. All clusters are labeled for easy reference. Source: [29]

```
Algorithm 1: CDThreshold($\mathcal{P}_l$, $\mathcal{P}_r$, $\tau$)
  Input: Sets of clusters $\mathcal{P}_l$ and $\mathcal{P}_r$ that adhere
         to the user-defined correlation
         pattern, correlation threshold $\tau$.
1  $(LB, UB) \leftarrow$ CalcMCBounds($\mathcal{P}_l, \mathcal{P}_r$)
2  if $LB \geq \tau$ then
3      Add $(\mathcal{P}_l, \mathcal{P}_r)$ to the result set
4  else if $UB < \tau$ then
5      Discard $(\mathcal{P}_l, \mathcal{P}_r)$
6  else
       // Replace largest cluster with subclusters
       and recurse
7      $C_{\max} \leftarrow \arg\max_{C \in \mathcal{P}_l \cup \mathcal{P}_r} \{C.radius\}$
8      Set $SC \leftarrow C_{\max}.subclusters$
9      for $S \in SC$ do
10         $(\mathcal{P}'_l, \mathcal{P}'_r) \leftarrow (\mathcal{P}_l, \mathcal{P}_r)$ with $C_{\max}$ replaced
           by $S$
11         CDThreshold($(\mathcal{P}'_l, \mathcal{P}'_r)$, $\tau$)
```

CD uses the following theorem to bound all possible multiple correlation values on *sets of clusters* solely based on the pairwise correlations that exist between those clusters, which it uses to heavily prune the search space.

Theorem 3.3.1 (Bounds for mc) *For any pair of clusters C_i, C_j, let $l(C_i, C_j)$ and $u(C_i, C_j)$ denote the largest/smallest pairwise correlations between the clusters' contents, i.e., $l(C_i, C_j) = \min_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \rho(\mathbf{x}, \mathbf{y})$ and $u(C_i, C_j) = \max_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \rho(\mathbf{x}, \mathbf{y})$. Consider two sets of clusters $\mathcal{P}_l = \{C_i\}_{i=1}^{l_{max}}$ and $\mathcal{P}_r = \{C_i\}_{i=1}^{r_{max}+1}$. Let $L(\mathcal{P}_1, \mathcal{P}_2) = \sum_{C_i \in \mathcal{P}_1, C_j \in \mathcal{P}_2} l(C_i, C_j)$, and $U(\mathcal{P}_1, \mathcal{P}_2) = \sum_{C_i \in \mathcal{P}_1, C_j \in \mathcal{P}_2} u(C_i, C_j)$. Then, for any two sets of vectors $X_l = \{\mathbf{x_1}, \dots, \mathbf{x_{l_{max}}}\}$, $X_r = \{\mathbf{x_{l_{max}+1}}, \dots, \mathbf{x_N}\}$ such that $\mathbf{x_i} \in C_i$, multiple correlation $\mathbf{mc}(X_l, X_r)$, can be bounded as follows:*

$$\text{(1) if } L(\mathcal{P}_l, \mathcal{P}_r) \geq 0: \frac{L(\mathcal{P}_l, \mathcal{P}_r)}{\sqrt{U(\mathcal{P}_l, \mathcal{P}_l)}\sqrt{U(\mathcal{P}_r, \mathcal{P}_r)}} \leq \mathbf{mc}(X_l, X_r) \leq \frac{U(\mathcal{P}_l, \mathcal{P}_r)}{\sqrt{L(\mathcal{P}_l, \mathcal{P}_l)}\sqrt{L(\mathcal{P}_r, \mathcal{P}_r)}}$$

$$\text{(2) if } U(\mathcal{P}_l, \mathcal{P}_r) \leq 0: \frac{L(\mathcal{P}_l, \mathcal{P}_r)}{\sqrt{L(\mathcal{P}_l, \mathcal{P}_l)}\sqrt{L(\mathcal{P}_r, \mathcal{P}_r)}} \leq \mathbf{mc}(X_l, X_r) \leq \frac{U(\mathcal{P}_l, \mathcal{P}_r)}{\sqrt{U(\mathcal{P}_l, \mathcal{P}_l)}\sqrt{U(\mathcal{P}_r, \mathcal{P}_r)}}$$

$$\text{(3) else: } \frac{L(\mathcal{P}_l, \mathcal{P}_r)}{\sqrt{L(\mathcal{P}_l, \mathcal{P}_l)}\sqrt{L(\mathcal{P}_r, \mathcal{P}_r)}} \leq \mathbf{mc}(X_l, X_r) \leq \frac{U(\mathcal{P}_l, \mathcal{P}_r)}{\sqrt{L(\mathcal{P}_l, \mathcal{P}_l)}\sqrt{L(\mathcal{P}_r, \mathcal{P}_r)}}$$

With this theorem, CD can efficiently bound the multiple correlation of any combination of clusters that satisfies the correlation pattern, without testing all possible materializations of this combination. It computes the pairwise bounds on clusters by pre-computing all correlations between pairs of vectors at initialization time, and caching it in an upper-triangular matrix. Then, during the execution of Alg. 1, CD computes $l(C_i, C_j)$ and $u(C_i, C_j)$ by iterating of the contents of C_i and C_j tracking the pairs with the largest/smallest correlations.

Handling additional constraints

For the irreducibility constraint, CD requires to test for any considered combination of clusters whether there exists a (simpler) combinations of any of the subsets of $\mathcal{P}_l$ and $\mathcal{P}_r$ that should be contained in the answers. This is done by discarding the combination $(\mathcal{P}_l,\mathcal{P}_r)$ if a pair of clusters $C_l, C_r \in (\mathcal{P}_l \cup \mathcal{P}_r)$ is found with $l(C_l, C_r) \geq \tau$ during the computation of the **mc** bounds. Note that this accounts only for simpler combinations of cardinality 2. Combinations of higher cardinality are filtered out ad-hoc, before returning the result set to the user. The case of minimum jump is handled analogous, combinations are discarded during execution of Alg. 1 if any $l(C_l, C_r) \geq UB - \delta$ is found. Simpler combinations of higher cardinality are filtered out later.

3.3.2 Top-k queries

Correlation Detective handles top-k queries by treating it as a threshold query with variable τ. It starts with a low estimate for τ, and progressively increases it by observing the intermediate answer set. Key to the algorithm's pruning power is the speed it can approach the true value of τ, the threshold separating the k subsets with the highest **mc** values from the rest. CD does this by considering the candidates in descending order of *estimated* correlation. We provide a high-level description of this process.

First, at the initialization phase, τ is set to the value of the highest k'th pairwise correlation (derived during computation of the pairwise correlation matrix). Then, it processes the possible cluster combinations in two stages. In the first stage, CD continuously updates τ and the intermediate answer while recursively going over possible candidates (i.e., cluster combinations). However, instead of considering the actual UB of cluster combinations, CD artificially lowers the UB to UB_{shrunk} so that only combinations with extremely high UB (i.e., containing materializations with potential of reaching the top-k) are broken up and considered more carefully. Cluster combinations that do not make this cut (i.e., with $UB_{shrunk} < \tau$) are assigned to ordered buckets based on the center of their bounds $\frac{UB-LB}{2}$ with respect to τ, indicating how promising they are. Then, in the second stage, CD iterates over the buckets in descending order, and evaluates the combinations again with their actual bounds to obtain the final answer set. This way, by only spending time on the most promising combinations in the first stage, the value of τ is increased prematurely which increases the pruning power in stage 2.

Chapter 4

CDStream

The following chapter provides an in-depth description of our proposed streaming algorithm for maintaining multiple correlations, named CDStream. We will start with a high-level summary of the algorithm, going into details on each of its components in later sections. The in-depth discussion of the algorithm is structured as follows. First, we describe the solution for maintaining threshold queries without any additional constraints. Next, we informally analyze the base algorithm's computational complexity and storage complexity. Last, the base solution is extended to enable support for the additional constraints and top-k queries.

4.1 General Idea and Intuition

One way to detect multiple correlations on streams is to re-run CD every time a batch of updates arrives, replacing the old result with the new one. However, as we will show in later experiments, this approach is rather inefficient as it does not use any information of previous runs to save computational effort later. CDStream builds on top of CD such that it *maintains* the solution as new data arrives. The following section will provide a high-level description of the followed approach and the rationale behind it. We will do this through consideration of alternative approaches, showing their weaknesses which provides insight on the research process which led to the final solution.

Recall from the previous chapter that many streaming algorithms for correlation monitoring are essentially extensions to a fast one-shot algorithm with the introduction of incremental updates. To date, Correlation Detective is the only one-shot algorithm for multivariate correlation discovery that has the same goals and assumptions as this study. The algorithm efficiently discovers multivariate correlations by clustering the data, and bounding the **mc**-values of combinations of clusters. The result is a set of vector combinations with **mc**-values larger than a threshold τ. These combinations were derived from decisive combinations of clusters, DCCs for short.

An initial approach for our problem could be to incrementally update the **mc**-values of vector combinations, subsequently recomparing it with τ. However, this approach does not address the combinatorial complexity that CD tries so hard to battle. Incremental updating namely speeds-up computation of correlations, but it does not limit the amount of correlations that have to be computed. One could instead argue for an approach where the **mc**-bounds of cluster combinations are incrementally updated. This way, one could continue where CD left off, benefiting from the pruning power of CD without suffering from the computation effort of clustering and the consideration of cluster combinations with indecisive bounds. While valid, the problem with this approach is that **mc**-bounds are not *distributive* over vector addition; roughly meaning that we cannot separate the bounds at $t+1$ into a term with the bounds at t and a term signifying the change (i.e., $\mathbf{mc}(\mathbf{a}+\delta,(\mathbf{b},\mathbf{c})) \neq \mathbf{mc}(\mathbf{a},(\mathbf{b},\mathbf{c})) + \mathbf{mc}(\delta,(\mathbf{b},\mathbf{c}))$ with $\mathbf{a}^{t+1} = \mathbf{a}^{t} + \delta$). This is mainly because of the square root in the denominator of all **mc**-bounds (see Theorem 3.3.1); in case one of the time series involved receives an update, pairwise correlation may change that make up the terms under the square-roots in the denominator. As square roots are not distributive under addition, this change cannot be separated meaning that the bound cannot be 'split' in two. Consequently, one has to reconsider all pairwise correlations that exist among the contents of the clusters, in order to update the $U(\cdot,\cdot)$ and $L(\cdot,\cdot)$ terms in the bound.

4.1.1 Our approach

The observation that multivariate correlations cannot be incrementally updated implies that the bounds of cluster combinations need to be fully recomputed. This is potentially very expensive considering the combinatorial rate at which they can increase, despite being substantially subordinate to the amount of vector combinations they represent. As we desire the algorithm to handle streams with second to millisecond arrival rates, alternative methods are necessary. CDStream does this by pruning the amount of cluster combinations that need to be considered. This approach relies on two main observations;

Observation 1: Most updates do not lead to significant updates on the final result.
For instance, consider tick data of stock prices with millisecond-level granularity. Here, hundreds of trades (i.e., data arrivals) may occur within a second causing the price to change with fractions of a cent, which are marginal fluctuations relative to the daily trend. Therefore, it is likely that pairwise correlations do not change significantly from these single arrivals if the sliding window spans a considerable time (e.g., an hour). The effect of these fluctuations is even more damped for multivariate correlations, as vectors are aggregated.

Observation 2: Streams are not synchronized. Consequently, only a subset time series may be updated meaning that some **mc**-values and bounds might not change at all. Therefore, recomputing the bounds of all cluster combinations can be wasteful.

Considering these observations, substantial effort could be saved by identifying only the combinations that impact the result set when receiving a certain update. Specifically, upon an arrival we would only check the decisive cluster combinations that are in danger of *changing state*, meaning their materializations move in or out of the result set. For a threshold query, these include only DCCs for which the bound closest to τ (i.e., *dominant bound*) moves closer towards the correlation threshold. This includes positive DCCs with a decrease in LB and negative DCCs with an increase in UB.

By analyzing the **mc**-bounds in Theorem 3.3.1, we observe that bounds of cluster combinations depend solely on the minimum and maximum pairwise correlations between all involved clusters. We call the pairs of time series that make up these correlations *extrema pairs*, and argue that changing DCCs can be located efficiently when storing and indexing them based on these pairs, and comparing correlations of extrema pairs to other updated pairwise correlations as new data arrives. As such, CDStream identifies changing DCCs by storing them in an auxiliary data structure called the *DCC Index*. This structure is effectively a nested hash map, with DCCs keyed by multiple variables, including its extrema pairs. As a batch of updates arrives, CDStream uses the DCC Index to locate the DCCs affected by the update, so bounds can be recomputed and changes to the result set can be made if necessary. Section 4.2.1 contains a detailed description of the index, including its rationale and additional optimizations that can be made to improve the algorithm's pruning power.

Instrumental to our pruning technique is the ability to reason about potential changes in multivariate correlations based on changes in pairwise correlations. To this end, we require to update a subset of all pairwise correlations upon arrival of a batch of data. This can be a costly ordeal if the batch size and sliding window size w are large (e.g., hundreds of updates, and $w > 100$). We therefore optimize this process by separating the definition of Pearson correlation into a term representing the correlation before the update, and a term signifying the change in correlation. This allows us to update both pairwise correlations and **mc**-bounds in constant time and space, even though we showed that multiple correlation coefficients cannot be updated incrementally themselves. Details of this process are provided in Section 4.2.2.

Incremental updating of correlations and the DCC Index constitute the main weapons in CDStream's arsenal to efficiently maintain threshold queries without any additional constraints. Section 4.3 describes how these components are combined to create the base solution. This base solution is then extended to support those additional constraints by slightly modifying the storage of DCCs in the index. Central to this method is the observation that the DCC index effectively supports a triggering functionality - it facilitates quick lookup and verification of the conditions that need to hold after an update such that bounds of DCCs remain unchanged. If these conditions do not hold for a group of cluster combinations, it triggers a reconsideration of the state of those combinations. In the absence of additional query constraints, these conditions concern only the validity of the extrema pairs in the decisive combinations. Triggers for when additional constraints are (no longer) violated can be created in a similar fashion, by simply storing the decisive combinations in additional locations in the index. Section 4.4 formally explains this process for handling both of the additional constraints.

Finally, CDStream is modified to support top-k queries by applying minor changes to the steps in which the result set is initialized and violated DCCs are queried from the index. The modification involves querying the top-k variant of CD for a slightly larger number of results than k. This buffer on the result set is added to aid quick replacement of top-k vector combinations that move out of the top-k set due to a decrease in correlation. After initialization of the buffered set, CDStream finds the minimum correlation in these results, and uses it as a threshold τ in the arrival phase of the algorithm. In this arrival phase, CDStream processes a batch of updates in two steps; (1) recomputing the correlations of elements in the buffered result set to obtain the new value of τ, (2) then querying the DCC Index for changing negative DCCs which are potentially added to the buffered set based on the updated threshold. As long as the size of the result set is at least k, the true top-k results will be a subset of the buffered result set maintained by the algorithm. Before returning the results to the user, the elements of the buffered set are sorted on correlation, and the top-k are returned to the user. We provide an in-depth discussion of this algorithm variant in Section 4.5.

4.2 Algorithm components

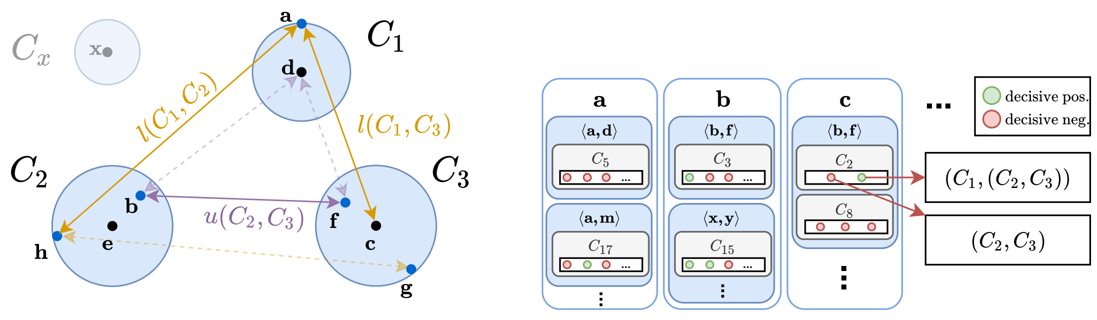
We now provide detailed descriptions of the main components of CDStream; the DCC Index and incremental updating of pairwise correlations. In these sections, we assume that the user provides a basic threshold query without any additional constraints.

4.2.1 The DCC Index

The main challenge in identifying DCCs that are in danger of changing state lies in reasoning about changes in bounds without actually computing them. Key in doing so is understanding what impacts the bound of a cluster combination. Observe from Theorem 3.3.1 that, even for combinations of three or more clusters, the bounds are determined by the minimum and maximum pairwise correlations between all involved clusters. In the inequalities these are denoted respectively as $l(C_i, C_j)$ and $u(C_i, C_j)$ for any pair of clusters C_i and C_j. From this we can conclude that any update that does not change $l(C_i, C_j)$ or $u(C_i, C_j)$ will not invalidate the previous bounds, or the previous solution. We refer to the pairs of vectors from C_i and C_j that are responsible for $l(C_i, C_j)$ and $u(C_i, C_j)$ as the **minimum extrema pair** and the **maximum extrema pair**, respectively.

Extrema pairs

Figure 4.1(a) illustrates the extrema pairs for a positive DCC with clusters from the running example. Note again that Euclidean distance is inversely related with correlation, meaning that



(a) (b)

Figure 4.1: (a) Illustration of extrema pairs for a positive DCC with $l_{\max} = 1, r_{\max} = 2$ (b) Visualization of the DCC Index

vectors (i.e., points) close to one other have relatively high correlations [32]. In this example, the minimum and maximum extrema pairs for (C_2, C_3) are $\langle \mathbf{h}, \mathbf{g} \rangle$ and $\langle \mathbf{b}, \mathbf{f} \rangle$, respectively. From the definitions in Theorem 3.3.1 we can conclude that the dominant bound of this DCC (i.e., the LB) is made up of the minimum extrema pairs of (C_1, C_2) and (C_1, C_3), and the maximum extrema pair of (C_2, C_3). We call these *active extrema pairs*. Naturally, the DCC's dominant bound will only decrease if any of these extrema pairs change undesirably, either by becoming more extreme (i.e., $\langle \mathbf{b}, \mathbf{f} \rangle$ increases) or by being surpassed by another pair (i.e., $\rho(\mathbf{e}, \mathbf{f})_{t+1} > \rho(\mathbf{b}, \mathbf{f})_{t+1}$). To analyze this geometrically, note that updates potentially cause a change in the digest value of the last basic window. From perspective of z-normalized vectors this means that all values change causing the vector to move in z-normalized space. Consider receiving an update for $\mathbf{e}$, then the bound of the DCC changes if $\mathbf{e}$ moves closer to a point in C_3 than $\mathbf{b}$ is to $\mathbf{f}$, or further away from C_1 than $\mathbf{b}$ is to $\mathbf{d}$.

This insight implies that if DCCs are stored and indexed based on their extrema pairs, one can retrieve potential 'state changers' by updating pairwise correlations and locating only the DCCs where active extrema pairs have changed. We do this by building an index that maps each vector $\mathbf{v}$ to a list of all decisive combinations (both negative and positive) that involve any cluster with $\mathbf{v}$. Figure 4.1(b) shows a visual representation this index, which we naturally named the DCC Index (note that the representation is in concordance with the running example). In practice, effectively two DCC Indices are built and maintained; one where combinations are indexed based on *upper bounds* on correlations of $\mathbf{v}$ (related to maximum extrema pairs), and one where combinations are indexed based on *lower bounds* on correlations of $\mathbf{v}$ (related to minimum extrema pairs). For conciseness, we will continue to refer to indexing decisive combinations as if they are contained in one overarching structure.

Grouping DCCs

Internally, the combinations for $\mathbf{v}$ are grouped in multiple levels. This decision comes from the observation that possibly many different cluster combinations have similar extrema pairs. There are two scenarios in which this can happen;

Scenario 1: identical cluster pairs. Consider a close alternative to the situation in the running example where the root cluster also included a vector $\mathbf{x}$, which is somewhat dissimilar from the other vectors. Then, it is likely that $\mathbf{x}$ would directly be split from the rest into a child cluster of the root C_x, only containing $\mathbf{x}$. The resulting situation is opaquely illustrated in Fig. 4.1(a). In this scenario it is perfectly possible that a decisive combination exists with $(C_1, (C_2, C_3))$ but also with $(C_x, (C_2, C_3))$. As the DCCs share the cluster pair (C_2, C_3), we may check the validity of its extrema pairs only once while making inferences on the bounds of both DCCs, saving effort.

Scenario 2: related cluster pairs. The scenario of overlapping extrema pairs among DCCs is not limited to the case of DCCs having identical cluster pairs. Active extrema pairs may also be shared among DCCs when they include clusters from the same branch in the cluster tree. For example, one decisive combination may include the cluster pair (C_2, C_3) while another includes (C_8, C_3), with $C_8 = \{\mathbf{b}\}, C_2 = \{\mathbf{b}, \mathbf{e}, \mathbf{h}\}$ (recall the running example in Fig. 3.1). As long as both decisive combinations have the same state (i.e., positive/negative) and the cluster pairs exist at the same side of the correlation pattern, the active extrema pairs will be the same. Thus, effort can be saved analogously to Scenario 1.

To exploit this observation, decisive combinations for $\mathbf{v}$ are first grouped by the extrema pairs that $\mathbf{v}$ can invalidate.

Example. In Fig. 4.1(b), the first 5 combinations for vector $\mathbf{c}$ are grouped under extrema pair $\langle \mathbf{b}, \mathbf{f} \rangle$. The first two of the five combinations both involve the pair of clusters C_2 and C_3. C_3 is the cluster that contains $\mathbf{c}$ and has $\mathbf{f}$ as an extremum, whereas C_2 has $\mathbf{b}$ as an extremum. All combinations with the same extrema pair are subsequently grouped by the cluster that does not contain the vector used for indexing (in this case, the vector $\mathbf{c}$). In our example, the first two combinations both involve the same extrema pair, and both have C_2 as the second cluster. The remaining three combinations under extrema pair $\langle \mathbf{b}, \mathbf{f} \rangle$ contain cluster C_8 instead as a second cluster, and therefore are placed in a different sub-group. We will refer to these clusters for the same extrema pair as the *extrema pair clusters*.

Querying DCCs

Recall that the key utility of the described index is to support a triggering functionality for when DCC bounds change in an undesired manner. It allows us to quickly locate and verify the extrema pairs related to each update. An update of any stream s_i is processed as follows. First, the index is used to retrieve the information related to s_i (line 2 of Algorithm 2). The algorithm iterates over the respective extrema pairs to verify that these did not change or move, despite the recent update of $\mathbf{v}_i$. Precisely, for each minimum extrema pair with correlation $\rho_{\min}$, it checks if the correlation of $\mathbf{v}_i$ with all points belonging to the second cluster is still at least equal to $\rho_{\min}$, whereas for each maximum extrema it verifies that all correlations of $\mathbf{v}_i$ remain less than $\rho_{\max}$ (line 8). If this is still the case, all decisive combinations are still correct and do not need to be checked one-by-one. If, on the other hand, an extrema pair is invalidated (e.g., if the updated correlation of $\mathbf{v}_i$ with a vector from the opposite cluster becomes less than $\rho_{\min}$), the respective decisive combinations are checked and their bounds are recomputed, re-indexing the combinations and updating the result set if necessary.

Checking whether $\rho_{\min}$ (resp. $\rho_{\max}$) are still valid requires computing the correlation of $\mathbf{v}$ with each of the vectors contained in all extrema pair clusters (lines 7-10). A critical observation is that there always exists one cluster in the extrema pair clusters that contains all others – otherwise the other clusters could not contain the same extrema vector. To avoid double work, the algorithm considers the extrema pair clusters in decreasing size. If the largest cluster passes the test, then all its decisive combinations and all the decisive combinations of all its sub-clusters are still valid and do not need to be checked (lines 12-13). In the running example, if $\langle \mathbf{b}, \mathbf{f} \rangle$ is still the maximum extrema pair between clusters C_3 and C_2, and its correlation did not change, then all combinations under $\langle \mathbf{b}, \mathbf{f} \rangle$ are still decisive. If the largest cluster does not pass the test, then the bounds for all its decisive combinations are verified (line 10). The combinations that are no longer decisive are updated accordingly, e.g., by splitting one of the involved clusters to sub-clusters, as described in Section 3.3. This process is referred to as *breaking cluster combinations*. Furthermore, since the decisive combinations of the largest cluster were updated, the second, third, etc. largest extrema pair clusters are tested recursively (lines 14-15). The process can stop as soon as one of these clusters passes the test.

This grouping of decisive combinations based on the extrema pairs and clusters is instrumental in the algorithm's performance. It allows us to multiply the pruning power of CD by considering only a fraction of all cluster combinations (which are already significantly less than the amount of vector combinations). The effective pruning power of CDStream is dependent on the *significance of updates*, i.e., how many extrema pairs they end up violating. The pruning power also comes with the overhead cost of maintaining and iterating over the index. More thorough analysis of the

```
Algorithm 2: QueryIndex($i, \mathcal{I}$)
  Input: A stream index $i$, the DCC Index $\mathcal{I}$
  Output: A set of DCCs $O$ that need to be checked
1  $O \leftarrow \{\}$
2  for $\langle \mathbf{a}, \mathbf{b} \rangle \in \mathcal{I}[i]$ do
3      $first \leftarrow true$
4      $V \leftarrow \{\}$
5      for $C \in \mathcal{I}[i][\langle \mathbf{a}, \mathbf{b} \rangle]$ do
6          if $first$ then
7              for $\mathbf{v}_j \in C$ do
8                  if $\rho(\mathbf{v}_i, \mathbf{v}_j)_t > \rho(\mathbf{a}, \mathbf{b})_t$ then
9                      $V \leftarrow V \cup \mathbf{v_j}$
10                     $O \leftarrow O \cup \mathcal{I}[i][\langle \mathbf{a}, \mathbf{b} \rangle][C]$
11             $first \leftarrow false$
12             if $V = \emptyset$ then
13                 break
14         else if $C \cap V \neq \emptyset$ then
15             $O \leftarrow O \cup \mathcal{I}[i][\langle \mathbf{a}, \mathbf{b} \rangle][C]$
16 return $O$
```

```
Algorithm 3: UpdateStream($S_i, d, v_{\text{old}}$)
  Input: The windowed digests $S_i$ of a stream $i$, an incoming data point $d$ for stream $i$, the value that needs to be overwritten $v_{\text{old}}$.
  // Update the digest value
1  $v_{\text{new}} \leftarrow$ Agg($S_i[L], d$)
2  $\mathbb{E}[S_i] \leftarrow \mathbb{E}[S_i] + \frac{1}{w}(v_{\text{new}} - v_{\text{old}})$
3  $\mathbb{E}[S_i^2] \leftarrow \mathbb{E}[S_i^2] + \frac{1}{w}(v_{\text{new}}^2 - v_{\text{old}}^2)$
4  for $j \leftarrow 1$ to $n \setminus \{i\}$ do
       // Prevent concurrency errors
5      if $S_j.last\_update < t$ then
6          $\mathbb{E}[S_i S_j] \leftarrow \mathbb{E}[S_i S_j] + \frac{v_{\text{old},j}}{w}(v_{\text{new}} - v_{\text{old}})$
7          $\rho(S_i, S_j) \leftarrow \frac{\mathbb{E}[S_i S_j] - \mathbb{E}[S_i]\mathbb{E}[S_j]}{\sqrt{\mathbb{E}[S_i^2]\mathbb{E}[S_i]^2}\sqrt{\mathbb{E}[S_j^2]\mathbb{E}[S_j]^2}}$
8  $S_i[L] \leftarrow v_{\text{new}}$
```

pruning power and overhead costs is provided in subsection 4.3 and the evaluation in Chapter 6.

4.2.2 Incremental updating of pairwise correlations

Recall from earlier that **mc**-bounds cannot be updated in an incremental fashion as new data arrives. However, due to the reduction of streams to sequences of basic windows, sliding windows are guaranteed to remain aligned which allows us to incrementally update pairwise correlations instead. This is favorable considering the relevance of pairwise correlations in checking the validity of extrema pairs. Namely, checking the validity of an extrema pair involves comparing it to updated pairwise correlations between cluster pairs. Seen that the cluster combinations encompass all possible vector combinations, it is likely that a substantial portion of the correlation matrix needs to be updated before querying the DCC Index. Hence, updating pairwise correlations in a more efficient manner will likely improve the algorithm's performance significantly.

The Pearson correlation function of two random variables X, Y can be decomposed as follows;

$$\rho(X,Y) = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[(X-\mu_X)(Y-\mu_Y)]}{\sqrt{\mathbb{E}[(X-\mathbb{E}[X])^2]}\sqrt{\mathbb{E}[(Y-\mathbb{E}[Y])^2]}} = \frac{\mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]}{\sqrt{\mathbb{E}[X^2]\mathbb{E}[X]^2}\sqrt{\mathbb{E}[Y^2]\mathbb{E}[Y]^2}} \tag{4.1}$$

In our context, X and Y are not random variables but digests the basic windows inside the sliding window (i.e., vectors). Therefore, the $\mathbb{E}$-terms refer to the average value of the expression within those terms. A term $\mathbb{E}[XY]_t$ in the current context indicatively boils down to $\frac{1}{w}\sum_{i=L+w-1}^{L}(S_x[i]S_y[i])$, with $S_X[i]$ referring to the i'th basic window digest of stream X. Recall that L is used to indicate the index of the running basic window (see Section 2.2.1). For brevity we will continue to denote the averages on sliding windows of stream with the expected value notation. Consider a stream x receiving an update between timepoint t and $t+1$. Then, each of these terms can be incrementally updated as follows[1];

$$\begin{aligned}
\mathbb{E}\left[s_x\right]_{t+1} &= \mathbb{E}\left[s_x\right]_t + \frac{1}{w}(S_x[L]_{t+1} - S_x[L]_t) \\
\mathbb{E}\left[s_x\right]^2_{t+1} &= \mathbb{E}\left[s_x\right]^2_t + \frac{1}{w}(S_x[L]^2_{t+1} - S_x[L]^2_t) \\
\mathbb{E}\left[s_x s_y\right]_{t+1} &= \mathbb{E}\left[s_x s_y\right]_t + \frac{1}{w}((S_x[L]_{t+1} * S_y[L]_{t+1}) - (S_x[L]_t * S_y[L]_t))
\end{aligned}$$

At completion of the running basic window, the terms can be updated analogously with the exception of replacing $S_x[L]_t$ with $S_x[L-w]_t$. From these equations we conclude that for each stream we only need to maintain the running sum, running sum of squares and the running products with other streams. Upon an update or move of the sliding window one can incrementally update the correlation using these statistics in combination with the change in digest value. This is significantly more efficient than doing a full pass over the data at every update. Alg. 3 provides a pseudo-code representation of the process of updating running statistics and pairwise correlations.

A critical observation is that conversely with CD, we do not z-normalize the windowed digests (i.e., vectors). Instead, we effectively z-normalize the vectors through the discussed recomputing of Pearson correlations by considering the general equation for Pearson[2]. The Pearson definition used in the proofs of Theorem 3.3.1 uses $\rho(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{m * \sigma_{\mathbf{x}} \sigma_{\mathbf{y}}}$ which is a derivation of the general definition of Pearson with the assumption that vectors are already z-normalized. The definition of **mc**-bounds from Theorem 3.3.1 can still be used as both Pearson definitions come to the same answer. This is because z-normalization merely re-scales vectors which does not impact the value of linear relationships like Pearson correlation. We ought to be careful though with using the original clustering algorithm of CD as it uses Euclidean distance as a distance measure, which is impacted by not z-normalizing vectors. This problem is easily solved by changing the distance measure to the inverse Pearson correlation value (i.e., large correlations have a small distance value). This will result in the same cluster tree, and pairwise correlations are computed prior to running CD anyway.

4.3 Algorithm overview

Now that we have the methods in place for initiating a first solution and efficiently handling arrivals, we can combine them into an algorithm that finds and maintains multivariate correlations. This section provides an overview of the full algorithm including an (informal) analysis of its complexity and pruning power.

4.3.1 Initialization phase

Naturally, the initialization phase starts with the first phase of CD. Here, statistics and pairwise correlations are computed from scratch with the first sliding window, which are subsequently used to construct the cluster tree (lines 1-2 of Algorithm 4). Then, a 'stream-aware' modification of CD is run which stores DCCs in the DCC Index when they are identified (line 3). Storing and indexing a decisive combination involves iterating over all possible cluster pairs in the combination, extracting the extrema pair of each cluster pair, and storing the DCC in the index keyed by the streams in each of the clusters, the respective extrema pair, and the cluster opposite to the respective stream. A full pseudo-code representation of this process can be found in Appendix B. The stream-aware variant of CD calls this method after lines 3 and 5 of Algorithm 1. Note that this modification will lead to a significant increase in runtime for CD as negative DCCs now also require to be stored. In the original CD algorithm these were simply ignored when found whereas positive DCCs required deconstruction and addition to result set.

[1]Complete derivations can be found in Appendix A

[2]$\rho(x, y) = \frac{\text{Cov}(x,y)}{\sigma(x)\sigma(y)}$

4.3.2 Arrival phase

Upon initialization of the result set and DCC Index, the algorithm is ready to start handling batches of arrivals. This process consist of three steps; (1) updating of streams (Alg. 3), (2) querying the index for potential violations (Alg. 2) and (3) recomputation of bounds leading to potential changes in the result set and reindexes (in order). Note that Algorithm 2 only shows the process of querying the DCC Index used for checking the validity of maximum extrema pairs. The process is analogous for querying the DCC Index used for minimum extrema pairs. Also, line 8 of Alg. 2 does not cover the case when $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ is the extrema pair itself. In this case, the algorithm compares $\rho(\mathbf{v}_i, \mathbf{v}_j)_t$ with $\rho(\mathbf{v}_i, \mathbf{v}_j)_{t-1}$. This was left out for simplicity and conciseness.

4.3.3 Parallelization

In practice, each processing step in the arrival phase is multithreaded, meaning that every task is parallelized but threads are joined at the end of every step. A thread-join after updating streams is necessary because querying the index for an updated vector $\mathbf{v}$ involves consideration of pairwise correlations excluding $\mathbf{v}$ (i.e., correlations of extrema pairs). In absence of a join between the two steps, it is possible that a thread considers the correlation of an extrema pair that is not yet updates as part of the task of another thread. This may lead to incorrect conclusions on the validity of extrema pairs. Furthermore, step 2 and 3 are separated for performance-related reasons related to thread partitioning. Namely, load imbalance may occur if the two steps are merged due to some streams receiving more significant updates than others. For example, consider we parallelize steps 2 and 3 with one CPU handling the query process for one stream in the batch. If a certain stream has more significant updates than others, more DCCs are violated meaning that the CPU has to recompute significantly more bounds than the others, which is bad for performance. Therefore, labeled DCCs are collected first, joining threads, after which work is again divided for recomputing and reindexing the combinations.

4.3.4 Cost analysis

As said before, the additional pruning power of CDStream comes at the cost of storage of DCCs and the cost of querying the index. Formal complexity analysis of the computation- and storage cost of the algorithm is difficult as both pruning power and index size are heavily dependent on the amount of DCCs in the system and the significance of updates. These factors depend on the data distribution; how well it can be clustered, and the amount of correlation drift over time. Accounting for these factors would result in a complex stochastic analysis of the algorithm's performance. This was left out of the scope of the study. Nonetheless, in this subsection we will reason about the main factors that influence the algorithm's computational effort. This will aid in interpreting the experimental results discussed in Chapter 6. This will be done for three aspects of the algorithm; storage of decisive combinations in the index, querying the index, and the pruning power that may be obtained from it.

Storage of decisive combinations

Recall that for each stream $s \in \mathcal{S}$, the index contains a separate partition with all the DCCs including s, indexed by the extrema pairs s can violate. Additional grouping of DCCs on opposing clusters of s is irrelevant for this analysis. In practice, the DCC Index is implemented as a nested structure of hash maps, with DCCs keyed by three factors; the stream's index i, the extrema pair in the DCC that s_i can invalidate, and the cluster in the DCC opposite to s_i, in that order. From the perspective of an individual DCC, this means that it has to be stored $(\mathcal{P}_l + \mathcal{P}_r - 1)n_{cc}$ times, where n_{cc} is the total amount of vectors in the cluster combination. This is because there exists an extrema pair for every cluster pair in the DCC. Every cluster in the DCC participates in $(\mathcal{P}_l + \mathcal{P}_r - 1)$ cluster pairs, so every stream in C can logically invalidate $(\mathcal{P}_l + \mathcal{P}_r - 1)$ cluster pairs.

We conclude from this that the storage of DCCs grows linearly with the total amount of DCCs, the average size of DCCs, and the values of $l_{\max}$ and $r_{\max}$. Simply put, CDStream benefits from small amounts of 'large' DCCs. Seen that the average size of DCCs is directly related to the total amount of DCCs, we can also conclude that the storage complexity of CDStream is directly dependent on CD's ability to cluster the data and the tightness of the **mc**-bounds. Note that these factors are also heavily dependent on the data itself and the value of τ.

Querying the index

Querying the index involves iterating over the extrema pairs that streams in the arrival batch can violate, and comparing the correlations between the stream at question and one 'opposing cluster' to that of the extrema pair. Extrema pairs are pairs of streams. Thus, for each unique stream in the arrival batch we may iterate over $\mathcal{O}(n^2)$ extrema pairs in the worst case. However, in practice we see that this complexity is actually closer to linear complexity. This has to do with the similarity between DCCs, as discussed in Section 4.2.1. When the clusters in DCCs are relatively large, many DCCs will have overlapping extrema pairs, resulting in less second-level keys in the index.

Again, the computational complexity of this task is mainly dependent on the average size of the clusters in decisive cluster combinations, which eventually translates to the average size of DCCs overall. Large clusters result in more similar DCCs, which limits the amount of work that has to been done on querying the index.

Pruning power

The power of pruning vector combinations for CDStream comes from reconsidering only a selection of decisive combinations inherited from CD, which themselves are already pruning combinations. This means that CDStream's pruning power is guaranteed to be at least that of CD. However, the overhead from enabling this extra pruning might not outweigh its costs (further discussed in Chapter 5). The fraction of DCCs to consider after a batch is dependent on the amount of extrema pairs those arrivals violate. We referred to the latter as the *significance* of arrivals.

Extrema pairs are violated by related pairwise correlations exceeding its own correlation at $t-1$. If pairwise correlations change more drastically, extrema pairs are more likely to be violated, making arrivals more significant. The change of a correlation is dependent on its value relative to the other values in the window, and the window sizes w and b_{bw}. This creates a trade-off between algorithmic performance and robustness of correlations. Namely, large values for w and b_{bw} result in less significant updates, and therefore higher pruning power, while a small values increase the result's sensitive to short-term trends. The latter is especially important for financial applications as it enables the algorithm to identify short-term opportunities earlier. Concluding, the values of w and b_{bw} need to be chosen carefully, depending on what is important in the problem context.

[3]InsertionSort is used as it is most efficient for structures that are already in approximate sorted order.

Algorithm 4: CDSTREAM($\mathcal{S}, w, b, \tau$)

```
Input: Set of streams $\mathcal{S}$, sliding window size $w$, basic window size $b_{bw}$, correlation threshold $\tau$
$\mathcal{V} \leftarrow$ InitWindows($\mathcal{S}, w, b$)
InitCorrelations($\mathcal{V}$)
$\mathcal{R}, \mathcal{I} \leftarrow$ CD($\mathcal{V}, \tau$)
$t \leftarrow 1$
while $true$ do
    $A \leftarrow \{\}$
    if $t \;\%\; b_{bw} = 0$ then
        $\mathcal{V} \leftarrow$ ProgressWindows($\mathcal{V}$)
        $A \leftarrow \mathcal{V}$
        for $S_i \in \mathcal{V}$ do
            UpdateStream($S_i, S_i[L-1], S_i[L-w]$)
    // Arrival of a batch $B$
    if $|B| > 0$ then
        for $(i, t, d) \in B$ do
            UpdateStream($S_i, d, S_i[L]$)
            $A \leftarrow A \cup \{S_i\}$
    for $S_i \in A$ do
        $O \leftarrow$ QueryIndex($i, \mathcal{I}$)
        for $(\mathcal{P}_l, \mathcal{P}_r) \in O$ do
            CDThreshold($\mathcal{P}_l, \mathcal{P}_r, \tau$)
    $t \leftarrow t + 1$
```

Algorithm 5: CDSTREAMTOPK($\mathcal{S}, w, b, k$)

```
Input: Set of streams $\mathcal{S}$, sliding window size $w$, basic window size $b_{bw}$, top-k value $k$
$\mathcal{V} \leftarrow$ InitWindows($\mathcal{S}, w, b$)
InitCorrelations($\mathcal{V}$)
$\mathcal{R}^{k+b}, \mathcal{I} \leftarrow$ CDTopK($\mathcal{V}, k$)
$\tau_b \leftarrow \mathcal{R}^{k+b}[last]$.mc
$\mathcal{R}^{k} \leftarrow \mathcal{R}^{k+b}[1:k]$
$t \leftarrow 1$
while $true$ do
    Slide windows if necessary (lines 6-11 Alg. 4)
    if $|B| > 0$ then
        for $(i, t, d) \in B$ do
            UpdateStream($S_i, d, S_i[L]$)
            $A \leftarrow A \cup \{S_i\}$
        Update mc-values of $\mathcal{R}^{k+b}$
        Initialize again if $> k(b-1)$ passed $\tau_b$
        $\tau_b \leftarrow$ SORT[3]($\mathcal{R}^{k+b}$)[$last$].mc
        for $S_i \in A$ do
            $O \leftarrow$ QueryIndex($i, \mathcal{I}$)
            for $(\mathcal{P}_l, \mathcal{P}_r) \in O$ do
                CDThreshold($\mathcal{P}_l, \mathcal{P}_r, \tau_b$)
        $\mathcal{R}^{k} \leftarrow$ SORT($\mathcal{R}^{k+b}$)[1 : $k$]
        Index($\mathcal{R}^{k+b}[kb : last]$)
        $\mathcal{R}^{k+b} \leftarrow \mathcal{R}^{k+b}[1:kb]$
    $t \leftarrow t + 1$
```

4.4 Handling additional constraints

Additional constraints such as irreducibility and minimum jump can be handled elegantly by slightly modifying the use of the DCC Index. Notice again that the index support a triggering functionality - a quick lookup and verification of the conditions that need to hold after an update on one or more streams. Triggers on the compliance to additional constraints can be created by storing decisive combinations in additional locations in the index. The following section describes this process in detail.

4.4.1 Irreducibility

Let X, Y, X', Y' denote sets of clusters. Consider combinations (X, Y), and $(X' \subseteq X, Y' \subseteq Y)$, with $|X \cup Y| > |X' \cup Y'|$. In words, we consider two cluster combinations; one enclosing correlation patterns of high cardinality, and another enclosing patterns of lower cardinality, using a subset of the clusters of the first combination.

Example. Consider two cluster combinations $(X, Y) = (C_1, (C_2, C_3, C_4))$ and $(X', Y') = (C_1, (C_2, C_3))$. The irreducibility constraint involves rejecting (X, Y) from the result set if (X', Y') is already in it (or needs to be). Preserving compliance to this constraint as data arrives requires detection of two cases: (a) (X, Y) needs to be removed from the result set because **mc**(X', Y') surpassed τ due to an update, and, (b) (X, Y) needs to be added to the result set, because (X', Y') was just removed from the result set. Notice that both cases can be triggered by an update of a vector from X or Y (hence, also from X' and Y').

Without the irreducibility constraint, the index contains the following extrema pairs: (a) for the negative DCCs, all pairs required for upper-bounding the **mc**-value, (b) for the positive DCCs, all pairs required for lower-bounding the **mc**-value. Case (a) of irreducibility violation can be detected by additionally monitoring the upper bounds of positive DCCs (i.e., those that encapsulate (X, Y)). To illustrate this, consider again the example where $X = \{C_1\}, X' = \{C_1\}, Y = \{C_2, C_3, C_4\}, Y' = \{C_2, C_3\}$, and $\mathbf{mc}(X, Y)_{LB} > \tau$, $\mathbf{mc}(X', Y')_{UB} < \tau$. If a materialization of (X', Y') receives an update such that its **mc**-values rises above τ, the UB of (X', Y') must also rise above τ, as it spans all of the possible **mc**-values of its materializations. Note that extrema pairs contributing to the UB of (X', Y') are a subset of those contributing to the UB of (X, Y), as shown in the following;

$$\mathbf{mc}(X, Y) \leq \frac{u(C_1, C_2) + u(C_1, C_3) + u(C_1, C_4)}{\sqrt{3 + 2(l(C_2, C_3) + l(C_2, C_4) + l(C_3, C_4))}} \tag{4.2}$$

Clearly, monitoring the bounds of cluster combinations of high cardinality simultaneously monitors the bounds of 'simpler' cluster combinations. This method is also applied for detecting case (b), by monitoring the lower bounds of negative DCCs with any $\mathbf{mc}(X', Y') > \tau$. Monitoring both bounds for a decisive combination involves indexing it under *all* extrema pairs, and checking them accordingly.

One might argue that the above method does not cover all (edge-)cases. Particularly, if the bounds of the simple combination are of a different bound case in Theorem 3.3.1 than the complex combination's bounds, the highlighted terms in Equation 4.2 become incorrect. While valid, this edge-case does not jeopardize the algorithm's correctness. This is because the above method effectively results in monitoring all extrema pairs in a cluster combination, instead of only those contributing to one of the **mc**-bounds. Regardless of bound cases, the extrema pairs of simple combinations will still be a subset of the extrema pairs in a complex combination. Therefore, if for each cluster pair both the minimum and maximum extrema pairs are monitored, both bounds will be effectively be monitored for both cluster combinations, and constraint violations will be detected.

4.4.2 Minimum jump constraint

Monitoring for the minimum jump constraint is analogous to that of the irreducibility constraint. The following cases need to be considered: (a) (X, Y) needs to be removed from the result set because $\mathbf{mc}(X', Y') > \tau$ and $\mathbf{mc}(X', Y') + \delta > \mathbf{mc}(X, Y)$, and (b) (X, Y) needs to be added in the result set because $\mathbf{mc}(X, Y) > \tau$ and $\mathbf{mc}(X', Y') + \delta < \mathbf{mc}(X, Y)$. Both cases are identified using the discussed method for monitoring the irreducibility constraint.

4.5 Top-k queries

CDStream can be modified to support top-k queries by applying minor changes to the process of initializing and maintaining the result set. Similar to Section 4.3, the following section goes through the main steps of the algorithm and shows how they are modified to support this query type. The section also includes a discussion on the changes it brings to the storage complexity of the DCC Index, and its potential pruning power.

4.5.1 Initialization phase

Recall that CDStream is initialized with the result set $\mathcal{R}$ of CD. For a top-k query, CDStream queries CD for a slightly larger number of results $k' = b * k$, where b is a small integer, greater than 1 (line 3 of Algorithm 5). As a result, CD returns a set $\mathcal{R}^{k+b}$ of size $b * k$ which contains the true top-k set $\mathcal{R}^k$ and a buffer set of vector combinations $\mathcal{R}^b$ (See Fig. 4.2 and Table. 4.1). This buffer is added to aid quick replacement of top-k vector combinations that move out of the result set $\mathcal{R}^k$. In its process, CD breaks positive DCCs down to multiple DCCs of singleton clusters to check compliance to additional constraints and to construct the correct top-k set. We modify

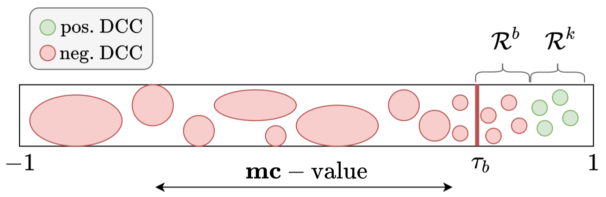


Figure 4.2: Visualization of top-k DCC sets

Symbol	Description
$\mathcal{R}^k$	top-k set
$\mathcal{R}^b$	buffer of top-k set
$\mathcal{R}^{k+b}$	$\mathcal{R}^k \cup \mathcal{R}^b$
τ_b	lowest **mc**-value in $\mathcal{R}^b$
τ_k	lowest **mc**-value in $\mathcal{R}^k$

Table 4.1: Nomenclature of CDStream top-k queries

the stream-aware version of CD (recall from Section 4.3) so that it does not index positive DCCs (i.e., DCCs in $\mathcal{R}^{k+b}$), only negative DCCs. This decision comes from the fact that $\mathcal{R}^k$ will need to be reconstructed from $\mathcal{R}^{k+b}$ after every update batch. CD progressively builds $\mathcal{R}^{k+b}$. This means that DCCs with bounds under the *intermediate* correlation threshold will be stored in the index. Furthermore, CD also continuously kicks-out cluster combinations from $\mathcal{R}^{k+b}$ when stronger combinations are found. The rejected cluster combinations will be stored in the index ad-hoc.

When CD is finished, CDStream finds the minimum correlation in $\mathcal{R}^{k+b}$, and uses it as a threshold τ_b in the streaming algorithm (lines 4-5). As long as the size of $\mathcal{R}^{k+b}$ is at least k, the true top-k results will always have a correlation higher than τ_b and will therefore be a subset of $\mathcal{R}^{k+b}$.

4.5.2 Arrival phase

When a batch of updates arrives, CDStream first recomputes the correlations of the elements in $\mathcal{R}^{k+b}$ so that τ_b is updated before the index is queried (lines 15 and 17). The latter is done next in identical manner as with threshold queries, i.e., by adding DCCs to the result set if they exceed the (intermediate) correlation threshold (lines 18-21). Subsequently, $\mathcal{R}^k$ is constructed from $\mathcal{R}^{k+b}$ and returned to the user (line 22). As a post-processing step, the buffer set $\mathcal{R}^b$ is reduced if it has excessively grown (e.g., $|\mathcal{R}^{b+k}| = 2bk$) due to outsiders entering the set (lines 23-24).

Observe that there exists a critical edge case in the above process where a DCC is wrongfully left out of $\mathbb{R}^k$, jeopardizing the algorithm's correctness. Consider the extreme case that in one epoch, all elements of $\mathcal{R}^{k+b}$ have a decrease in correlation resulting in that they all move under τ_b. Then, τ_b is updated, bounds for moving DCCs are recomputed, but no DCC moves into $\mathcal{R}^{k+b}$. However, if a DCC exists with a dominant bound less than τ_b at t, and its bounds stay constant in the transition to $t+1$, the DCC will not be reconsidered while it should be included in $\mathcal{R}^k$ due to the huge downshift in τ_b. This edge case is covered by rerunning the initialization phase (including CD) if more than $k(b-1)$ elements from $\mathcal{R}^{k+b}$ move under τ_b in one epoch (line 16).

4.5.3 Combining with additional constraints

Additional indexing of positive DCCs to monitor compliance to irreducibility and minimum jump constraints can be omitted as this is checked anyway when recomputing the bounds of DCCs in $\mathcal{R}^{k+b}$. Negative DCCs with any $\mathbf{mc}(X', Y') > \tau$ (irreducibility) or $\mathbf{mc}(X', Y') + \delta > \mathbf{mc}(X, Y)$ (min. jump) are handled by adding them to $\mathcal{R}^{k+b}$, regardless of their disconformity to the constraints. This means that their bounds are reconsidered at every epoch. They are, however, not considered when updating τ_b and $\mathcal{R}^k$.

4.5.4 Impact on storage and performance

Scaling factor b controls the trade-off between the robustness of the buffer set, and the efficiency of the algorithm. A $b = 1$ may lead to more situations where the discussed edge case applies and the algorithm needs to be reinitialized, which is very expensive. Conversely, a large b will lead to a higher number of intermediary results, and to more effort for computing the exact correlation of these results, which is necessary for retaining the top-k results. Our experiments with a variety of datasets have shown that $b = 2$ is already sufficient to provide good performance without compromising the robustness of the algorithm.

The ability of CD to quickly reach the 'true' correlation threshold does not only impacts the performance of CD, but also that of CDStream. Namely, the more combinations that have to be kicked out of the intermediate result set, the more DCCs of clusters of size one (i.e, *singleton* cluster) will be stored in the DCC Index. Moreover, throwing DCCs of singleton clusters from the buffered set back into the pond of indexed negative combinations also results in more small DCCs in the index. However, this is less of a problem as those DCCs have correlations close to the threshold which makes them likely to enter $\mathcal{R}^{k+b}$ again in the future, for which they need to be broken down anyway. Still, recall from Section 4.3.4 that the average size and total amount of DCCs in the index negatively impacts the algorithm's performance. Therefore, we should try to limit this impact by carefully choosing b, and optimizing the parameters of CD.

Another way to battle this phenomenon is to attempt to keep DCCs in their original 'unbroken' state in $\mathcal{R}^{k+b}$ while still making inferences on the location of their materializations in the buffer and top-k set. Unfortunately, prototypes of solutions all showed that the cost of this concept did not outweigh the benefits.

Chapter 5

CDHybrid

The following chapter addresses the sustainability of CDStream's performance over time, and argues that there are certain scenarios that may jeopardize the performance short-term and worsen the efficiency of the index. To battle this, we propose an extension to CDStream which enables it to adapt to anomalous behavior of in streams. Similar to the previous chapter, we start by discussing the algorithm's motivation and general idea, followed by in-depth descriptions of each of its components.

5.1 Motivation

Following from the cost-benefit analysis in Section 4.3.4, we expect CDStream to outperform CD in situations where the amount and *significance of updates* is small. This involves the majority of the applications we focus on, which generally have no drastic changes in the data, and data arriving at reasonable velocities. However, at sudden events, the overhead of CDStream can grow excessively for a few epochs, potentially leading to data congestion issues. Think of situations like prominent business people tweeting about certain crypto-currencies [1], which can cause prices to behave in odd ways, disturbing commonly stable correlation patterns. During these periods, it is likely that many DCCs need to be reconsidered and broken (i.e., splitting clusters), which is expensive and not good for the state of the DCC Index. Therefore, we might want to run CD instead during this time, as it is invariant to arrival significance, which may cause it to temporarily outperform CDStream. Note again that this only happens when the overhead of maintaining the index is larger than the pruning power one gains from it.

It is likely that after this period, correlation patterns grossly return to the way they were. If we *freeze* the state of the DCC Index during these periods, we might be able to maintain some DCCs in their original state, while they would have been broken down if CDStream was kept running. This is favorable as we know from the cost analysis that relatively larger DCCs should result in better performance. In case the correlation patterns do not go back to their previous state, we will at least delay the process of reconsidering impacted DCCs to a period of less arrivals or less significant arrivals. This way, we have some extra time on our hands to do some repair work, having to worry less about data congestion issues. The nature of this repair work will become apparent in later sections.

In this chapter we present CDHybrid, an algorithm that orchestrates CD and CDStream, transparently managing the switch between the two algorithms based on the properties of the input stream. This process involves (1) making an intelligent decision on which algorithm to run, and (2) efficiently switching to the another algorithm if strategies have changed, both of which are not trivial. The following subsection describes our solutions to the two issues in-depth.

5.2 Algorithm selection

To decide between CD and CDStream, CDHybrid needs to estimate the cost of each approach for handling each batch. As discussed before, the cost of CDStream is mainly dependent on the state of the DCC Index (i.e., amount and size of stored combinations), and the significance of the batch. Therefore, a good predictor for the cost would be recent processing times for similar batches. A simple measure of batch similarity is the number of updates in the batch (assuming time-based batching). The cost of CD is expected to be more stable, making the overall average processing time a good cost predictor for that algorithm. CDHybrid accounts for both factors by employing online linear regression to model the relationship between the number of updates, and the execution time of each algorithm. Consider the model function

$$c = \alpha + \beta B$$

which describes a line with slope β and intercept α, mapping the amount of updates B to an execution time c. Training of the model involves finding estimated values $\hat{\alpha}$ and $\hat{\beta}$ for the coefficients which would provide the 'best' fit in some sense for the data points. In this work, the best fit will be understood as in the least-squares approach: a line that minimizes the sum of squared residuals.

CDHybrid trains two of these models (i.e., one CD and one for CDStream) in order to estimate the cost of each algorithm at the start of an epoch. Initial data is gathered for the models by running a brief training or *warm-up period* at the initialization of CDHybrid, testing out the performance of the two algorithms for a couple of epochs. During this period, CDHybrid collects statistics on the observed arrival count and execution time.

We will show how the coefficients of a simple linear regression model can be estimated and maintained in constant time and space. This will enable us to continue training the models after the training phase with negligible overhead costs.

Given a sample of values for c and B of size T, simple linear regression offers elegant expressions for the estimates of α and β which minimize the sum of squared residuals [19];

$$\begin{aligned}\hat{\alpha} &= \bar{y} - (\hat{\beta}\bar{B}),\\ \hat{\beta} &= \frac{\sum_{i=1}^{T}(B_i - \bar{B})(y_i - \bar{y})}{\sum_{i=1}^{T}(B_i - \bar{B})^2}\\ &= \frac{s_{B,c}}{s_B^2}\end{aligned}$$

where:

$\bar{B}, \bar{c}$ = the sample average of B and y , respectively
$s_{B,c}$ = sample covariance of B and c
s_B^2 = sample variance

Note that the equations include the same statistics used in the computation of pairwise correlations. We previously showed how these statistics can be incrementally updated. However, this situation is slightly different as these statistics span an ever-increasing period while the statistics used for pairwise correlations consider the values inside a sliding window. Still, the sample statistics that make up the estimators can be incrementally updated in a similar way;

$$\begin{aligned}\bar{B}_{t+1} &= \bar{B}_t + \frac{B_{t+1} - \bar{B}_t}{t+1}\\ s^2_{(B,c),t+1} &= s^2_{(B,c),t} + \frac{\frac{t}{t+1}(B_{t+1} - \bar{B}_t)^2 - s^2_{(B,c),t}}{t+1}\\ s_{(B,c),t+1} &= s_{(B,c),t} + \frac{\frac{t}{t+1}(B_{t+1} - \bar{B}_t)(c_{t+1} - \bar{c}_t) - s_{(B,c),t}}{t+1}\end{aligned}$$

By only storing these intermediate statistics and updating them at the end of an epoch, we can continuously train our cost estimators.

The two estimators are used by CDHybrid at the start of every epoch (after the warm-up period) to estimate the cost of each algorithm, given the number of updates in the batch that just arrived. CDHybrid then chooses the algorithm that minimizes the expected cost to handle the batch of updates. Note that the cost of CD is estimated using the same model function as for CDStream. This implies that the amount of updates is also considered in the cost estimation of CD, even though the algorithm's performance is expected to be uncorrelated with this factor. While the value of β for CD's regressor is therefore likely to be around 0, we allow the model to take the factor into account in case our assumptions on CD are incorrect.

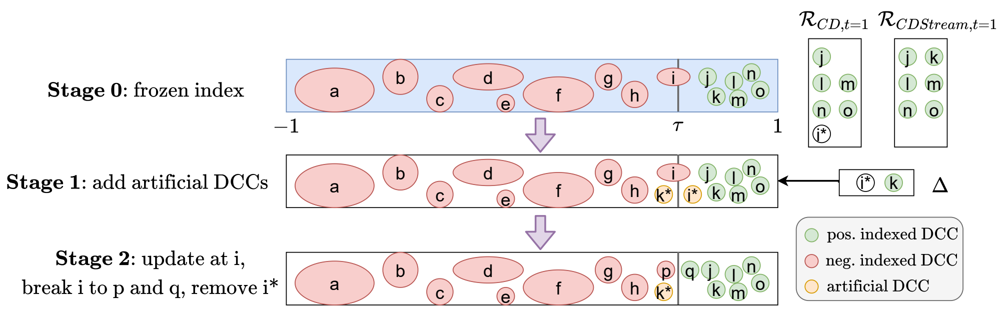
5.3 Switching between algorithms

Switching from CDStream to CD is fairly trivial. It mainly involves preparation of data structures which aid the (potential) switch back to CDStream in the future. After the selection for CD, we cache the current results of CDStream (referred to as $\mathcal{R}_{CDStream}$) and stop maintaining the index. Furthermore, we also cache the state of the pairwise correlations in case changes in the correlations of extrema pairs need to be considered. This applies to cases where a vector in the extrema pair receives an update, and its validity is checked by analyzing the correlation's direction of change (Related to line 8 of Alg. 2). After these caches CDHybrid transitions to running CD, which involves updating streams and passing them to CD for producing the result $\mathcal{R}_{CD}$.

Switching back from CDStream to CD is less trivial. Since the DCC Index was not updated for some time, we need to synchronize it so the state of the DCCs (i.e., positive or negative) conforms with the new correlations. This can be done through a full re-initialization or by updating the original DCCs to match the current state. The optimal strategy depends on the distance between the state of correlations before and after the period of CD. For simplicity, we will assume that correlation patterns go back to the way they were after an anomalous period, limiting the discrepancy between these two states. Therefore, we go with maintenance of the index instead of replacement.

There are multiple ways we can update the index to the new state. A naive approach would be to iterate over all DCCs, update their bounds and break them down if necessary. This approach is expensive, and requires all the work to be done within one epoch, which may delay updates on the result set. Furthermore, it might involve a lot of unnecessary checks on DCCs which remained unchanged and/or do not impact the result set. We instead use a lazy approach which focuses only on those DCCs that pose a threat to the correctness of the result set. We first compute the symmetric difference Δ between $\mathcal{R}_{CD}$ and $\mathcal{R}_{CDStream}$. Any result r contained in $\Delta \cap \mathcal{R}_{CDStream}$ is translated to a negative decisive combination, whereas any r contained in $\Delta \cap \mathcal{R}_{CD}$ leads to a new positive decisive combination. These new combinations are added in the index.

Example. If we find a vector combination $(\mathbf{v}_1, (\mathbf{v}_2, \mathbf{v}_3))$ in $\mathcal{R}_{CD}$ but not in $\mathcal{R}_{CDStream}$, we create an artificial cluster combination $(\{\mathbf{v}_1\}, (\{\mathbf{v}_2\}, \{\mathbf{v}_3\})$ and store it in the index as a positive DCC (object i^* in Fig. 5.1). Note that this action only accounts for cluster combinations in the index with materializations that actually changed state during the freeze. By artificially adding those combinations to the index we make sure future changes are detected, which would have been missed otherwise as their original DCCs are still indexed by their old extrema pairs. In our example, this implies that there may still exist a cluster combination $(\{\mathbf{v}_1, \mathbf{v}_5, \mathbf{v}_6\}, (\{\mathbf{v}_2\}, \{\mathbf{v}_3, \mathbf{v}_4\}))$ in the index, that is monitored as a negative DCC (object i in Fig. 5.1). These 'outdated' DCCs can be updated in a lazy manner, reconsidering them when the old extrema pairs are violated. This will cause the DCC to break, and the algorithm will try to add/remove some of its materializations to/from the result set. These materializations will already be in Δ, and will have been added to the index as an artificial DCC. In these scenario's we will prioritize the original DCC and delete the artificial DCC from the index. This involves process involves iterating over the symmetric difference before



Stage 0: frozen index
Stage 1: add artificial DCCs
Stage 2: update at i, break i to p and q, remove i*
-1
τ
1
$\mathcal{R}_{CD,t=1}$
$\mathcal{R}_{CDStream,t=1}$
Δ
pos. indexed DCC
neg. indexed DCC
artificial DCC

Figure 5.1: Visualization of lazy updating of DCCs in CDHybrid

adding/removing one or more vector combinations from the result set. We make sure that this is done at most once per DCC.

Artificial DCCs are effectively temporary solutions to guarantee correctness while lazily updating decisive combinations. There still exists an edge-case where an artificial DCC has ’moved to the same side’ as its original DCC before it is reconsidered. This causes the algorithm to fail in identifying the artificial DCC, allowing it to remain in the index. We cover this case by retaining Δ and labeling artificial DCCs to notice when this happens. If this happens, we still delete the artificial version as we prioritize larger DCCs.

The lazy approach of updating the DCC Index after a switch to CDStream requires significantly less work than the naive approach (which involves reconsidering all DCCs in the index). Still, it does add some switching cost to the algorithm. This cost is amortized through multiple epochs, which guards the update frequency of the result set short-term. To prevent switching costs from influencing the predictions on algorithm execution times, we refrain from updating the regression model during a short period after any switch.

Chapter 6

Evaluation

The following chapter analyzes the performance of CDStream in different situations. The purpose of the experiments was threefold: (a) to assess the scalability and efficiency of our methods, (b) to compare the 'maintenance' approach of CDStream with repeated execution of CD, and (c) to investigate opportunities for improvement.[1]

Before discussing the results, we present the experimental set-up, including description of the used datasets. Then, we discuss the results of the experiments one-by-one, starting at the evaluation of CDStream, ending at the evaluation of CDHybrid.

6.1 Experimental set-up

Multiple real-world datasets from different contexts were used for our experiments:

- **Stock.** Intraday tick data from the New York Stock Exchange (NYSE) including prices and quantities of individual trades taking place at February 13th 2009. The dataset considers trades at millisecond-level granularity for all stocks listed on NYSQ, AMES, NASDAQ, and SmallCap issues. During the peak hours, there are hundreds of trades for each stock in a second. Each stock had its own update frequency, with averages ranging from 10 msec to 1 sec. All prices were normalized with log-return normalization, as is standard in finance. This involves measuring the change in price from one point to another, and taking the natural logarithm of that value. The dataset was offered by Wharton Research Data Services [38], and is only available for partners of Wharton. The dataset was also used by Zhu et. al for their evaluation of StatStream [49].

- **ISD.** Segment of an overarching dataset from the National Centers for Environmental Information called the Integrated Surface Dataset (ISD) [37]. ISD includes worldwide surface weather observations from over 35,000 sensors, spread over the globe. Sensors measure a wide range of parameters such as atmospheric pressure, temperature, winds, ocean waves, and sea level pressure. For our experiments we use the measurements of two parameters; sea level pressure (SLP) and atmospheric temperature (Temp). We only consider measurements collected throughout the year 2000 by a diverse selection of sensors. Measurements were of an hour-level granularity with sensors having variable update frequencies. Pre-processing involved removing the sensors that did not have any arrivals for more than five consecutive days. This resulted in a total of 1898 available time-series of SLP data, and 2927 time-series with Temp data.

- **Crypto.** The Crypto dataset contained hourly closing prices of 7075 crypto-currencies between April 14, 2021 and July 13, 2021. Although currencies had a minimal update fre-

[1]We acknowledge that it would be best to compare CDStream to current state-of-the-art algorithms for multivariate correlation detection over streams. However, as noted earlier, these algorithms do not yet exist. Therefore, we deem CD as the state-of-the-art algorithm closest to our problem context.

quency of one hour, the data was of minute-level granularity, and arrivals took place throughout the whole hour. Currencies launched during the time span of the dataset were filtered out, resulting in 3937 time-series with a maximum of 2160 observations each. With cryptocurrency prices being effectively financial asset data, further pre-processing was identical to that of the Stock dataset. Data was retrieved through the CoinGecko API [6].

- **fMRI** Dataset containing functional MRI data of a person watching a movie, from the Naturalistic Neuroimaging Database of OpenNeuro.org [39]. The original data was already pre-processed by the owners for voxel-based analytics. This involved time-alignment of time series and normalization by 3-dimensional de-trending. We further pre-processed the data by mean-pooling with kernels of 2x2x2, 3x3x3, 4x4x4, 6x6x6 and 8x8x8 voxels, each representing the mean activity level at a cube of voxels. This was done in order to create meaningful subsets of the time series of sizes 237, 509, 1440, 3152, and 9700. Time series with constant activity level (i.e., variance equal to zero) were also removed. The resulting time series covering a period of 1.5 hours with data of second-level granularity. Due to the original pre-processing of the owners, time series are synchronized, meaning that every stream receives an update every second.

Previous works already demonstrated the usefulness of discovering correlations and/or multivariate correlations, on many of the above data types, e.g., [2, 3, 25, 49]. However, most of the above data was previously used for evaluation of one-shot algorithms, and not to simulate streams (excluding the stock-tick dataset). We acknowledge that datasets of second- to hour-level granularity are less applicable for evaluating algorithms that ought to handle data streams with millisecond velocity. Unfortunately, no such applicable datasets are openly available, to the best of our knowledge. Therefore, the bulk of the evaluation will be conducted with the Stock dataset, using the others solely to assess the context sensitivity of the algorithm. An overview of the datasets can be found in Table 6.1, including the default parameters used to simulate their streams. Default values for τ and δ were chosen for each dataset such that it resulted in a reasonable result set size (below 10k) for $\mathbf{mc}_{\text{size}}(1, 2)$ with default n.

6.1.1 Technicalities

To generate the streams from our datasets, we initialized sliding windows on the head of the time series, and fed the remaining values in a priority queue indexed by their timestamps. After completion of an epoch with a time-based model, a new batch was created by collecting all data from the queue indexed up to the next timepoint. In case of an arrival-based model, arrivals were read from the queue until a desired amount was reached. Sliding windows were moved based on the amount of simulated time passed (i.e., timestamps of data).

We evaluated the performance of CDStream by using both CDStream and CD (without incremental updates) to process batches, measuring the runtime. The primary performance indicator we use is average update time, which is the time taken to fully update the result set for a batch of updates (including sliding of windows and updating statistics). Experiments were conducted for a variety of query parameters to observe potential performance dependencies. Performance of CDHybrid was evaluated by simulating a common stream of arrivals followed by a period of impactful updates, running all three algorithm variations to analyze performance and evaluate the algorithm selection method of CDHybrid. The high-impact period was simulated by artificially increasing the batch-size for a certain time span (using a time-based batching model). Basic window sizes were decreased during those periods such that the ratio with the batch-size remained constant, preserving correlation quality. Batch-sizes were increased maximally, effectively having to set b_{bw} to 1. For example, for the Stock dataset, the timepoint value was set to 1000 during this period. Mind that this implies sliding windows are slid at every epoch. In contrast to other experiments, where the Stock dataset is used by default, the evaluation of CDHybrid focuses on the results on Crypto. This has to do with the fact that switching to CD at anomalous times is not always relevant for all datasets (as shown in Section 6.3).

Dataset	**Characteristics**		**Default parameters**								
	Time-series	**Time span**	**n**	**Time-point***	**b_{bw}**	**w**	τ	δ	**Digest agg.**	**Digest first val.**	**Epochs**
Stock	1906	8h	1000	1msec	1000 (1sec)	3600 (1h)	0.8	0.05	sum	0	5000 (10sec)
SLP	1898	1y	1000	1h	12 (0.5d)	180 (90d)	0.99	Irred	avg	last avg.	720 (30d)
Temp	2927	1y	1000	1h	12 (0.5d)	180 (90d)	0.99	Irred	avg	last avg.	720 (30d)
Crypto	5342	90d	1000	1min	60 (1h)	216 (9d)	0.8	0.05	sum	0	1440 (1d)
fMRI	9700	1.5h	1440	1sec	1sec	1800 (30min)	0.9	0.05	last	last value	1200 (20min)

* I.e., the time-granularity of the data, effective epoch-size.

Table 6.1: Dataset overview and configurations

Runtimes for epochs with no arrivals are also considered when computing average processing times, even though runtime is close to zero then, as they are relevant w.r.t. potential data congestion issues. The reported results correspond to averages after 10 repetitions. All experiments were executed on a server equipped with a 24-cores Intel Xeon Platinum 8260 Processor, and 400GB RAM.

6.2 CDStream evaluation

The following section discusses the effect of query parameters on the performance of CD and CDStream, and their ratio. Each experiment will focus on the impact of one parameter, by varying that parameter and keeping the others at their default value. Experimental parameters can be found in Table 6.2, with default values presented in Tables 6.1 and 6.2. Results of experiments will be discussed one by one, evaluating CDStream on three main aspects; (1) scalability and long-term performance, (2) prevention of data congestion (i.e., ability to handle streaming data), (3) relationship to CD. The same holds for the evaluation of CDHybrid.

6.2.1 Effect of correlation pattern and number of streams

The first experiment assessed the scalability of CDStream by measuring the mean processing time per batch for different correlation patterns, for subsets of the Stocks dataset ranging from 200 to 1900 randomly chosen time-series. Figure 6.1 shows that CDStream is significantly more efficient than CD for all correlation patterns, requiring fractions of milliseconds - generally several orders of magnitude less than CD. Consequently, CDStream's average runtime stays sub-ordinate to the batch-size up to $n = 500, \mathbf{mc}_{\text{size}}(1, 3)$, while CD never manages this. When average runtimes exceed the inter-arrival time (i.e., batch-size, shown with black dotted line in Fig. 6.1), arrival overflow will occur. In our context, we call this data congestion. These issues can be battled by increasing the batch-size (i.e., number of updates in arrival-based batching, time-period in time-based batching), allowing the algorithm to obtain efficiency gains from handling a bigger batch of arrivals. This strategy will work best for CD, as it is practically invariant to the amount of updates. In any way, increasing the batch-sizes implies the result set is updated at a slower

Parameter	**Values**
Dataset	**Stock**, SLP, Temp, Crypto, fMRI
k	1000, 2000, 3000, 4000, 5000
Corr. pattern	$\mathbf{mc}_{\text{size}}(\mathbf{1}, \mathbf{2})$, $\text{mc}_{\text{size}}(1, 3)$, $\text{mc}_{\text{size}}(2, 3)$
Vectors	200, 500, **1000**, 1500, 1900
Batch Model	**Time-based**, Arrival-based
Timepoint	**1**, 10, 50, 100, 200, 500, 1000
Batch size*	100, 200, 400, 800, 1000, 2000, 4000, 8000, 16000, 32000
Query type	**Threshold**, Top-k
τ	0.6, 0.7, **0.8**
δ	None, Irreducibility, **0.05**, 0.10, 0.15

*For arrival-based batching

Table 6.2: Experimental parameters (default values in bold)

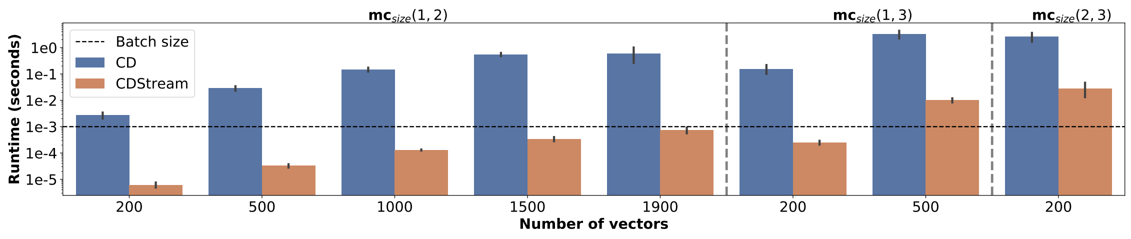


Figure 6.1: Effect of dataset size and correlation pattern

rate, which may delay the identification of significant events. Therefore, we can conclude that CDStream is able to offer high refresh rates for higher complexities, making it a more attractive streaming algorithm.

Also note that, even though the number of comparisons increases at a combinatorial rate with the number of time-series, execution time grows at a substantially slower rate. This can be attributed to the grouping techniques inside the DCC Index, which effectively reduces the work for processing each update. The difference between the average runtimes of CD and CDStream becomes smaller for larger correlations patterns, such as $\mathbf{mc}_{\text{size}}(2,3)$. Most likely, this has to do with the fact that large correlation patterns result in more DCCs with identical extrema pairs (i.e., more DCCs in same groups). Consequently, the violation of an extrema pair leads to relatively more DCCs that have to be checked, which is expensive. Variance of CDStream's runtime also seemingly increases as patterns become larger. This might be attributed to the fact that handling arrivals become more expensive, so the differences between epochs with and without arrivals become bigger.

Lastly, note that experiments are not shown for values of $n > 500$ for larger correlation patterns. This was because machine resources could not handle the memory complexities for these configurations, related to storing and indexing DCCs. From this we can conclude that space complexity it the limiting factor of CDStream. In contrast, CD was able to handle these complexities on the same machine, as it does not require to store DCCs (only the result set). Therefore, it can be used as a replacement for CDStream when answering such queries.

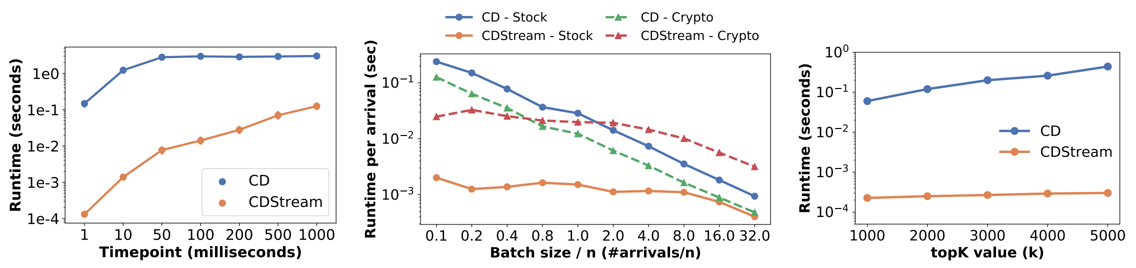
6.2.2 Effect of batch-size (time-based)

Figure 6.2a plots the average processing time per batch for varying timepoint values, i.e., the *batch-size*. Similar to the experimental setup of CDHybrid, b_{bw} is decreased as the timepoint value increases, such that their ratio remains constant. This is done to roughly preserve the meaning of correlations among different batch-sizes. Unfortunately, this also limits us to a maximum batch-size of 1000. Results for batches with more arrivals are discussed in the following subsection.

As predicted, we see that the runtime of CD is invariant to the timepoint value, and thus the amount of arrivals in a batch. In contrast, CDStream's runtime seems to increase linearly with the batch-size. This increase was actually expected to be sub-linear, as it was reasoned that a growing batch size may increase the probability that some vectors receive multiple updates in the same batch. In this case, updates would be aggregated[2] potentially cancelling each-other out. This would imply that doubling the batch size would roughly halve the time required for processing a single update. However, this does not seem to happen for this range of batch sizes. We conclude from this that CDStream has a relatively fixed processing time *per update*, meaning that one is free to choose the values for timepoint and b_{bw} that leads to the desired update rate of results. This observation partly lead to the choice of 1 msec as the default batch-size, as higher refresh rates are generally better for contexts where event reaction-time is key.

Despite this linear increase in runtime, CDStream outperforms CD for all batch-sizes. However, one might argue that the lines will intersect if the batch-size is increase further. This hypothesis was rejected by the experiment on the arrival-based batching model (Fig. 6.2b), discussed shortly.

[2]Within-batch aggregation of updates is done with the data's default aggregation method.



(a) Effect of batch size for time-based model (b) Effect of batch size for arrival-based model (c) Effect of k

Figure 6.2: Effect of query parameters on performance of CDStream

Lastly, note that both CD and CDStream have significantly smaller runtimes for a batch-size of 1 msec. This is due to the increased amount of epochs without any arrivals, resulting in a runtime of 0, which skews the average. Nevertheless, results are still valid and make a case for choosing the smallest batch-size possible.

6.2.3 Effect of batch-size (arrival-based)

The next experiment assessed the impact of batch-size when using an arrival-based batching model for both the Stocks and Crypto datasets. This model allows us to consider situations with more arrivals. Fig. 6.2a plots the average processing time *per update*, for varying batch sizes and for both datasets. The batch size (X-axis) is presented as a multiplicative factor on the number of considered streams n in each dataset ($n = 1000$). The results show that CDStream's runtime per arrival progressively decreases as the batch size increases. This implies that the initial expectation that CDStream's runtime increases sub-linearly still seems to be (partly) correct. Consequently, the batch size offers a tunable tradeoff between the algorithm's throughput and update rate of the results: increasing the batch size increases the efficiency, but also reduces the freshness of the results.

The relationships between runtimes and batch sizes appears to hold for both datasets. However, for Crypto, CD starts to outperform CDStream around a batch size of 600. This confirms the observation from Chapter 5 noting that running CD might be more efficient in times of highly significant updates, which lead to the development of CDHybrid. We further hypothesize from this that the relationship between CD and CDStream is context-dependent, but the effect of query parameters on CDStream's performance remains fairly constant. This hypothesis is further assessed in subsection 6.2.5

6.2.4 Effect of k

Figure 6.2c shows the average processing times for a top-k query $\mathbf{mc}_{\text{size}}(1, 2)$, with different values of k. We see that processing time for both algorithms increases with k. In CD, execution time grows almost linearly with k (from 60 to almost 600 msec) whereas for CDStream the time increases by roughly a factor of two for the same values. This discrepancy is attributed to the fact that CDStream only maintains the top-k solutions, already having a good estimate for the threshold of the top-k highest correlation from previous runs, whereas CD starts each run from scratch. Furthermore, CDStream's performance is negatively correlated with the value of k as the preferred size of the buffered result set was set as a multiple of k. Consequently, the cost of extracting the result set from the buffered set scales linearly with k.

Also note that CDStream's performance for top-k queries is generally worse compared to the average runtime for threshold queries. This is due to the fact that top-k prevents CDStream from pruning positive DCCs, as the (buffered) result set needs to be sorted and maintained at each non-empty batch. As a result, the complexity of top-k is effectively the same as that of threshold queries, with some additional overhead. However, this overhead is not excessive.

Dataset	Avg. arrivals	Avg. results	Avg. runtime* CD	Avg. runtime* CDS** (% of CD)	Avg. time* per arrival - CDS	Weighted Avg. DCC checks***
Stock	7.486	6169	1.478e-1	1.301e-4 (0.10%)	1.74e-5	5.52e3
SLP	106.22	5657	5.883e-1	1.049e-1 (17.83%)	9.88e-4	1.53e5
Temp	752.03	3803	7.230e-1	3.907e-1 (54.05%)	5.20e-4	1.85e5
Crypto	17.16	8790	1.417e0	9.574e-2 (6.76%)	5.58e-3	6.61e6
fMRI	1440.0	1948	1.611e0	9.662e-1 (60.00%)	6.71e-4	8.94e5

* Runtime in seconds
** CDStream
*** Average number of reconsidered DCCs, weighted by the #streams in the DCC

Table 6.3: Effect of dataset on CD and CDStream

For CD the relation between performance and query types is opposite; CD's performs better on top-k queries (for this parameter configuration). This can be attributed to the fact that the value for k is significantly smaller than the average amount of results of a threshold query with default parameters. Consequently, the top-k highest correlation lies higher than $\tau = 0.8$, enabling CD to prune more comparisons provided that it is able to rapidly increase its intermediate correlation threshold.

6.2.5 Effect of dataset

Table 6.3 presents the results of threshold queries on different datasets using the parameters configuration from Table 6.1. A general observation is that all statistics differ significantly between datasets, except for result set size and runtime of CD, which are fairly constant. The difference in average arrivals per epoch can be attributed to the diversity in time-granularities between datasets. Stock logically has a relatively small number of arrivals per epoch, as the timepoint size is set to only one millisecond. We conclude from the average arrival count that every stock has around 7.486 trades per second, resulting in a relatively high level of aggregation within basic windows. This is no issue considering the aggregation method; summing logreturns which effectively computes the change in price occurring within the basic window span. No information is lost this way. Both weather datasets have hour-level granularity, so the arrival rates tell us that sensor for sea level pressure measure once every 10 hours, and temperature sensors measure at a near hourly rate. The low arrival rate for Crypto can be explained by the fact that data is of minute-level granularity while considering currencies that are updated at an hourly rate. As a result, a single epoch contains updates for only a small subset of the currencies. Put differently, each stream has only one update per hour, but updates are spread over this hour. Recall from earlier that fMRI was pre-processed by the original authors, resulting in a global update model. This reflected by the arrival rate, being equal to n for that dataset. Result set sizes are very similar, as this was done by design from selecting the values for τ and δ to facilitate fair comparison between datasets.

Considering the average runtimes per batch, we see that both CD and CDStream have runtimes sub-ordinate to the epoch-sizes for all datasets (i.e., the time between two batches of arrivals), with the exception of CD for Stock and fMRI. More important, CDStream outperforms CD on all datasets, concluding that our methods are robust to different data contexts. The difference in runtime does vary heavily between datasets. Particularly, we see that CDStream performs best in situations where the number of arrivals is relatively small, which makes sense considering the difference in approach. Weighted average DCC checks of CDStream serves as good a indicator of arrival significance. Recall that arrival significance is defined as the extend in which new data violates existing decisive combinations, i.e., how many prior correlation patterns are disrupted. When arrivals disrupt more prior correlation patterns, more DCCs are violated, which means that maintaining the DCC Index requires more (computational) effort. We see that this statistic is also not constant over datasets. CDStream requires relatively the most amount of time to process updates on Crypto, which is positively correlated with the amount and size of the reconsidered DCCs. A reasonable explanation for this observation is that Crypto markets are known to be very volatile (i.e., often experience period of unpredictable, and sharp, price movements), and updates

	Runtime CD*			Runtime CDStream*		
δ/τ	**0.6**	**0.7**	**0.8**	**0.6**	**0.7**	**0.8**
None	4.02e-1	2.62e-1	1.68e-1	3.75-4	2.98e-4	0.87e-4
Irred.	3.21e-1	2.01e-1	0.88e-1	4.64-4	3.67e-4	1.31e-4
0.05	3.87e-1	2.55e-1	1.49e-1	4.32-4	3.43e-4	1.30e-4
0.10	3.25e-1	2.20e-1	1.08e-1	4.44-4	3.32e-4	1.23e-4
0.15	3.02e-1	2.12e-1	0.98e-1	4.36-4	3.31e-4	1.33e-4

* Average runtime in seconds

Table 6.4: Effect of τ and δ on CD and CDStream

Statistic	Value (% of above)
Vector combinations	5.00e8
Total DCCs	7.12e6 (1.428%)
Avg. DCCs checked*	1.60e4 (0.022%)
Avg. impactful DCCs**	1.81e0 (0.113%)
Avg. DCC Size***	7.02e0

* Measured only over epochs with arrivals
** DCCs that affect the result set
*** Vectors involved in a DCCs

Table 6.5: Vector combinations examined with CDStream on Stock dataset

are expected to be significant seeing that they encapsulate the market activity of an entire hour.

6.2.6 Effect of the correlation threshold and constraints

Table 6.4 shows the effect of τ and constraints (minimum jump and irreducibility) on CDStream's performance. We see that the absence of any additional constraint leads to better processing times for CDStream. This is as expected, as additional constraints merely lead to additional DCCs in the index, which leads to extra storage cost and more potential checks. When additional constraints are in effect, the exact configuration (i.e., irreducibility or δ value) has a negligible effect on the CDStream's performance. This makes sense as it does not affect the number of decisive combinations that need to be monitored.

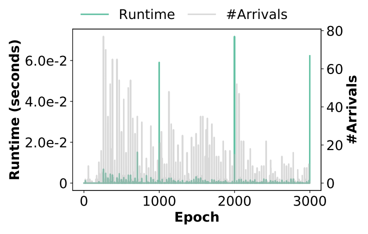
In contrast, an increasing value of τ leads to better performance for both algorithms, as CD is able to reach decisive combinations earlier. Specifically as the vast majority of the search space is usually not strongly correlated, a higher τ leads to DCCs with clusters at higher levels in the cluster tree, and thus overall larger and less DCCs. Besides, low τ leads to more positive DCCs which may ought to be indexed on minimum jump factors for CDStream. Cost analysis in Chapter 4 showed that both factors are favorable for CDStream's performance. We thus conclude that CDStream benefits from large τ and no additional constraints.

6.2.7 Long-term performance

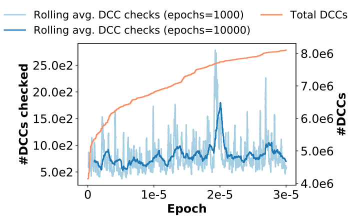
The next experiment was aimed at measuring the sustainability of CDStream's performance long-term, also providing insights into the overall behavior of the processing time over individual epochs. This was done by running CDStream for an extended period of 300,000 epochs, measuring the processing times as well as statistics on the DCCs in the Index and how many were checked. The results are shown through several sub-figures in Figure 6.3, focusing on both short-term and long-term measurements.

Considering the short-term development of runtime in Fig. 6.3a, we immediately note that there are three clear outliers, each at epochs which are multitudes of b_{bw}. This makes sense as these are the points where sliding windows are moved, resulting in a global update. Global updates are significantly more expensive for CDStream, as they lead to many DCC checks and breaks. This observation is again confirmed by Fig. 6.3b, where we see the total number of DCCs increasing in a stepwise manner. In-depth analysis showed that these stepwise increases typically happen when sliding windows are moved. We conclude from this that sufficient thought needs to go into selecting the sizes of both basic and sliding windows. Small basic window sizes might be desired to limit the level of aggregation within basic windows, but it may also lead to worse performance of CDStream, as global updates occur more frequently. We further note that the runtime is positively correlated with the number of arrivals, though difficult to see due to the outliers from global updates. This is in line with earlier experiments. There also seems to be a small peak in runtime around epoch 700, which cannot be explained by a global update or a significant peak in arrivals. Analysis of the run showed that this peak was due to garbage collection of the Java Virtual Machine (JVM).

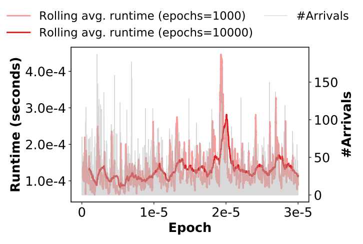
Focusing on the long-term measurements in Figures 6.3b and 6.3c, we note that the number of DCCs increases very fast in the earlier stages, converging later. Over the whole run, the number



(a) Development of runtime and arrivals (short-term)



(b) Development of DCCs (long-term)



(c) Long-term avg. runtime development

Figure 6.3: Development of runtime and arrivals (long-term)

of DCCs roughly doubles, which may cause one to worry about the sustainability of CDStream. Specifically, we concluded in the cost analysis in Chapter 4 that CDStream will suffer from a large amount of small DCCs. The increase in DCCs tells us that DCCs are being broken (i.e., splitting clusters), which leads to more DCCs of smaller size. However, we still do not conclude that the sustainability of CDStream is jeopardized as the rolling average of DCC checks shows only a small structural increase in checks. Additionally, the rolling averages of processing times in Fig. 6.3c tell us that this does not necessarily lead to a structural increase in processing time. One might still argue that even the long-term runs only span a period of 5 minutes, and that this structural increase in runtime might come later. While being valid criticism, we argue that this is unlikely to become a problem as the number of DCCs will probably continue to converge, stagnating this structural increase in runtime. Furthermore, even if the runtime eventually increases to 1 msec we can simply initialize the algorithm again, which will reset the DCCs in the index. Therefore, we can conclude that our methods will work over the long term.

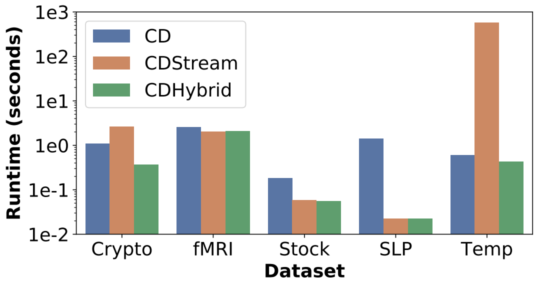
6.2.8 Pruning results

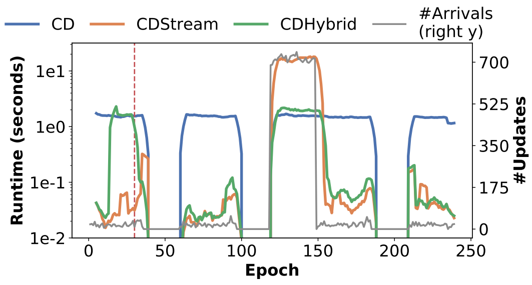
Table 6.5 presents a breakdown of the vector combinations pruned through the components of CDStream. By using CD as an initialization algorithm, CDStream reasons about cluster combinations instead of vector combinations, reducing the amount of objects to only 1.425% of the total search space. Then, through the triggering functionality of the DCC Index, CDStream only checks 0.022% of those cluster combinations as a non-empty batch arrives. In other words, CDStream effectively decreases the search space by six orders of magnitude. This observation explains the results of the discussed experiments, and is the essence of what allows CDStream to handle 'real' streaming data. However, also note that on average only 1.8 DCCs actually change the result set, which is another three orders of magnitude less than what we check. We thus conclude that there is still room for improvement in future work, focused on developing an algorithm with even higher pruning power.

6.3 CDHybrid evaluation

Our final two experiments examined the ability of CDHybrid to switch between CD and CDStream based on the stream properties. Recall from Section 6.1.1 that we simulated stream burst by speeding-up the updates (switching the values for timepoint and b_{bw}) around the middle of the stream, for an eighth of the length of the run. For fMRI we multiplied the epoch- and basic window size by 4 for the sake of experimentation, despite implying qualitatively worse correlations. Furthermore, we ran a total of 240 epochs for all datasets except Stock, for which we ran a total of 24000 epochs. The first 12.5% of the runs were reserved as warm-up time, in order to train the runtime regressors of CDHybrid.

Figure 6.4a shows the average processing times per algorithm for all datasets (excluding warm-up time). A more in-depth view of the behavior of the three algorithms is shown in Figure 6.4b, plotting rolling averages of processing times for the last 5 batches throughout time. The right Y axis shows the number of arrivals within each batch (i.e., epoch).





(a) Effect of dataset on CDHybrid. (b) Efficiency of CDHybrid on **Crypto**

Figure 6.4: Performance of CDHybrid

The results on different datasets (see Fig. 6.4a) show that CDHybrid automatically chooses the best approach, outperforming the other algorithms every time. In the case of Stock and SLP this means that it does not switch at all, running CDStream during arrival wave as it still outperforms CD then. The differences in performance for the Temp dataset are striking. Here, CDStream's runtime explodes during the arrival wave with excessive DCC breaks ruining the DCC Index. This results in excessive processing times also after the arrival wave. Our observation from Chapter 5 is thus confirmed that freezing the DCC Index does only protect performance during anomalous periods, it also improves the sustainability of the algorithm long-term.

Analyzing the development of runtimes and arrival rates for Crypto in Fig. 6.4b, we immediately note the clear cyclical pattern in the number of arrivals. Apparently prices of cryptocurrencies are updated at a fixed rate, with arrivals uniformly distributed over the first $\sim$ 40 min of each hour, and no new arrivals for the remaining 20 min. The wave of arrivals starts at epoch 120 and ends at epoch 150. We also observe that CDHybrid quickly switches to the best method, outperforming the other algorithms. In the epochs following a switch to CDStream, the cost of CDHybrid is marginally higher than CDStream that is attributed to the additional cost for updated the outdated index. After some time CDHybrid's performance is back in line with CDStream, indicating that the process of updating the index after the switch is not expensive. Also recall that part of this cost (for removing the expired decisive combinations from the index) is amortized through a large number of epochs, due to the lazy updating algorithm discussed in Chapter 5. Comparing CDHybrid to the non-adaptive algorithms, we see that CDStream suffers from erratic update-times during the arrival wave, almost failing to remain sub-ordinate to the batch-size at times. We thus conclude that it is sub-optimal at managing anomalous events. However, also mind that CDStream's high processing times are not solely caused by an increased amount of arrivals, sliding windows are also moved at every epoch, which causes some overhead. As expected, CD performance is uninfluenced by arrival rates.

Chapter 7

Conclusions

Prior work has shown the relevance of correlation analysis for streaming data. While most work on stream correlation discovery only concerns pairwise correlations, recent effort has shown the potential of multivariate correlations to lead to new insights. Efficient algorithms have been proposed to discover multivariate correlations, but they only focus on one-shot computations over static data. The current work was aimed at developing an algorithm that would also work on streaming data, as it may improve large-scale event monitoring systems.

Where prior stream correlation discovery solutions assume synchronous streams, we argued that that assumption is somewhat crude and generally does not hold in applications relevant to our context. Through the introduction of basic windows we proposed a method for handling asynchronous streams over which meaningful correlations can be computed in an efficient manner.

When considering related work, we concluded that few techniques applied to our problem. Most known streaming algorithms for pairwise correlations were extensions of fast one-shot algorithms to streaming contexts. Though, none of these streaming algorithms could be extended to multivariate correlations. Also, no existing one-shot algorithms for multivariate correlations could be extended to our streaming setting. Affiliated work fortunately offered a novel one-shot algorithm for multivariate correlations called *Correlation Detective*, that outperformed the prior state-of-the-art. As it considered the same goals and assumptions as the current work, it served as an ideal initialization algorithm of our solution.

We then proposed a solution called CDStream, which efficiently maintains the results of CD through incremental updating of correlations, and construction of an index that allows quick locating of impactful vector combinations. Cost analysis of the algorithm showed that cost is heavily dependent on the amount and size of the elements in the index. This also determines the algorithm's pruning power, in combination with properties of the arriving data, and hyperparameter values.

CDStream breaks down elements in the index as it queries it, theoretically increasing the cost over time which arguably makes it an unsustainable approach. We thus proposed methods to maintain and repair the index, in order to fight back against its degradation. These methods included continuous re-clustering of streams and freezing of the index at anomalous events, resorting to a repeated one-shot approach to continue reporting results.

Experiments showed that CDStream is able to handle streams with millisecond-level arrival rates, and is robust to data from different domains. Processing times of batches turned out to be strongly dependent on the amount of arrivals in the batch, and the occurrence of a window slide. Although the latter resulted in much alterations to the index of CDStream, endurance runs showed that degradation was limited and performance could be maintained long-term with the proposed techniques.

Concluding, the proposed algorithm serves as the first-steps towards a streaming algorithm for multivariate correlation discovery. Nevertheless, it still has its limitations which offers opportunities for future work. The following sections briefly discuss those factors.

7.1 Limitations and future work

Due to time constraints, not all issues related to CDStream could be solved, and not all ideas could be pursued. The following section addresses those issues, and provides suggestions for future work tangent to the problem definition of this study.

Issue 1: Memory complexity The first thing that became apparent in CDStream's evaluations is that it is limited by its memory complexity, making it unable to handle many streams and consider correlation patterns of high complexity (e.g., $\mathbf{mc}_{\text{size}}(2,3)$). Though the combinatorial complexity of the problem means that finding correlations for $n = 1000$ and $\mathbf{mc}_{\text{size}}(1,3)$ can already be considered challenging, many of the datasets included significantly more time series, which were not always manageable for CDStream. We therefore may argue that CDStream is not applicable to situations of 'real' big data (although that term is famously ambiguous), involving 10,000 to 100,000 streams. Future work should be focused on improving the memory complexity of the current approach, or develop an alternative approach altogether that uses less storage. Naturally, this will be challenging due to the inherently vast search space of the problem. Though maybe techniques could be developed that reason about vector combinations without having to store them, or a compressed representation of them (e.g., DCCs).

Issue 2: Absence of performance guarantees Although most experiments showed that CDStream was able to process batches faster than they were coming in, there were also situations where data congestion might have become an issue in the real world. For example, results on CDHybrid showed that neither CD or CDStream reported average processing times under the batch size when arrival rates were artificially increased (e.g., see results on Stock in Fig. 6.4a). Those issues might be battled by increasing the batch size, but together with the absence of a formal complexity analysis, our evaluation does not provide guarantees that CDStream or CDHybrid will work that well in all situations. This makes it a relatively risky choice for implementing in real world applications. Though, as there currently does not exist an alternative, the choice for CDHybrid is still better than no choice, even if it leads to relatively slow updates on the results. We may complete formal complexity analysis of CDStream in future work, such that reliable performance estimations can be offered based on data characteristics.

Issue 3: Caveats of using basic windows The introduction of basic windows and digests enabled us to drop the common assumption of synchronized streams. However, the technique does come with some limitations, most related to the quality (i.e., meaningfulness) of the correlations they make up. For instance, one may argue that the computation of correlations over (online) resampled time-series might lead to spurious relationships. This hypothesis is reinforced by the clear changes in correlations after windows are slid. If (multivariate) correlations in data are allegedly fairly constant over time, why does sliding of windows lead to so many breaks in DCCs? The argument that correlations over basic windows may differ from correlations over the original data (with missing value imputation) is true. However, computing correlations over artificially synchronized streams is not necessarily better. They too lead to spurious relations when too many values are imputed. The usage of basic windows with the correct aggregation method actually results in original values having an increased impact on correlation values. As long as we keep w large and b_{bw} relatively small, the difference in correlations will be limited between the two methods.

A second point of potential criticism is that there still exists some factor of time-misalignment between the values over which correlations are computed, particularly between the running basic windows. If one time-series s_1 receives an update at timepoint t and another time series s_2 does not, the digest $S_2[L]$ will be outdated with respect to $S_1[L]$. This will lead to different correlations compared to if we forward-filled the last value of s_2 and used averaging as an aggregation method. However, this does not happen when we use sum or last-value as an aggregation method, and updating averages with imputed values on itself is also questionable. Our method simply ensures that we are not reconsidering the correlation between streams that all did not receive an update, which is something related work does do. We thus argue that future work should be focused on optimizing the handling of global updates with the current methods, instead of resorting back to assuming synchronized streams.

Other suggestions for future work The following presents other suggestions for future work, which are briefly discussed.

- **Alternative correlation measures** CDStream currently only supports the detection of vector combinations with high multiple correlation. However, future research on extending the current methods to alternative correlation measures such as Multipoles, Total Correlation, and Canonical Correlation Analysis. This work would involve the extension of Correlation Detective to supporting these measures, and analysis of the dependencies of correlation bounds of cluster combinations, such that they can be indexed on those factors.

- **Decaying window model** Recall from Chapter 2 that there are several commonly used temporal spans for computing statistics over streams. We chose to use sliding windows over the landmark model and damped window model due to its generality and vast theoretical background. However, the damped window model might be more applicable for financial applications, as it weighs recent observations more heavily than old ones. In future work, we could implement this model and analyze the differences in results with those of the sliding window model. This implementation should not be difficult as the incremental updating of statistics involves simply computing the weighted sum of the change in digest value and the old statistic.

- **Support of negative correlations** Recall the notion in Chapter 2 that some applications (additionally) require the result set to contain combinations of vectors with **mc** coefficients *lower* than some threshold $\tau \in [-1, 1]$, or merely those with the *lowest* correlation. Along with this notion, we argued that support of such thresholds would involve trivial modifications to the presented theory and methods. Namely, if one looks to find all subsets of $\mathcal{S}$ with an **mc** coefficient *lower* than some threshold $\tau \in [-1, 1]$, CDStream would simply have to be configured such that positive DCCs are indexed on their UB instead of their LB, and negative DCCs are indexed on their LB instead of their UB. As long as the value of τ is set such that the result set size is comparable to the results reported in Chapter 6, CDStream is expected to report similar results.
  Support of both a lower and higher correlation threshold would involve indexing negative DCCs on both their UB and LB. This modification is expected to worsen the performance of CDStream somewhat, as negative DCCs will have to be checked more often.
  Modifications to CD for such thresholds would only involve re-configuration of the constraints used to determine the state of a cluster combination. For example, given we look to find all subsets with a **mc** coefficient lower than some threshold $\tau_{\text{lower}} = -0.95$ or higher than some $\tau_{\text{higher}} = 0.95$, we would consider cluster combinations with both bounds above τ_{higher} or below τ_{lower} positive DCCs, those with bounds in $[-0.95, 0.95]$ negative DCCs, and all others *indecisive*. It would be interesting to see if our hypotheses on the performance of such extensions are confirmed. Therefore, we leave the implementation of the above modifications for future work.

- **Case study** The evaluation in the current study involves running our algorithms on *simulations* of streams originating from different domains. However, it would be interesting to apply the methods to a real world application to see if it manages to process the data as well as offer interesting insights. For example, we could implement the algorithms in flash-trading applications and see if it is able to spot profitable opportunities that it would not be able to identify without it.

Bibliography

[1] Bitcoin falls after Elon Musk tweets breakup meme. https://www.cnbc.com/2021/06/04/bitcoin-falls-after-elon-musk-tweets-breakup-meme.html, 2021. Accessed: 2021-07-13. 32

[2] Saurabh Agrawal, Gowtham Atluri, Anuj Karpatne, William Haltom, Stefan Liess, Snigdhansu Chatterjee, and Vipin Kumar. Tripoles: A new class of relationships in time series data. In *Proceedings of the 23rd SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 697–706. ACM, 2017. 1, 2, 9, 13, 37

[3] Saurabh Agrawal, Michael Steinbach, Daniel Boley, Snigdhansu Chatterjee, Gowtham Atluri, Anh The Dang, Stefan Liess, and Vipin Kumar. Mining novel multivariate relationships in time series data using correlation networks. *IEEE TKDE*, 32(9):1798–1811, 2020. 1, 2, 37

[4] I. Antonioua, V. Ivanova, Va. V. Ivanovb, and P. V. Zrelova. On the log-normal distribution of stock market data. 2015. 12

[5] Nurjahan Begum, Liudmila Ulanova, Jun Wang, and Eamonn Keogh. Accelerating dynamic time warping clustering with a novel admissible pruning strategy. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015-August:49–58, 2015. 13

[6] CoinGecko. Coingecko api. https://www.coingecko.com/en/api, 2021. Accessed: 2021-07-13. 37

[7] Richard Cole, Dennis Shasha, and Xiaojian Zhao. Fast window correlations over uncooperative time series. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (August 2017):743–749, 2005. 11

[8] Abhimanyu Das and David Kempe. Algorithms for subset selection in linear regression. In *Proc. 40th ACM Symposium on Theory of Computing*, STOC '08, page 45–54. ACM, 2008. 14

[9] Hoang Anh Dau, Nurjahan Begum, and Eamonn Keogh. Semi-supervision dramatically improves time series clustering under Dynamic Time Warping. *International Conference on Information and Knowledge Management, Proceedings*, 24-28-October-2016:999–1008, 2016. 13

[10] Simons Foundation. SPARK for autism. https://sparkforautism.org/portal/page/autism-research/, 2021. Accessed: 2021-07-30. 1

[11] Simons Foundation. SPARK gene list. https://d2dxtcm9g2oro2.cloudfront.net/wp-content/uploads/2020/07/13153839/SPARK_gene_list_July2020.pdf, 2021. Accessed: 2021-07-30. 1

[12] Pierre-Alexis Gros, Hervé Le Nagard, and Olivier Tenaillon. The Evolution of Epistasis and Its Links With Genetic Robustness, Complexity and Drift in a Phenotypic Model of Adaptation. *Genetics*, 182(1):277–293, 05 2009. 1

[13] Tian Guo, Saket Sathe, and Karl Aberer. Fast distributed correlation discovery over streaming time-series data. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, page 1161–1170, New York, NY, USA, 2015. Association for Computing Machinery. 2, 6

[14] Daniel A. Handwerker, Vinai Roopchansingh, Javier Gonzalez-Castillo, and Peter A. Bandettini. Periodic changes in fmri connectivity. *NeuroImage*, 63(3):1712–1719, 2012. 1

[15] H. Hasan Örkcü. Subset selection in multiple linear regression models: A hybrid of genetic and simulated annealing algorithms. *Applied Mathematics and Computation*, 219(23):11018–11028, 2013. 14

[16] Zhiyong Huang, Hua Lu, Beng Chin Ooi, and Anthony K.H. Tung. Continuous skyline queries for moving objects. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1645–1658, 2006. 3

[17] Wolfgang Karl Härdle. *Applied Multivariate Statistical Analysis*, pages 321–330. Springer, 2007. 2, 14

[18] W. Johnson. Extensions of lipschitz mappings into hilbert space. *Contemporary mathematics*, 26:189–206, 1984. 11

[19] J.F. Kenney and E.S. Keeping. *Mathematics of Statistics*, volume 3, chapter 15, pages 252–285. NJ: Van Nostrand, Princeton, 1962. 33

[20] Chungho Lee and Incheon Paik. Stock Market Analysis from Twitter and News Based on Streaming Big Data Infrastructure. (iCAST):312–317, 2017. 6

[21] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2nd edition, 2014. 11

[22] Xiang Lian, Lei Chen, and Bin Wang. *Approximate Similarity Search over Multiple Stream Time Series*, volume 5005 LNCS. 2008. 6, 11

[23] Xiang Lian, Lei Chen, Jeffrey Xu Yu, Jinsong Han, and Jian Ma. Multiscale representations for fast pattern matching in stream time series. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):568–581, 2009. 12

[24] Silvan Licher, Shahzad Ahmad, Hata Karamujić-Čomić, Trudy Voortman, Maarten J. G. Leening, M. Arfan Ikram, and M. Kamran Ikram. Genetic predisposition, modifiable-risk-factor profile and long-term dementia risk in the general population. *Nature Medicine*, 25(9):1364–1369, 2019. 1

[25] Stefan Liess, Saurabh Agrawal, Snigdhansu Chatterjee, and Vipin Kumar. A teleconnection between the west siberian plain and the enso region. *Journal of Climate*, 30(1):301 – 315, 2017. 1, 37

[26] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD '03*, pages 2–11, 2003. 12

[27] Jessica Lin, Eamonn Keogh, and Wagner Truppel. Clustering of streaming time series is meaningless. *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD '03*, pages 56–65, 2003. 12, 15

[28] Myles E. Mangram. A simplified perspective of the markowitz portfolio theory. *Global Journal of Business Research*, 7(1):59–70, 2013. 1

[29] Koen Minartz, Jens D'Hondt, and Odysseas Papapetrou. Multivariate correlation discovery in static and streaming data. Technical report. Source. vi, vi, 2, 3, 15, 16, 17

[30] Ileena Mitra, Alinoë Lavillaureix, Erika Yeh, Michela Traglia, Kathryn Tsang, Carrie E. Bearden, Katherine A. Rauen, and Lauren A. Weiss. Reverse pathway genetic approach identifies epistasis in autism spectrum disorders. *PLOS Genetics*, 13(1):1–27, 01 2017. 1

[31] Douglas Montgomery. *Applied statistics and probability for engineers*. Wiley, Hoboken, NJ, 2011. 14

[32] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *Proc. ACM International Conference on Management of Data*, SIGMOD '10, page 171–182. ACM, 2010. 10, 11, 22

[33] Ethan Namvar and Lawrence Harris. The Economics of Flash Orders and Trading. *SSRN Electronic Journal*, pages 1–17, 2012. 3

[34] Hoang Vu Nguyen, Emmanuel Müller, Periklis Andritsos, and Klemens Böhm. Detecting correlated columns in relational databases with mixed data types. In *Proc. 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14. ACM, 2014. 2, 14

[35] Hoang Vu Nguyen, Emmanuel Müller, Jilles Vreeken, Pavel Efros, and Klemens Böhm. Multivariate maximal correlation analysis. In *Proc. 31st International Conference on Machine Learning - Volume 32*, ICML'14, pages 775–783, 2014. 2

[36] Philon Nguyen and Nematollaah Shiri. Fast correlation analysis on time series datasets. *International Conference on Information and Knowledge Management, Proceedings*, pages 787–795, 2008. 13

[37] National Oceanic and Atmospheric Administration. NOAA integrated surface dataset (global). `https://www.ncei.noaa.gov/access/search/dataset-search`, 2021. Accessed: 2021-07-30. 36

[38] Wharton University of Pennsylvania. Wharton Research Data Services. `https://wrds-www.wharton.upenn.edu/`, 2021. Accessed: 2021-08-25. 36

[39] OpenNeuro. Naturalistic neuroimaging database. `https://openneuro.org/datasets/ds002837/versions/2.0.0`, 2021. Accessed: 2021-08-25. 37

[40] Örjan Carlborg and Chris S. Haley. Epistasis: too often neglected in complex trait studies? *Nature Reviews Genetics*, 5(8):618–625, 2004. 1

[41] Odysseas Papapetrou and Minos Garofalakis. Monitoring distributed fragmented skylines. *Distributed and Parallel Databases*, 36(4):675–715, 2018. 3

[42] Liudmila Ulanova, Nurjahan Begum, Mohammad Shokoohi-Yekta, and Eamonn Keogh. *Clustering in the Face of Fast Changing Streams*, pages 1–9. 2016. 13

[43] VLDB. VLDB 2022 - 48th International Conference on Very Large Data Bases. `https://vldb.org/2022/`, 2021. Accessed: 2021-08-05. 3

[44] Satosi Watanabe. Information theoretical analysis of multivariate correlation. *IBM Journal of Research and Development*, 4(1):66–82, 1960. 2

[45] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proc. International Conference on Management of Data*, SIGMOD'19, page 1223–1240. ACM, 2019. 1

[46] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013. 16

[47] Tiancheng Zhang, Dejun Yue, Yu Gu, Yi Wang, and Ge Yu. Adaptive correlation analysis in stream time series with sliding windows. *Computers and Mathematics with Applications*, 57(6):937–948, 2009. 11

[48] Xiang Zhang, Feng Pan, Wei Wang, and Andrew Nobel. Mining non-redundant high order correlations in binary data. *Proc. VLDB Endow.*, 1(1):1178–1188, 2008. 1, 2, 6

[49] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. 28th International Conference on Very Large Data Bases*, VLDB '02, page 358–369, 2002. 3, 5, 6, 7, 10, 11, 36, 37

Appendix A

Incremental updating of running statistics on asynchronous streams

This appendix contains the full derivations for incremental updating of the running statistics discussed in Chapter 4.
$\mathbb{E}\left[s_x\right]$ can be incrementally updated as follows;

$$\begin{aligned}
\mathbb{E}\left[s_x\right]_{t+1} &= \frac{1}{w}\sum_{i=L-w+1}^{L} S[i] \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} (S[i]) + S[L]_{t+1}\right) \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} (S[i]) + S[L]_t - S[L]_t + S[L]_{t+1}\right) \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} (S[i]) + S[L]_t\right) + \frac{1}{w}\left(S[L]_{t+1} - S[L]_t\right) \\
&= \mathbb{E}\left[s_x\right]_t + \frac{1}{w}\left(S[L]_{t+1} - S[L]_t\right)
\end{aligned}$$

$\mathbb{E}\left[S^2\right]$ can be incrementally updated as follows;

$$\begin{aligned}
\mathbb{E}\left[s_x^2\right]_{t+1} &= \frac{1}{w}\sum_{i=L-w+1}^{L} S[i]^2 \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} \left(S[i]^2\right) + S[L]_{t+1}^2\right) \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} \left(S[i]^2\right) + S[L]_t^2 - S[L]_t^2 + S[L]_{t+1}^2\right) \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} \left(S[i]^2\right) + S[L]_t^2\right) + \frac{1}{w}\left(S[L]_{t+1}^2 - S[L]_t^2\right) \\
&= \mathbb{E}\left[s_x^2\right]_t + \frac{1}{w}\left(S[L]_{t+1}^2 - S[L]_t^2\right)
\end{aligned}$$

$\mathbb{E}\left[s_x s_y\right]$ can be incrementally updated as follows;

$$\begin{aligned}
\mathbb{E}\left[s_x s_y\right]_{t+1} &= \frac{1}{w} \sum_{i=L-w+1}^{L} S_x[i]S_y[i] \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} (S_x[i]S_y[i]) + S_x[L]_{t+1}S_y[L]_{t+1}\right) \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} (S_x[i]S_y[i]) + S_x[L]_t S_y[L]_t - S_x[L]_t S_y[L]_t + S_x[L]_{t+1}S_y[L]_{t+1}\right) \\
&= \frac{1}{w}\left(\sum_{i=L-w+1}^{L-1} (S_x[i]S_y[i]) + S_x[L]_t S_y[L]_t\right) + \frac{1}{w}\left(S_x[L]_{t+1}S_y[L]_{t+1} - S_x[L]_t S_y[L]_t\right) \\
&= \mathbb{E}\left[s_x s_y\right]_t + \frac{1}{w}\left(S_x[L]_{t+1}S_y[L]_{t+1} - S_x[L]_t S_y[L]_t\right)
\end{aligned}$$

Appendix B

Pseudo-code algorithm for storing and indexing DCCs

The following pseudo-code algorithm represents the process of indexing and storing DCCs. Note that it only shows how positive DCCs of the first bound case are stored. However, the process can be modified trivially allow indexing of positive and negative DCCs of other bound cases, by changing the argmin/argmax terms in lines 3,10,17 to the relevant terms from the bounds in Theorem 3.3.1.

```
Algorithm 6: INDEX($\mathcal{S}_l, \mathcal{S}_r, \mathcal{I}^+, \mathcal{I}^-$)
  Input: A decisive cluster combination with sets of clusters $\mathcal{S}_l$ and $\mathcal{S}_r$, the two DCC
         Indices $\mathcal{I}^+$ and $\mathcal{I}^-$.
  // Index on LHS extrema pairs
1 for $C_1 \in \mathcal{S}_l$ do
2     for $C_2 \in \mathcal{S}_l \setminus \{C_1\}$ do
3         $\langle \mathbf{a}, \mathbf{b} \rangle \leftarrow \arg\max_{i \in C_1, j \in C_2} \{\rho(i,j)\}$
4         for $s_i \in C_1$ do
5             $\mathcal{I}^+[i][\langle \mathbf{a}, \mathbf{b} \rangle][C_2]$.ADD$((\mathcal{S}_l, \mathcal{S}_r))$
6         for $s_j \in C_2$ do
7             $\mathcal{I}^+[j][\langle \mathbf{a}, \mathbf{b} \rangle][C_1]$.ADD$((\mathcal{S}_l, \mathcal{S}_r))$
  // Index on RHS extrema pairs
8 for $C_1 \in \mathcal{S}_r$ do
9     for $C_2 \in \mathcal{S}_r \setminus \{C_1\}$ do
10        $\langle \mathbf{a}, \mathbf{b} \rangle \leftarrow \arg\max_{i \in C_1, j \in C_2} \{\rho(i,j)\}$
11        for $s_i \in C_1$ do
12            $\mathcal{I}^+[i][\langle \mathbf{a}, \mathbf{b} \rangle][C_2]$.ADD$((\mathcal{S}_l, \mathcal{S}_r))$
13        for $s_j \in C_2$ do
14            $\mathcal{I}^+[j][\langle \mathbf{a}, \mathbf{b} \rangle][C_1]$.ADD$((\mathcal{S}_l, \mathcal{S}_r))$
  // Index on Between-side extrema pairs
15 for $C_1 \in \mathcal{S}_l$ do
16     for $C_2 \in \mathcal{S}_r$ do
17         $\langle \mathbf{a}, \mathbf{b} \rangle \leftarrow \arg\min_{i \in C_1, j \in C_2} \{\rho(i,j)\}$
18         for $s_i \in C_1$ do
19             $\mathcal{I}^-[i][\langle \mathbf{a}, \mathbf{b} \rangle][C_2]$.ADD$((\mathcal{S}_l, \mathcal{S}_r))$
20         for $s_j \in C_2$ do
21             $\mathcal{I}^-[j][\langle \mathbf{a}, \mathbf{b} \rangle][C_1]$.ADD$((\mathcal{S}_l, \mathcal{S}_r))$
```