Fast Approximate Correlation for Massive Time-series Data

Abdullah Mueen
UC Riverside
mueen@cs.ucr.edu

Suman Nath
Microsoft Research
sumann@microsoft.com

Jie Liu
Microsoft Research
liuj@microsoft.com

ABSTRACT

We consider the problem of computing all-pair correlations in a warehouse containing a large number (e.g., tens of thousands) of time-series (or, *signals*). The problem arises in automatic discovery of patterns and anomalies in data intensive applications such as data center management, environmental monitoring, and scientific experiments. However, with existing techniques, solving the problem for a large stream warehouse is extremely expensive, due to the problem's inherent quadratic I/O and CPU complexities.

We propose novel algorithms, based on Discrete Fourier Transformation (DFT) and graph partitioning, to reduce the end-to-end response time of an all-pair correlation query. To minimize I/O cost, we partition a massive set of input signals into smaller batches such that caching the signals one batch at a time maximizes data reuse and minimizes disk I/O. To reduce CPU cost, we propose two approximation algorithms. Our first algorithm efficiently computes approximate correlation coefficients of similar signal pairs within a given error bound. The second algorithm efficiently identifies, without any false positives or negatives, all signal pairs with correlations above a given threshold. For many real applications, our approximate solutions are as useful as corresponding exact solutions, due to our strict error guarantees. However, compared to the state-of-the-art exact algorithms, our algorithms are up to 17× faster for several real datasets.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval

General Terms

Algorithm, Performance

Keywords

Correlation Matrix, Discrete Fourier Transform

1. INTRODUCTION

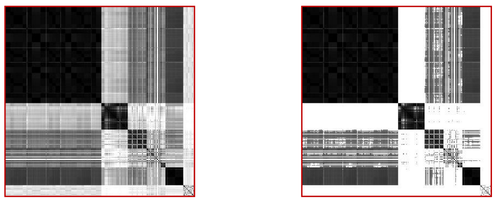
The increasing instrumentation of physical and computing processes has given us unprecedented capabilities to collect massive volumes of data. Applications for data center management, environmental monitoring, financial engineering, scientific experiments, and mobile asset tracking produce massive time series streams (or, *signals*) from various (physical and virtual) sensors. Such applications typically require stream warehousing systems (SWS) that, unlike typical data stream systems, can archive data for a long period of time and efficiently support various statistical and data mining queries on historic data.

Minimizing response times of ad-hoc queries in an SWS is very important for effective user interaction. However, achieving this is extremely challenging as (i) the volume of the data of interest is massive, and (ii) the benefit of pre-processing techniques (such as materialized views and indices) can be minimal since the queries are composed on-the-fly. We use a *data center management application* as the running example in the paper. Since data centers are large capital investments for online service providers, they are closely monitored for operating conditions and utilizations by collecting various software and hardware performance counters (e.g., a server's CPU and memory utilization, an application's response time, etc.). A typical SWS, such as DataGarage [16], monitors multiple data centers, each of which may contain tens of thousands of servers. Assume that 500 performance counters are collected from each server. Then, data centers with 100,000 servers will yield 50 million concurrent data streams and, with a mere 15-second sampling rate, more than 30 billion records (or about 1TB data) a day. While mining historical data over several months for tasks such as capacity planning, workload placement, pattern discovery, and fault diagnostics seems appealing, the sheer volume of the data can make useful ad-hoc data mining queries impractically slow.

Recent work [20] has shown efficient techniques for compressing data and running simple queries in a SWS. In this paper, we consider a more complex query of computing a correlation matrix: given n signals of equal length m, compute a $n \times n$ matrix C such that $C[i, j], 1 \leq i, j \leq n$ is the correlation coefficient $corr(i, j)$ of signals i and j. We consider Pearson's correlation coefficients; given two signals $\mathbf{x}$ and $\mathbf{y}$ of equal length m, with averages μ_x and μ_y and standard deviations σ_x and σ_y respectively, their Pearson correlation coefficient is defined as,

$$corr(\mathbf{x}, \mathbf{y}) = \frac{1}{m} \sum_{i=0}^{m-1} \left(\frac{x_i - \mu_x}{\sigma_x} \right) \left(\frac{y_i - \mu_y}{\sigma_y} \right)$$

Previous works have shown the importance of computing cross correlation of a large number of signals. In [20], authors mention five important queries in a data center management application, out of which three queries related to server dependency analysis, load balancing, and anomaly detection involve computing correlation matrices. For example, Figure 1(a) shows a correlation matrix of 350 signals, where each signal represents the number of TCP con-



(a) Correlation Matrix C (b) Threshold Corr. Matrix C^T

Figure 1: Correlation matrices. Darker pixels represent higher correlation coefficients (e.g., a black pixel represents a corr. coeff. of 1).

nections in a server measured every 30 seconds in a day. The matrix shows the existence of several load balanced clusters of machines. If such a correlation matrix is periodically computed, any deviation between successive matrices can indicate possible anomalies. Similarly, many eScience questions are better understood by correlation matrices [9]. In [21] and [27], authors provide examples of sensing and stock trading applications requiring cross correlation of tens of thousands of signals.

We consider ad-hoc queries, where individual signals are stored on disks and a target set of signals and a target time window are defined during the query time. We assume that no precomputed index specifically designed for correlation computation (such as [1, 27]) exists because of its large overhead and inefficiency to handle ad-hoc queries. Fast computation of a large correlation matrix in this setting is challenging because of its high I/O and CPU overhead. High I/O costs are encountered because, due to limited memory, signals may need to be read from disk to memory multiple times. High CPU costs result from examining all pairs of signals and doing expensive floating point operations. As we will show in Section 3, computing a correlation matrix for 10,000 signals of length 2880 using a naive approach can take several hours in a standard desktop computer. If multiple such matrices are to be computed (e.g., one matrix for each day of signals), this can take up to several days on a single machine. Such large response times are unacceptable for interactive data exploration tasks in a SWS.

To address the above challenges, we consider a slightly relaxed, yet almost equally useful, version of the problem. In many applications, including the data center management application, users are typically interested in *correlated* (e.g., correlation above a given threshold T) signal pairs—uncorrelated signal pairs are typically not of much interest. Therefore, while the exact correlations of correlated pairs need to be computed, that of uncorrelated pairs can often be safely ignored. Therefore, the general problem we consider in this paper is as follows.

PROBLEM 1 (THRESHOLD CORRELATION MATRIX). *Given n signals of equal length m and a threshold T, for $1 \leq i, j \leq n$, compute a $n \times n$ threshold correlation matrix C^T such that $C^T[i,j] = corr(i,j)$ if $|corr(i,j)| \geq T$ and 0 otherwise.*

Figure 1(b) shows an example of threshold correlation matrix with a threshold of $T = 0.5$. As shown, some of the gray pixels, with correlation < 0.5 in Figure 1(a) are absent in Figure 1(b); yet, Figure 1(b) shows the prominent clusters of similar signals.

To reduce I/O costs, we propose a novel data partitioning algorithm that divides a given dataset into batches such that each batch fits in the available memory. More importantly, batches are carefully created such that most signal pairs within a single batch are correlated, while many pairs with signals in different batches are uncorrelated. Thus, as signals are read into the memory one batch at a time, signals that are mutually correlated with each other reside in the memory at the same time—in this way, the correlation coefficients of a cached signal with a large number of other cached signals can be computed without additional I/Os. We show that the problem can be modeled as a graph partitioning problem and be solved using efficient heuristics.

To reduce CPU costs, we propose two novel approximation algorithms for computing a correlation matrix. Our first approximation algorithm computes correlation coefficients within a given error bound, while the second algorithm identifies signal pairs with correlation above a threshold without any false positives or false negatives. Our algorithms take the advantage of computational shortcuts in the Discrete Fourier Transformation (DFT) space and are significantly faster than an exact algorithm; yet, their approximation guarantees keep them useful for many real-world applications, including our running example of the data center management application.

Our approximation algorithms can be used in combination (with or without an exact algorithm) during interactive data exploration. For example, a data center operator can first use our first approximation algorithm for identifying highly correlated signal pairs. If the correlation pattern looks interesting, he can further use our second approximation algorithm to remove any false positives. If further needed, he can use an exact algorithm to compute exact correlations. Such an approach gives a user the flexibility to stop data exploration early (e.g., after the first step) or continue for greater fidelity answers at the cost of increasing computational complexity.

In summary, we make the following contributions.

- We propose a novel caching algorithm for computing a threshold correlation matrix. The algorithm uses DFT and graph partitioning to optimize overall I/O cost (§ 4).
- We propose two efficient approximation algorithms. The first algorithm approximates entries in a threshold correlation matrix within a given error bound. The second algorithm efficiently identifies all correlated signal pairs without any false positives or negatives (§ 5).
- We propose extensions to our basic algorithms to support anti-correlation and lagged correlation (§ 6).
- Using several real datasets, we evaluate our proposed algorithms. Our evaluation shows that our algorithms are up to $17\times$ and $71\times$ faster than existing algorithms for synchronous and lagged correlation respectively (§ 7).

This work is a part of a broader project, called DataGarage [16], for building a data-driven data center management system. DataGarage aims to collect and archive monitoring data from tens of thousands of servers, to enable users to run ad-hoc and routine data mining queries on massive data, and to provide useful control feedback to data center operators for load balancing, energy optimization, anomaly detection, etc. In that context, this work provides an important building block to mine massive data and to gather valuable insights for data center operators.

2. RELATED WORK

Correlation is a similarity measure and prior works have extensively considered the problem of discovering similar sequences from a large number of sequences. Traditionally, the Euclidean distance is used to capture the similarity. The original work by Agrawal el at. [1] considered discovering similarity between an online sequence and an indexed database of previously obtained sequence information. The proposed techniques focused on whole sequence matching and utilized the DFT to transform data from the time domain into frequency domain and used multidimensional index structure to index the first few DFT coefficients. The work was later generalized to allow subsequence matching [4] and transformations such as scaling and shifting [19]. Our work differs from them in that i) we consider Pearson correlation coefficient, which is optimal for detecting linear relationships, ii) we do not assume existence of any precomputed index, since our sequence set and the target time window are defined ad-hoc during query time, and iii) we consider computing correlation of all sequence pairs, instead of only the pairs involving a given sequence. From algorithmic point of view, our caching and approximation algorithms are novel and were not considered by this previous work.

StatStream [27] and HierarchyScan [12], like our work, consider correlation coefficients as a similarity measure. StatStream uses DFT to maintain a grid data structure to quickly identify similar streams within a sliding window in real-time. HierarchyScan considers a stream warehouse setting and performs correlation between the stored sequences and a given template pattern in the transformed domain (e.g., using DFT or DWT) hierarchically. [2] uses sketches to correlate *uncooperative* (i.e., noisy) signals, which are not prevalent in our target applications. Our work differs from these works in that: i) none of the work considers I/O optimization ii) none of the work considers bounded approximation of correlation (e.g., HierarchyScan may output false negatives, sketch may output both false positives and negatives).

Our use of DFT is in the similar spirit of reducing dimensionality of signals. Previous works have used similar ideas to achieve tight lower bounds for pruning signals (e.g., to answer similarity queries [22, 23]). Some existing dimensionality reduction techniques (e.g., APCA [11], PAA [10], MSM [15]) provide better lower bounds for pruning than DFT and DWT. However, none of these techniques allows computing correlation (with bounded error) in the reduced dimensionality space. More specifically, even though these techniques reduce dimensionality to prune uncorrelated (or dissimilar) signals, correlation coefficients (or some other similarity metrics) of signals are computed in the time domain; in contrast we use DFT to compute approximate correlation in the frequency domain.

Many other prior works consider similarity or related queries in streaming scenario [7, 13, 14, 26]. These techniques are not adequate for our target applications since the techniques are not designed for stream warehouse settings, and/or do not explicitly consider correlation coefficients, and/or are not shown to scale to tens of thousands of streams.

3. MOTIVATION

We use a real dataset to demonstrate I/O and CPU complexities of computing a large correlation matrix C in a stream warehouse environment. The dataset, called *DataCenter1*, records a performance counter from a Microsoft data center (more details of the dataset is in Section 7). It consists of $n = 10,752$ sequences, each of length $m = 2880$. Each sequence is stored on disk as a separate file. [1] We use a 2.67 GHz quad core machine with 6GB RAM and a 750 GB Hitachi hard disk of 7200 rpm for this experiment.

Our experiments show that computing a correlation matrix for the above dataset is very expensive, especially with limited or no caching (Table 1). In the worst case, *without any caching*, a signal needs to be read from the disk $(n-1)$ times, to correlate with all $(n-1)$ other signals. In such a case, computing a correlation matrix for our above dataset takes approximately 129 hours! Obviously, most of the time is spent in reading signals from the disk. Our experiments show an average CPU utilization of only 2% without any cache. On the other hand, if the entire dataset can be cached in DRAM, each signal needs to be read from disk only once and hence the I/O cost is minimized. In such case, the correlation matrix can be computed in around 1 hour 37 minutes, highlighting the fact that more than 127 hours is spent for I/O in the no-cache scenario.

[1]Our conclusions hold if all data is stored in a single file.

Cache size, # signals		CPU time (Mins)	I/O time (Mins)
n	(Full cache)	93	4
$\frac{n}{32}$	(Partial cache)	93	40
2	(No cache)	93	7639

Table 1: Total time for computing a correlation matrix of $n = 10,752$ signals for various cache sizes.

However, in practice, it may not be possible to completely cache a large dataset. For example, the $DataCenter1$ dataset for 20,000 signals for a month is more than 24GB, which is significantly bigger than the available memory. Thus, only a fraction of the signals can be cached in the DRAM at a time. In such a case, a single signal may need to be read from the disk multiple times to correlate with all other signals. For example, a signal $\mathbf{x}$ may be evicted from the limited cache before another signal $\mathbf{y}$ is read from the disk; then $\mathbf{x}$ must be reread from disk later to compute $corr(\mathbf{x}, \mathbf{y})$.

An Optimal Baseline Caching Algorithm. Consider the following optimal baseline caching algorithm for dealing with a limited cache. We define the cache size as the number of signals it can hold at a time. Given n signals and a cache of size $(n/q + 1)$, signals are partitioned into batches $\{B_i\}$ of size n/q each, except of the last batch which can be smaller. Batches are determined according to signal IDs; i.e., the first n/q signals are in the first batch B_1, the second n/q signals are in the second batch B_2, and so on. Each batch B_i is brought to the cache at once, and correlation coefficients of all pairs of signals $(\mathbf{x}, \mathbf{y}), \mathbf{x} \in \mathbf{B_i}, \mathbf{y} \in \mathbf{B_i}$ are computed. Before the batch B_i is evicted from the cache, all remaining signals $\mathbf{z} \in B_{j>i}$ are read from the disk one at a time. When such a signal $\mathbf{z}$ is read, correlation coefficients of all pairs $(\mathbf{x}, \mathbf{z}), \mathbf{x} \in B_i$ are computed. After that, the next batch of signals is loaded into the cache and the process continues. This simple caching strategy is optimal because every time a signal is read from the disk, the number of correlation coefficients computed with it is exactly n/q, which is the maximum possible with a cache of size $(n/q + 1)$.

We use the above baseline strategy in our experiments with limited cache. As shown in Table 1, even if we have a cache large enough to hold $1/32$ of the entire dataset (a *partial cache* scenario), I/O cost remains significant (40 minutes). A careful back of the envelope calculation shows that the I/O cost to compute an all-pair correlation matrix of n signals with a cache of size $n/q + 1$ is proportional to $n(1 + q)/2$. Thus, the I/O cost linearly decreases with increasing cache size. We empirically validate this in Section 7.

The above empirical results highlight two main components of the total execution time.

- **High I/O cost:** With a limited cache, a significant amount of the time is spent for reading data from disk.
- **High CPU cost:** Even if there is enough cache to hold the entire dataset, computation remains expensive, as shown by the Full cache scenario in Table 1.

We next present techniques to reduce these two costs. For simplicity, we first assume only positive correlation and synchronized signals in the next two sections. We consider anti-correlation and lagged correlation in Section 6. Table 2 summarizes the symbols we use.

Symbol	Definition
n	Number of signals
m	Length of each signal
$\mathbf{x}, \mathbf{X}$	A signal and its DFT
$\widehat{\mathbf{x}}, \widehat{\mathbf{X}}$	Normalization of $\mathbf{x}$ and DFT of $\widehat{\mathbf{x}}$
$d(\mathbf{x}, \mathbf{y})$	Euclidean distance of $\mathbf{x}$ and $\mathbf{y}$
$d_k(\mathbf{x}, \mathbf{y})$	Euclidean distance of first k elements of $\mathbf{x}$ and $\mathbf{y}$
T	Correlation threshold

Table 2: Symbols and definitions

4. REDUCING I/O COSTS

In this section, we present a novel technique to reduce the total I/O costs required to answer a threshold correlation matrix query. As a shorthand, we call two signals *correlated* if their correlation coefficient is above the given threshold, or *uncorrelated* otherwise. Thus we need to compute correlation coefficients for correlated signal pairs only. A naïve algorithm would require computing correlation of all pairs of n signals and have an $O(n^2)$ I/O cost. A hypothetical optimal algorithm can reduce the cost in at least two ways. The first technique is *pruning*. If the algorithm *magically* knew which pairs of signals are correlated, it could compute correlation coefficients (and read relevant data from disk) for those pairs only and ignore uncorrelated pairs. The second technique is *intelligent caching*. The algorithm can read signals from the disk in an optimal order such that signals that are mutually correlated with each other reside in the cache at the same time—thus, a cached signal can be compared with a large number of other cached signals, reducing the amortized I/O cost of reading a signal from the disk.

Realizing such an algorithm needs answering two questions. First, *how does the algorithm know which signals are correlated?* This must be done with an I/O cost significantly smaller than the $O(n^2)$—pruning becomes useless if it itself has an I/O cost close to the $O(n^2)$ I/O cost of a naïve algorithm. Without any background knowledge about the nature of input signals, the algorithm must examine all the signals at least once (i.e., read each signal from disk at least once), implying an $O(n)$ lower bound of I/O cost.

Even after all correlated pairs are identified, I/O cost to compute correlation coefficients of correlated pairs can still be significantly high if a good *caching strategy* is not used. We define a caching strategy as the order in which signals are read to and evicted from the cache. The impact of different caching strategies on I/O cost is illustrated by an example in Figure 2. In Figure 2(a), a black cell (i, j) implies that signal i and signal j are correlated (e..g, their correlation coefficient is above a given threshold), and hence we need to compute their exact correlation coefficient. Knowing this information (e.g., from an oracle), an algorithm can read the signals from disk in many different orders, including the two strategies shown in Figure 2(b) and (c). Each step of a strategy shows the signals that are read from the disk (1st column), the cache content after the signals are read (2nd column), and the pairs of signals whose correlation coefficients have been computed at this step. We assume the cache can hold at most 4 signals at a time. As shown, both strategies compute the correlations of the same set of signal pairs; but Strategy 2 does that with almost half the I/O costs of Strategy 1.[2] The example illustrates the importance of choosing a good caching strategy. This leads us to the second challenge: *how can one find a good caching strategy to minimize I/O costs?*

We next address these two challenges.

[2]Note that correlation is symmetric; i.e., computing $corr(\mathbf{x}, \mathbf{y})$ gives $corr(\mathbf{y}, \mathbf{x})$.

Algorithm 1 $PruneUncorrelated(S, k)$

```
Require: A set $S$ of $n$ signals, with each signal $s_i \in S$ is of length $m$
Ensure: Report likely correlated signal pairs
1: for each signal $s_i \in S, 0 \leq i < n$ do
2:     Read $s_i$ from disk
3:     Normalize $s_i$ to $\widehat{s}_i$
4:     $DFT[i] \leftarrow$ first $k$ DFT coefficients of $\widehat{s}_i$
5: for each signal $s_i \in S, 0 \leq i < n$ do
6:     for each signal $s_j \in S, i < j < n$ do
7:         if $d_k(DFT[i], DFT[j]) \leq \sqrt{2m(1-T)}$ then
8:             Report the pair $(i, j)$ as likely correlated
```

4.1 Identifying Correlated Pairs

We use Discrete Fourier Transform (DFT) to identify correlated signal pairs in an I/O efficient manner. The DFT of a signal $\mathbf{x} = x_0, x_1, \ldots, x_{m-1}$ is a sequence $\mathbf{X} = X_0, X_1, \ldots, X_{m-1} = DFT(\mathbf{x})$ of complex numbers given by

$$X_f = \frac{1}{m} \sum_{k=0}^{m-1} x_i e^{\frac{-2\pi i f}{m} k}, f = 0, 1, \ldots, m-1$$

We also define the normalization of $\mathbf{x}$ as $\widehat{\mathbf{x}} = \widehat{x}_0, \widehat{x}_1, \ldots, \widehat{x}_{m-1}$, such that $\widehat{x}_k = (x_i - \mu_x)/\sigma_x$, where μ_x and σ_x are mean and standard deviation of the values $x_0, x_1, \ldots, x_{m-1}$.

As the following lemma suggests, the correlation coefficient of signals can be reduced to the Euclidean distance between their normalized series.

LEMMA 1 ([18]). *The correlation coefficient of two signals $\mathbf{x}$ and $\mathbf{y}$ is $corr(\mathbf{x}, \mathbf{y}) = 1 - \frac{1}{2m} d^2(\widehat{\mathbf{x}}, \widehat{\mathbf{y}})$, where $d(\widehat{\mathbf{x}}, \widehat{\mathbf{y}})$ is the Euclidean distance between $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{y}}$.*

By reducing the correlation coefficient to Euclidean distance, we can apply the techniques in [27] to report signals with correlation coefficients higher than a specific threshold.

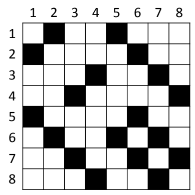
LEMMA 2 ([27]). *Let DFT of the normalized forms of two signals $\mathbf{x}$ and $\mathbf{y}$ be $\widehat{\mathbf{X}}$ and $\widehat{\mathbf{Y}}$. Then,*

$$corr(\mathbf{x}, \mathbf{y}) \geq T \Rightarrow d_k(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) \leq \sqrt{2m(1-T)}$$

where $d_k(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}})$ is the Euclidean distance between sequences $\widehat{X}_0, \widehat{X}_1, \ldots, \widehat{X}_{k-1}$ and $\widehat{Y}_0, \widehat{Y}_1, \ldots, \widehat{Y}_{k-1}$ for some $k \leq \frac{m}{2}$.

Lemma 2 implies that we can safely ignore signal pairs for which $d_k(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) > \sqrt{2m(1-T)}$, since they cannot have correlation coefficients above a given threshold T. By ignoring such pairs, we will get a set of *likely correlated* signal pairs. This is a superset of the correlated signal pairs, but there will be no false negatives. Similar technique has been used in previous works[1, 27]. For a large class of real-world signals (called *cooperative signals* [2]), including our data center data and stock prices, the first few low frequency DFT coefficients are sufficient to capture the overall shape of a signal. For such signals, computing only a small number of low frequency coefficients, e.g., using $k = 5$, is sufficient for identifying likely correlated signal pairs. The number of false positives can be reduced by using a larger k, which comes at the cost of increased computational overhead.

The above properties of DFT lay the foundation of our I/O efficient detection of correlated pairs. Algorithm 1 shows the details. Given n signals of length m on disk, we read one signal at a time to compute first k DFT coefficients of each signal, resulting in an



(a) Pruning Matrix

Signals read	In Cache	Computed pairs
$1, 2, 3, 4$	$1, 2, 3, 4$	$(1, 2), (3, 4)$
$5, 6, 7, 8$	$5, 6, 7, 8$	$(5, 6), (6, 7), (7, 8)$
$1, 2$	$1, 2, 5, 6$	$(1, 5), (2, 6)$
$3, 4, 7, 8$	$3, 4, 7, 8$	$(3, 7), (4, 8)$

(b) Caching Strategy 1, 14 disk reads

Signals read	In Cache	Computed pairs
$1, 2, 5, 6$	$1, 2, 5, 6$	$(1, 2), (1, 5), (2, 6), (5, 6)$
7	$2, 5, 6, 7$	$(6, 7)$
$3, 4, 8$	$3, 4, 7, 8$	$(3, 4), (3, 7), (4, 8), (7, 8)$

(c) Caching Strategy 2, 8 disk reads

Figure 2: Computing a threshold correlation matrix with two different caching strategies. The cache can hold 4 signals at a time. Strategy 2 is $1.75\times$ more I/O efficient than Strategy 1.

$O(n)$ total I/O cost. Since $k \ll m$, we can maintain the coefficients in cache. Then, we examine DFT coefficients of all pairs and identify the pairs of signals that are likely to be correlated (using Lemma 2). Conceptually, the algorithm produces a matrix like the one shown in Figure 2(a), where all pairs with correlation above a threshold and some pairs with correlation below the threshold (i.e., false positives) are marked as 1, and all other pairs are marked as 0. We call this a *Pruning Matrix* P and use it in subsequent steps.

4.2 Caching Strategy

A caching strategy involves deciding which set of signals to bring into the cache together and how to evict them from the cache. We use the same general framework we used for the optimal baseline algorithm in Section 3: we divide signals into batches and bring them into the cache one batch at a time. However, we introduce two optimizations in the baseline algorithm. First, before a batch is evicted from the cache, the baseline algorithm brings *all* signals in remaining batches, one at a time, to compute correlation coefficients of all signal pairs having exactly one signal in the currently cached batch. In contrast, we use the Pruning Matrix to ignore the uncorrelated pairs; thus we bring a signal in the cache only if it is likely correlated with at least one signal in the current batch. In the best case, if a batch is not correlated with any other signals, no additional signals need to read before eviction of the batch.

Our second, and the most important, caching optimization carefully chooses the batches. Note that, for each likely correlated signal pair whose two signals are in two different batches, we need to incur an additional disk read. Suppose the Pruning Matrix suggests that signals $\mathbf{x}$ and $\mathbf{y}$ are likely correlated and hence we need to compute $corr(\mathbf{x}, \mathbf{y})$. If they are put in the same batch, they will be read to the cache together and hence $corr(\mathbf{x}, \mathbf{y})$ can be computed without additional disk I/O. In contrast, if they are put in different batches, and if the batch containing $\mathbf{x}$ is read to cache before the batch containing $\mathbf{y}$, $\mathbf{y}$ will be read from disk at least twice—once just before the batch containing $\mathbf{x}$ is evicted, to compute $corr(\mathbf{x}, \mathbf{y})$, and again when the batch containing $\mathbf{y}$ is read to the cache. Thus, computing correlation between signals in different batches incurs additional I/O costs, and therefore we aim to *partition the signals into batches such that such additional I/O cost is minimized*. In Figure 2, caching strategy 1 uses two batches as $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8\}$, which results in four likely correlated signal pairs across batches. In contrast, caching strategy 2, which outperforms caching strategy 1, uses two batches as $\{1, 2, 5, 6\}$ and $\{3, 4, 7, 8\}$, resulting in only one such pair $(6, 7)$ across different batches.

Optimal data partitioning. Fortunately, we can formulate the above optimization problem as the *node capacitated graph partitioning problem* [5]. Given a graph $G = (V, E)$, edge weights w_e for $e \in E$, and a capacity B, the goal is to find a partition $(V_1, V_2, \ldots, V_\phi)$ of V such that $|V_i| < B$ for $1 \leq i \leq \phi$ and such

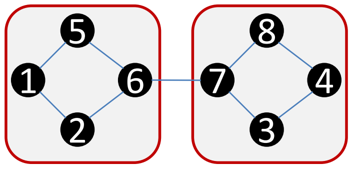


Figure 3: Partitioning signals into two batches to minimize the multicut size

that $\sum_{e \in \Delta} w_e$ is minimized, where Δ is the set of edges whose end nodes belong to different elements of the partition, typically called a multicut. The resulting partitioning is called minimum multicut size partitioning, or min-cut partitioning in short.

In our setting, the cache size B defines the capacity, the graph has the set of signals as the nodes, and the weight w_e of the edge e between node i and node j is $P[i, j]$, where P is the Pruning Matrix. Thus, different elements of the resulting partition denote different batches of signals that are read to the cache together. Figure 3 shows an example graph (only edges with weight 1 are shown) representing the Pruning Matrix in Figure 2(a), and two batches of signals resulting in the caching strategy 2 in Figure 2(c). Intuitively, we try to avoid pairs of signals that are likely correlated with each other (as indicated by the Pruning Matrix) to place in different batches.

The above graph partitioning problem is NP-complete [5]. There are many heuristics-based and approximation algorithms for balanced graph partitioning [3, 6, 8, 25]. Many of the algorithms are used in offline circuit partitioning in VLSI design, and hence they optimize for accuracy at the cost of increased execution time. In contrast, we need to partition graph online, during query execution. Hence, we have chosen the simplest and the fastest of these existing algorithms: the F-M (Fiduccia-Mattheyses) algorithm [6]. F-M is a bi-partitioning algorithm that partitions a given graph into two equal size partitions while minimizing the size of the multicut. We use it recursively to get a multi-way balanced partitioning such that each partition is smaller than or equal to the cache size B and the multicut size is minimized. (Such a recursive approach has been shown to yield smaller multicut size than iterative approaches [24].)

Since graph bi-partitioning is NP-Hard, the F-M algorithm uses heuristics for bi-partitioning. It starts with a random balanced bi-partitioning and iteratively reduces the multicut size. It defines the *gain* of a vertex as the difference between the number of its adjacent vertices in its opposite partition and the number of its adjacent vertices in its current partition. In each iteration, the algorithm considers each vertex in the descending order of gains and tentatively moves it to the opposite partition. After a vertex is moved to an opposite partition, the gains of all its adjacent vertices are updated.

Finally, the algorithm finalizes the first k moves such that the total gain of the first k vertices is maximized and the resulting partitions are balanced. The algorithm stops whenever an iteration cannot improve the current bi-partitioining. The algorithm is shown to converge in very few iterations (< 10) [6].

For multi-way partitioning, we recursively use the F-M algorithm to partition an input graph with n vertices into M partitions such that the size of each partition is $\leq B$. Ideally, we should continue recursive partitioning until we get $M' = \lceil n/B \rceil$ partitions. However, such a restriction does not provide the partitioning algorithm enough flexibility to find good partitions. So, we continue partitioning until we get $M > M'$, say $M = 2\lceil n/B \rceil$, partitions. This results in good partitions, along with a few partitions significantly smaller than B. At the end, we merge those small partitions together to produce larger partitions of size $\leq B$. The final partitions determine the batches of signals in our caching strategy mentioned before.

5. REDUCING COMPUTATIONAL COST

The optimizations presented so far help reducing the I/O overhead of answering a threshold correlation matrix query. However, as mentioned in Section 3, computing the matrix remains expensive even if we completely ignore the I/O cost. In this section we present techniques to reduce this computational complexity.

We exploit the fact that real-world applications, including our running example of data center management application, can often tolerate some small, bounded approximation errors in computing correlation. Specifically,

- It is often sufficient to report correlation coefficients within a small error bound. For example, an application looking for all signal pairs with correlation > 0.9 may tolerate an error of ± 0.02 in the computed correlation coefficients.
- It may often be sufficient to only identify all and only the pairs with correlation above a given threshold; knowing the exact correlation coefficients is not strictly necessary. Note that the *Pruning Matrix* computed in the previous section is not sufficient for this purpose since it contains false positives.

In this section, we present two algorithms that can compute approximate threshold correlation matrices satisfying the above requirements, but run significantly faster than a corresponding exact algorithm. Figure 4 shows an example of these approximate matrices, highlighting the fact that they are almost as useful as the exact matrix in Figure 1(a) in identifying correlation patterns.

Note that even in situations where exact correlations may be required, our approximate algorithms can still be useful for quickly identifying the existence of interesting correlation patterns in the data. The algorithms can give users the flexibility to avoid expensive exact correlation computation if no interesting patterns are found.

5.1 Approximate Threshold Correlation Matrix

In this section, we provide solutions for efficiently computing ϵ-approximate correlation of signals. An ϵ-approximate correlation coefficient $corr^{\epsilon}(i,j)$ of two signals i and j is a value within ϵ of their exact correlation coefficient $corr(i,j)$; i.e., $corr(i,j) - \epsilon \leq corr^{\epsilon}(i,j) \leq corr(i,j) + \epsilon$. Thus, we consider the following problem.

PROBLEM 2 (APPROX. THRESHOLD CORR. MATRIX). *Given n sequences of equal length m, a threshold T, and an error bound ϵ, compute an $n \times n$ approximate threshold correlation matrix matrix C^{ϵ} such that $C^{\epsilon}[i,j] = corr^{\epsilon}(i,j)$ if $corr^{\epsilon}(i,j) > T$, and $C^{\epsilon}[i,j] = 0$ otherwise.*

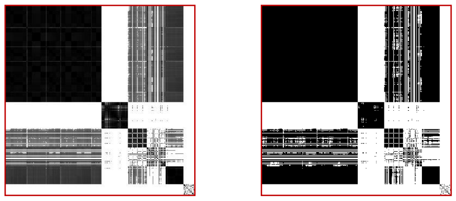

(a) C^{ϵ} ($\epsilon = 0.04, T = 0.5$) (b) $C^{B}(T = 0.5)$

Figure 4: (a) An approximate threshold corr. matrix C^{ϵ}, and (b) a Boolean threshold corr. matrix C^{B}, corresponding to the matrix C in Figure 1(a).

Figure 4(a) shows an approximate threshold correlation matrix C^{ϵ} corresponding to the exact threshold correlation matrix C^{T} in Figure 1(b). If we zoom in, we would notice that a few white pixels in C^{T} are shown as gray in C^{ϵ} (i.e., a few pairs with zero correlation in C^{T} are shown to have non-zero correlation in C^{ϵ}). This happens because, due to approximation, a few (not all) pairs with correlation coefficients within the range $[T - \epsilon, T)$ are reported to have a correlation $\geq T$. However, as shown in the figure, such occurrences are rare. Moreover, such approximation is acceptable since the threshold is not a hard one in most applications. Also note that such approximation does not introduce any false negatives—no pairs with correlation above the threshold will be omitted in C^{ϵ}.

5.1.1 Approximation with Prefix Distance

According to Lemma 1, correlation of two signals $corr(\mathbf{x}, \mathbf{y})$ can be computed by their Euclidian distance $d(\hat{\mathbf{x}}, \hat{\mathbf{y}})$. Thus, one way to approximate $corr(\mathbf{x}, \mathbf{y})$ is to use an approximate value for $d(\hat{\mathbf{x}}, \hat{\mathbf{y}})$; e.g., $d_k(\hat{\mathbf{x}}, \hat{\mathbf{y}})$, for $k < m$. For significant savings in computational complexity, it is important that $k \ll m$. Unfortunately such approximation does not work well in the time domain—for $k \ll m$, $d_k(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ is typically not a good approximation for $d(\hat{\mathbf{x}}, \hat{\mathbf{y}})$.

We therefore approximate distance in the frequency domain. Since DFT is a linear transformation, the Euclidean distance is preserved under the transformation. Therefore, $\frac{1}{m} d^2(\hat{\mathbf{x}}, \hat{\mathbf{y}}) = d^2(\hat{\mathbf{X}}, \hat{\mathbf{Y}})$. This allows us to rephrase Lemma 1 as follows:

$$corr(\mathbf{x}, \mathbf{y}) = 1 - \frac{1}{2} d^2(\hat{\mathbf{X}}, \hat{\mathbf{Y}})$$

Thus, approximate correlation can be computed by using an approximate distance $d(\hat{\mathbf{X}}, \hat{\mathbf{Y}})$, such as $d_k(\hat{\mathbf{X}}, \hat{\mathbf{Y}})$ for $k < m$. Since for many real-world datasets, first few DFT coefficients (i.e., a small prefix of the DFT coefficient vector) capture most information of the original sequences, $d_k(\hat{\mathbf{X}}, \hat{\mathbf{Y}})$ is a good approximation of $d(\hat{\mathbf{X}}, \hat{\mathbf{Y}})$ for $k \ll m$. Thus, unlike in time domain, approximating distance d with d_k is computationally attractive in the frequency domain.

However, computing each DFT coefficient requires $O(n)$ computation. Thus, to minimize overall computation cost while guaranteeing a given approximation error bound, we need to compute the smallest number k of DFT coefficients that ensure the target accuracy. However, the value of k depends on datasets. For some datasets (e.g., periodic or random-walk signals), a small k may be sufficient to ensure a good approximation; while a very large k may be required for some other signals (e.g., white noise signal).

Our main result in this section shows a relationship between the approximation error in a correlation coefficient and the number of leading DFT coefficients used for approximating the coefficient. It uses the notion of *energy* of a signal. The energy of a signal $\mathbf{x}$ is defined as $E(\mathbf{x}) = \parallel \mathbf{x} \parallel = \sum_{i=0}^{m-1} x_i^2$. We also define the energy captured by the first k components of the signal $\mathbf{x}$ as $E_k(\mathbf{x}) = \sum_{i=0}^{k-1} x_i^2$. The following lemma shows that normalization bounds the total energy of a signal.

LEMMA 3. *Let $\widehat{\mathbf{X}}$ be the DFT of normalized signal $\widehat{\mathbf{x}}$, then*

$$\begin{aligned} E(\widehat{\mathbf{X}}) &= \sum_{i=0}^{m-1} |\widehat{X}_i|^2 = 1 \\ E_k(\widehat{\mathbf{X}}) &\leq 1, \textit{for } k \leq m \end{aligned}$$

In the following lemma, we show an upper bound of approximation errors if we use the first k DFT coefficients to compute distance of two signals.

LEMMA 4. *Let $\widehat{\mathbf{X}}$ and $\widehat{\mathbf{Y}}$ be the DFTs of normalized signals $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{y}}$ and $\eta = \min(2\sum_{i=0}^{k} |\widehat{X}_i|^2, 2\sum_{i=0}^{k} |\widehat{Y}_i|^2)$ for some $k \leq \frac{m}{2}$. Then*

$$2d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) \leq d^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) \leq 2d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) + 4(1 - \eta)$$

PROOF. The first inequality is obvious. On the second inequality, using the symmetry of $\widehat{\mathbf{X}}$, we get

$$\begin{aligned} d^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) &= \sum_{i=0}^{m-1} \left|\widehat{X}_i - \widehat{Y}_i\right|^2 \\ &\leq 2d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) + \sum_{i=k+1}^{m-k-1} (|\widehat{X}_i|^2 + |\widehat{Y}_i|^2 + 2|\widehat{X}_i\widehat{Y}_i|) \end{aligned}$$

Now, using Cauchy and Schwarz inequality, we get

$$\leq 2d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) + \sum_{i=k+1}^{m-k-1} (|\widehat{X}_i|^2 + |\widehat{Y}_i|^2) + 2\sqrt{\sum_{i=k+1}^{m-k-1} |\widehat{X}_i|^2 \sum_{i=k+1}^{m-k-1} |\widehat{Y}_i|^2}$$

Now, using Lemma 3, we get

$$\begin{aligned} &\leq 2d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) + 1 - \eta + 1 - \eta + 2\sqrt{(1-\eta)(1-\eta)} \\ &= 2d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) + 4(1 - \eta) \end{aligned}$$

□

The above lemma leads to our main result.

THEOREM 1. *Given two signals $\mathbf{x}$ and $\mathbf{y}$ and an error bound ϵ,*

$$corr(\mathbf{x}, \mathbf{y}) - \epsilon \leq 1 - d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}}) \leq corr(\mathbf{x}, \mathbf{y}) + \epsilon$$

where the value of k is chosen such that

$$\min(2\sum_{i=0}^{k} |\widehat{X}_i|^2, 2\sum_{i=0}^{k} |\widehat{Y}_i|^2) \geq 1 - \frac{\epsilon}{2}$$

The theorem enables us to incrementally compute the smallest number of DFT coefficients that are sufficient to guarantee the approximation error to be $\leq \epsilon$. For example, if $\epsilon = 0.04$, we only need to compute as many DFT coefficients that capture 0.98 of the normalized energy. Many real-world data sets are *cooperative* [2]; i.e., most of their energy is concentrated in the lower frequencies and the energy decreases quickly with higher frequencies. For example, in a set of random walk signals (which model, e.g., stock prices) of length 1000, 98% of the energy is captured by the first 60 coefficients for more than 90% of the signals. Thus, approximate correlation within an error of 0.04 can be computed by using only 60 numbers, instead of by using the entire signals that consist of 1000 numbers each.

Also note that the value of $d_k^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}})$ monotonically increases with increasing k. As soon as $d_l^2(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}})$, for $l < k$, becomes larger than a threshold implying $corr(\mathbf{x}, \mathbf{y}) < T$, we can abandon the distance computation as well as computing the rest of the k DFT coefficients for both signals.

5.1.2 Computational Cost

The above optimization comes with the additional cost of actually computing the DFT coefficients, as an algorithm $\mathcal{A}$ for computing exact correlation in the time domain does not need expensive DFT operations. However, many systems [20] compute and archive DFT coefficients along with or in place of raw signals for data compression and efficient data mining; we can simply leverage these already computed DFT coefficients. Even if the DFT coefficients are computed on the fly, our overall query processing cost can still be smaller than $\mathcal{A}$ for two reasons. First, we need to compute only a small number of DFT coefficients. Second, DFT coefficients need to be computed only once per signal and can be reused for all pairs involving the signal. A simple back of the envelope calculation shows the savings. Consider a set of n signals, each of length m. Computing all pairwise exact correlations in the time domain has a computational complexity of $cost_1 = \binom{n}{2}m$. In contrast, our approach has the complexity of $cost_2 = kmn + \binom{n}{2}k$, where the first component is for computing k DFT coefficients of all n signals and the second component is for computing pairwise correlations. Assuming $k \ll m$, we get $cost_2 < cost_1$ for $n > 2k$. Thus, as long as we deal with $> 2k$ signals, the total cost of computing DFT coefficients and pairwise correlations remains less than computing exact correlations in the time domain. The benefit becomes more pronounced with a large number of signals. Our experimental results in Section 7 include DFT computation overhead and still show significant speedup.

5.2 Threshold Boolean Correlation Matrix

The second type of approximation we consider is to identify if a pair of signals has correlation above a given threshold T. More precisely, we consider the following problem.

PROBLEM 3 (THRESHOLD BOOLEAN CORR. MATRIX). *Given n sequences of equal length m and a threshold T, compute a threshold Boolean correlation matrix C^B such that,*

$$C^B[i,j] = \begin{cases} 0 & \textit{if } corr(i,j) < T \\ 1 & \textit{if } corr(i,j) \geq T \end{cases}$$

Figure 4(b) shows the C^B for the threshold correlation matrix C^T in Figure 1(b). As shown, C^B still preserves the darker regions in C and captures valuable information such as clusters of servers showing similar behavior.

Suppose, given two signals $\mathbf{x}$ and $\mathbf{y}$, we somehow know, without actually computing the exact distance $d(\mathbf{x}, \mathbf{y})$, the upper bound $UB(\mathbf{x}, \mathbf{y})$ and the lower bound $LB(\mathbf{x}, \mathbf{y})$ of their Euclidean distance. Then, using these bounds with Lemma 5, we may be able to conclude if $\mathbf{x}$ and $\mathbf{y}$ are correlated or not. More specifically,

LEMMA 5. *Let $\mathbf{x}$ and $\mathbf{y}$ are the two signals, then*

$$C^B[\mathbf{x}, \mathbf{y}] = \begin{cases} 1 & \textit{if } UB(\mathbf{x}, \mathbf{y}) \leq \theta \\ 0 & \textit{if } LB(\mathbf{x}, \mathbf{y}) > \theta \\ \textit{Undetermined} & \textit{Otherwise} \end{cases}$$

Algorithm 2 $BooleanApproximation(S, T)$

```
Require: A set $S$ of $n$ signals, with each signal $s_i \in S$ is of length $m$, and a threshold $T$
Ensure: Report a 0/1 matrix ; 1 for correlated signal pairs and 0 for uncorrelated pairs
1: $\theta = \sqrt{2m(T-1)}$
2: Initialize temporary matrices $UB$ and $LB$ with $\infty$ and 0, respectively
3: for $0 \leq i < n-1$ do
4:     $UB[i, i+1] = LB[i, i+1] = d(s_i, s_{i+1})$
5:     $C^B[i, i+1] = 1$ if $d(s_i, s_{i+1}) \leq \theta$, 0 otherwise
6: for each diagonal $k, 1 < k < n$ do
7:     for each cell $(i, j), 0 \leq i < n-k, j = i+k$ do
8:         $UB[i, j] = \min_{i<v<j}\{UB[i, v] + UB[v, j]\}$
9:         $LB[i, j] = \max_{i<v<j}\{\max\{LB[i, v] - UB[v, j], LB[j, v] - UB[v, i]\}$
10:        if $UB[i, j] \leq \theta$ then
11:            $C^B[i, j] = 1$
12:        else if $LB[i, j] > \theta$ then
13:            $C^B[i, j] = 0$
14:        else
15:            $UB[i, j] = LB[i, j] = d(s_i, s_j)$
16:            $C^B[i, j] = 1$ if $d(s_i, s_j) \leq \theta$, 0 otherwise
```

where $\theta = \sqrt{2m(T-1)}$ *and* $UB(\mathbf{x}, \mathbf{y}) \geq d(\mathbf{x}, \mathbf{y})$ *and* $LB(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{y})$.

The proof of the lemma follows Lemma 2. Obviously, the tighter the upper and the lower bounds are, the more likely it is to determine the correlation status of a pair.

The key question is *how do we efficiently find good bounds on distances of two signals?* One efficient way to compute the lower bound is to compute $d_k(\widehat{\mathbf{X}}, \widehat{\mathbf{Y}})$. While using a small k is computationally cheap, it may not provide a tight lower bound. Moreover, this does not provide any hints of the upper bound. We address this problem with the triangular inequality property of Euclidean distance. The following lemma shows how one can get the bounds using triangular inequality and a random reference point.[3]

LEMMA 6. *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *are two signals and* $\mathbf{r}$ *is a reference signal, then*

$$UB(\mathbf{x}, \mathbf{y}) = d(\mathbf{x}, \mathbf{r}) + d(\mathbf{y}, \mathbf{r}) \geq d(\mathbf{x}, \mathbf{y})$$

and

$$LB(\mathbf{x}, \mathbf{y}) = |d(\mathbf{x}, \mathbf{r}) - d(\mathbf{y}, \mathbf{r})| \leq d(\mathbf{x}, \mathbf{y})$$

One straightforward way to use the above property in computing C^B is to compute distances of all signals from a random reference signal and then use the distances to compute bounds for all $O(n^2)$ pairs. Unfortunately, apart from being computationally expensive, the resulting upper bounds are not tight and are practically useless because a random reference signal is expected to be equally far from all of the signals in a very high ($= m$) dimensional space.

To overcome this problem, we need to choose reference signals carefully. Ideally, for a pair of signal $\mathbf{x}$ and $\mathbf{y}$, we want a third signal $\mathbf{v}$ (called a *verifier*) that is either (i) very close to both $\mathbf{x}$ and $\mathbf{y}$ implying that $\mathbf{x}$ and $\mathbf{y}$ are likely to be close, and hence correlated to each other, or (ii) very close to $\mathbf{x}$ but not to $\mathbf{y}$ (or vice versa), implying that $\mathbf{x}$ and $\mathbf{y}$ are not correlated. Such verifiers provide tighter upper and lower bounds of distances between signals and are effective in labeling signals as correlated or uncorrelated without expensive distance computation.

The last piece of the puzzle is to efficiently find effective verifiers. Since we deal with a database of a large number of signals; many of which are correlated with each other; we can find such good verifier signals within the database itself. We now present a dynamic programming based algorithm to systematically search the input signals for finding verifiers for different signal pairs.

[3]We cannot use triangular inequality on correlation values because correlation is not a metric.

A Dynamic Programming Algorithm. Given a set of signals, the algorithm computes, for each signal pair, an upper and a lower bound of their distances. These bounds are then compared to a threshold (see Lemma 5) to label pairs as correlated or uncorrelated. For the pairs whose correlation status cannot be decided based on its upper and lower bounds, their true distances are computed to determine their correlation status.

The basic observation is that computing the lower and the upper bounds of the true distance between signals $\mathbf{s_i}$ and $\mathbf{s_j}$, $i < j$ can be decomposed into the following recursive substructure. This enables us to compute bounds for a pair of signals from the bounds already computed for other pairs.

$$UB(\mathbf{s_i}, \mathbf{s_j}) = \min_{i<v<j} \{UB(\mathbf{s_i}, \mathbf{s_v}) + UB(\mathbf{s_v}, \mathbf{s_j})\} \quad (1)$$

$$LB(\mathbf{s_i}, \mathbf{s_j}) = \max_{i<v<j} \{\max\{LB(\mathbf{s_i}, \mathbf{s_v}) - UB(\mathbf{s_v}, \mathbf{s_j}), LB(\mathbf{s_j}, \mathbf{s_v}) - UB(\mathbf{s_v}, \mathbf{s_i})\}\} \quad (2)$$

Algorithm 5.2 shows the pseudocode of our algorithm for computing C^B. It fills the upper-right half of the matrix C^B, one diagonal at a time.[4] First, it computes the true distances (and hence exact correlations) of the signal pairs on the principal diagonal of C^B (Line 4). The true distances are assigned as lower and upper bounds of corresponding signal pairs. These bounds are then reused for computing correlation of other pairs. Next, the algorithm considers subsequent diagonals, and for each signal pair on a diagonal, it uses Equations 1 and 2 to compute upper and lower bounds of their true distances (Line 8 and 9). Diagonals are considered in order, starting from the principal diagonal towards the top-right corner of C^B. This ensures that before a signal pair (i, j) is considered, bounds of all the signal pairs $(i, v), (v, j), i < v < j$ are already computed; hence, these already computed bounds and Equations 1 and 2 can be used for the pair (i, j). Finally, the algorithm uses Lemma 5 to decide a pair's correlation status based on its distance bounds (Line 11 and 13). If the status of a pair cannot be determined based on its distance bounds, the algorithm computes exact distance of the pair and decides its correlation status based on the true distance (Line 16).

Note that, examining all previously computed diagonals for verifiers can be expensive than computing the exact correlation when $n \gg m$. In conditions like this, we can use a shorter search range for the verifier. For example, to consider no more than 5 signals as verifiers, we can use $i < v < t$, $t = min(j, 5)$ instead of $i < v < j$ in line 8 and line 9.

6. EXTENSIONS

In this section we describe several useful extensions of our approximation algorithms described in Section 5.

6.1 Negative Correlation

In real datasets, signals often show negative correlation (or, anti-correlation). For example, number of bytes available in the main memory of a server is negatively correlated with the number of TCP connections made to that server. Both our approximation algorithms can be extended to handle such negative correlations. The basic idea is that $corr(\mathbf{x}, \mathbf{y}) = -corr(\mathbf{x}, -\mathbf{y})$. Thus, for a given negative threshold T, $corr(\mathbf{x}, \mathbf{y}) \leq T \Rightarrow corr(\mathbf{x}, -\mathbf{y}) \geq -T$.

[4]C^B is symmetric, so computing half of it is sufficient.

We can easily extend our approximate threshold correlation algorithm in Section 5.1 for deciding if a signal pair $(\mathbf{x}, \mathbf{y})$ has an anti-correlation smaller than a threshold $T < 0$, as follows. The modified algorithm compares $d_k(\widehat{\mathbf{X}}, -\widehat{\mathbf{Y}})$ with a threshold based on $-T$, where the appropriate value of k is determined by Theorem 1. Note that the modified algorithm does not need to compute any additional DFT coefficients; the same DFT coefficients computed for positive correlations are sufficient.

Similarly, our algorithm in Section 5.2 for threshold Boolean correlation query can be extended for negative correlation between signals $\mathbf{x}$ and $\mathbf{y}$ by computing lower and upper bounds of the distance $d(\mathbf{x}, -\mathbf{y})$.

The above extensions have the same asymptotic computational complexity as our original algorithms.

6.2 Lagged Correlation

So far we have considered *synchronous correlation* where signals to be correlated are assumed to be aligned with each other in the time dimension. However, signals in real datasets often exhibit correlation with unknown lag. For example, in a typical two-tiered web service deployment, an increase in the number of TCP connections in the front-end server typically precedes an increase in the CPU load in the back-end server. Intuitively, two signals have a lagged correlation with lag l if they look very similar when one signal is delayed by l time ticks. Formally, given two signals $\mathbf{x}$ and $\mathbf{y}$ of equal length m, their correlation with lag l is $corr_l(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{m-l-1} \frac{(x_i - \mu_x)}{\sigma_x} \frac{(y_{i+l} - \mu_y)}{\sigma_y}$, where $\mu_x, \mu_y, \sigma_x, \sigma_y$ are defined on the overlapping part of two signals. Note that synchronous correlation is a special form of lagged correlation with $l = 0$.

We further define the *maximum lagged correlation* of two signals as the maximum of their correlations with all possible lags. Note that in the above definition, only one signal has been lagged or shifted. If both signals are periodic, shifting any of the signals yields the same maximum lagged correlation. Otherwise, if any of the signals is aperiodic, both the signals need to be shifted to find the maximum lagged correlation. For simplicity of description, we here consider shifting only one signal.

We now show how to extend our previous algorithms to consider maximum lagged correlations, instead of synchronous correlation. Our approach is similar to BRAID [21], which discovers maximum lagged correlation of a signal pair in $O(\lg m)$ time, where m is the maximum possible lag. Instead of computing correlations for all possible lags to find the maximum, BRAID probes in geometric progression and interpolates the remaining values of the cross correlation function. Although BRAID is designed for streaming applications, it can easily be adapted to use in a stream warehousing scenario. However, BRAID computes correlations in the time domain, which can be significantly expensive for a large number of long signals. We now show how BRAID can be used in the frequency domain to avoid such cost.

Note that to compute lagged correlation of $\mathbf{x}$ and $\mathbf{y}$ with a lag l, one signal is first shifted (or, lagged) by l time ticks while keeping the other signal fixed, and then the correlation is computed over their trimmed, common parts of length $(m - l)$. Without loss of generality, assume that the common parts include a prefix of signal $\mathbf{x}$ and a suffix of signal $\mathbf{y}$, both of length $(m-l)$; i.e., x_0 is aligned with y_l to compute lagged correlation with a lag of l. To work in the frequency domain, a naïve solution would compute DFT of all prefixes of $\mathbf{x}$ and suffixes of $\mathbf{y}$. However, the following lemma shows that we can compute DFT coefficients of a signal once, and then reuse them to compute coefficients for any prefix or suffix of the signal.

LEMMA 7. *Let* $\mathbf{x}$ *be a signal of length* m *with DFT* $\mathbf{X}$. *Then, for* $r = 0, 1, \ldots, m - l - 1$
(i) [Prefix] The DFT of $\mathbf{x}^p = x_0, x_1, \ldots, x_{m-l-1}$ *is* $\dot{\mathbf{X}}$ *where*

$$\dot{X}_r = X_{\frac{mr}{m-l}} + \frac{1}{m-l}\Big[\sum_{p=0, p\neq \frac{mr}{m-l}}^{m-1} X_p \frac{e^{2\pi i(p - \frac{pl}{m} - r)} - 1}{e^{2\pi i(\frac{p}{m} - \frac{r}{m-l})} - 1}\Big]$$

(ii)[Suffix] The DFT of $\mathbf{x}^s = x_l, x_{l+1}, \ldots, x_{m-1}$ *is* $\ddot{\mathbf{X}}$ *where*

$$\ddot{X}_r = S_{\frac{mr}{m-l}} + \frac{1}{m-l}\Big[\sum_{p=0, p\neq \frac{mr}{m-l}}^{m-1} S_p \frac{e^{2\pi i(p - \frac{pl}{m} - r)} - 1}{e^{2\pi i(\frac{p}{m} - \frac{r}{m-l})} - 1}\Big]$$

where S_p *is the l-shift of* X_p *defined as* $S_p = e^{\frac{2\pi i}{m} lp} X_p$.

PROOF.

$$\begin{aligned}
\dot{X}_r &= \frac{1}{m-l}\sum_{j=0}^{m-l-1} x_j e^{-\frac{2\pi i r}{m-l} j} \\
&= \frac{1}{m-l}\sum_{j=0}^{m-l-1}\sum_{p=0}^{m-1} X_p e^{\frac{2\pi i j}{m} p} e^{-\frac{2\pi i r}{m-l} j} \\
&= \frac{1}{m-l}\sum_{p=0}^{m-1} X_p \sum_{j=0}^{m-l-1} e^{2\pi i(\frac{p}{m} - \frac{r}{m-l})j} \\
&= X_{\frac{mr}{m-l}} + \frac{1}{m-l}\Big[\sum_{p=0, p\neq \frac{mr}{m-l}}^{m-1} X_p \frac{e^{2\pi i(p - \frac{pl}{m} - r)} - 1}{e^{2\pi i(\frac{p}{m} - \frac{r}{m-l})} - 1}\Big]
\end{aligned}$$

The proof for suffix is similar. □

Thus, once the DFT coefficients for $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{y}}$ are computed, they can be reused to compute DFT coefficients for all their prefixes and suffixes and hence be used for computing correlations with arbitrary lags.

The above result enables us to efficiently compute lagged versions of C^{ϵ} and C^B in the frequency domain. In a lagged C^{ϵ}, an entry $C^{\epsilon}[i, j]$ is an ϵ-approximation of the maximum lagged correlation of signals i and j. Similarly, in a lagged C^B, an entry $C^B[i, j]$ is 1 iff the maximum lagged correlation of signals i and j is above the given threshold.

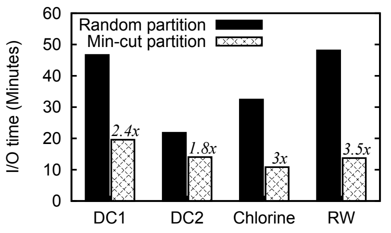
However, there is a caveat. The basic idea above requires all DFT coefficients of an original signal; in contrast, our approximation algorithms compute only a first few DFT coefficients. Thus, we need to approximate remaining coefficients with zeros, which introduces errors in the DFT coefficients we compute for prefixes and suffixes. In general, this may cause our algorithms to violate approximation guarantees. In practice, however, the effect is very small because our algorithms compute as many DFT coefficients as required to capture the most of the energy of a signal; hence, ignoring the remaining coefficients does not affect the accuracy much. We will experimentally validate this in Section 7.

7. EVALUATION

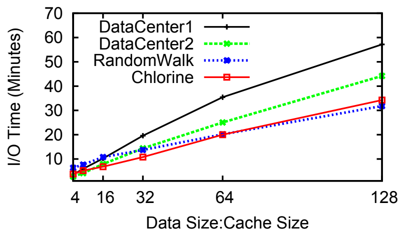
We evaluate our algorithms using the same machine described in section 3.

We use four datasets. To perform experiments on massive sized data, we replicate every signal in a dataset equal number of times with small additive white noise. This preserves the pairwise correlation structure in the original dataset.

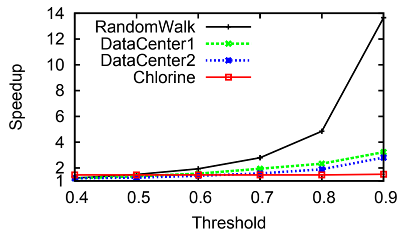
- *DataCenter1* contains measurements of the number of TCP connections established over a day to 350 servers in a real data center. One measurement is made every 30 seconds, and so a signal for a day consists of $m = 2,880$ samples. The dataset contains $n = 11,200$ signals.



(a) I/O cost reduction



(b) Impact of cache size



(c) CPU cost reduction

Figure 5: Impact of I/O and CPU optimizations for computing a threshold correlation matrix. In (a), DC1=$DataCenter1$, DC2=$DataCenter2$, and RW=$RandomWalk$.

- *DataCenter2* is a collection of measurements of the CPU utilization of 120 servers in a real data center. One measurement is made every 30 seconds, and a signal for a day consists of $m = 2,880$ samples. The dataset contains $n = 4,745$ signals.
- *Chlorine* [17] is a collection of signals representing chlorine concentration at different junctions in a water distribution network. The original dataset has 166 signals of two weeks long signal traces (one sample in every 5 minutes). We use week-long ($m = 2,155$ samples) traces and replicate them to create a dataset of $n = 10,624$ signals.
- *RandomWalk* is a collection of $n = 16,384$ random walk signals generated synthetically using Gaussian steps. Each signal is $m = 2,880$ samples long.

7.1 Impacts of I/O Optimizations

To show the benefit of our min-cut partitioning based caching strategy, we compare it with a baseline caching strategy where signals are randomly partitioned. The only difference between the two caching strategies is how they partition signals into batches—both are used for computing a C^T, and they both prune signal pairs based on the same Pruning Matrix. We assume the cache is big enough to hold $\frac{1}{32}$th of a dataset. The result is shown in Figure 5(a). As shown, our min-cut partitioning significantly (1.8× - 3.5×) reduces the I/O time for all the datasets (the factor of reduction in I/O time is shown at the top of the second bar). This reduction is attributed to our careful partitioning that reduces the number of disk I/O required to compute correlation of signal pairs across different batches. However, the overhead of partitioning is never more than 30 seconds, which is very tiny compared to the end-to-end response time.

The I/O cost of our caching strategy can be reduced by using a bigger cache. To show the impact of cache size, we vary the cache size keeping the datasize fixed. Figure 5(b) shows I/O costs as a function of the ratio of cache size and data size. As shown, the I/O time decreases linearly with the increase in available cache size, and becomes < 10 minutes for a cache of size $n/16$. This is due to the fact that with a larger cache, data is partitioned into fewer batches, and hence fewer disk I/Os are required to compute correlation of signal pairs across batches. The slopes of different lines demonstrate the amount of correlation present in different datasets. The more correlated pairs in a dataset are, the larger the slope is. The $RandomWalk$ dataset has the least slope among all datasets, as it has the least correlation among signals.

7.2 CPU Speedup due to Approximation

We now show how much our approximation algorithms reduce the CPU cost of computing a correlation matrix. As a shorthand, we use the following notations:

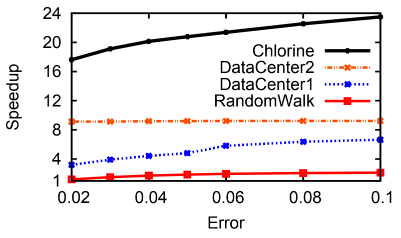
- $\mathcal{A}$: an algorithm to compute an all-pair exact correlation matrix C in the time domain,
- $\mathcal{A}_T$: an algorithm, described below, for computing an exact threshold correlation matrix C^T,
- $\mathcal{A}_\epsilon$: our algorithm for computing an approximate threshold correlation matrix C^ϵ (Section 5.1),
- $\mathcal{A}_B$: our algorithm for computing a threshold Boolean correlation matrix C^B (Section 5.2).

For different algorithms, we report the *speedup* factors. The speedup of an algorithm is the ratio of the end-to-end CPU time of a baseline algorithm to that of the algorithm. The higher the speedup, the faster the algorithm. For our approximation algorithms $\mathcal{A}_\epsilon$ and $\mathcal{A}_B$, we use $\mathcal{A}_T$ as the baseline. Before reporting the speedups of our algorithms, we first report the speedup factor of $\mathcal{A}_T$, with $\mathcal{A}$ as the baseline. This will allow us to interpret the speedup factors of $\mathcal{A}_\epsilon$ and $\mathcal{A}_B$ with respect to $\mathcal{A}$ as well.

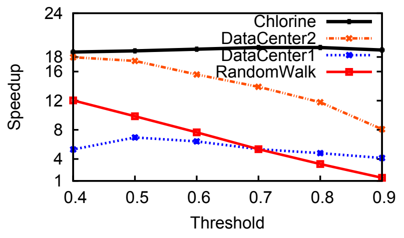
►**Threshold Correlation Matrix.** To compute C^T for a given threshold, $\mathcal{A}_T$ prunes uncorrelated signal pairs based on their distances of $k = 5$ first DFT-coefficients (similar to the methods in [1, 27]). The exact correlations of likely correlated signal pairs are then computed in the time domain. Figure 5(c) shows, for different correlation thresholds, the speedup factors of $\mathcal{A}_T$, with respect to $\mathcal{A}$ (which takes more than 90 minutes of CPU time for all the datasets). As shown, $\mathcal{A}_T$ can be several times faster than $\mathcal{A}$, specifically with high thresholds (e.g., with $T = 0.9$, $\mathcal{A}_T$ is $> 3\times$ faster than $\mathcal{A}$). The speedup increases as the threshold increases; this is because more and more uncorrelated signal pairs can be pruned as the threshold increases.

►**Approximate Threshold Correlation.** Figure 6(a) shows the speedup of $\mathcal{A}_\epsilon$, with respect to $\mathcal{A}_T$, for different approximation error bounds ϵ. As before, we use $k = 5$ DFT coefficients for pruning and $T = 0.9$ as the threshold. The graph shows that even with a very small error, e.g., 0.02, $\mathcal{A}_\epsilon$ is significantly faster than $\mathcal{A}_T$ for all real datasets. For example, with $\epsilon = 0.02$, the speedups for the $Chlorine$ and $DataCenter2$ datasets are 17 and 9 respectively. The speedup is small for $RandomWalk$, because most of its energy is captured by a very few of its leading coefficients [1], helping the baseline algorithm $\mathcal{A}_T$ to perform extremely good (also shown in Figure 5(c)) with such data. The speedup increases with error tolerance, as this allows $\mathcal{A}_\epsilon$ to compute fewer DFT coefficients.

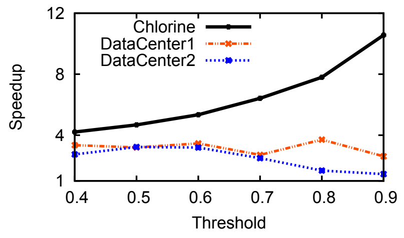
Figure 6(b) shows the speedup of $\mathcal{A}_\epsilon$ for different thresholds. In general, the absolute execution time of $\mathcal{A}_\epsilon$ is not affected much by different thresholds. In contrast, the baseline algorithm $\mathcal{A}_T$ runs faster with bigger thresholds (as shown in Figure 5(c)). Therefore, the speedup of $\mathcal{A}_\epsilon$ with respect to $\mathcal{A}_T$ decreases with increasing threshold, as shown in Figure 6(b).



(a) Impact of varying ϵ



(b) Impact of thresholds



(c) Boolean approximation

Figure 6: Speedup for computing an approximate and a Boolean threshold correlation matrix. The speedup is computed with respect to computing an exact threshold correlation matrix.

Dataset	CPU time (minutes)			
	$\mathcal{A}$	$\mathcal{A}_T$	$\mathcal{A}_\epsilon$	$\mathcal{A}_B$
$DataCenter1$	106	50.7	9.5	18.2
$DataCenter2$	19	12	0.85	4.9
$Chlorine$	98	67.5	3.5	10.4
$RandomWalk$	207	35.2	6.6	29.0

Table 3: Absolute CPU time for different algorithms, with $T = 0.7$ **and** $\epsilon = 0.04$

►**Boolean Threshold Correlation.** Finally, we test our $\mathcal{A}_B$ algorithm with varying thresholds. Figure 6(c) shows speedups for different datasets. In most of the cases speedup is more than 2 and for $Chlorine$ it reaches up to $> 10\times$ for $T = 0.9$. In general, $\mathcal{A}_B$ is slower than $\mathcal{A}_\epsilon$; this is because $\mathcal{A}_\epsilon$ needs to search for good verifier signals in order to avoid any false positives and negatives.

►**Speedup with respect to $\mathcal{A}$.** Since the speedup of $\mathcal{A}_T$ is reported with respect to $\mathcal{A}$, and other speedups are reported with respect to $\mathcal{A}_T$, we can combine the speedups. For example, for the $DataCenter1$ dataset, $\mathcal{A}_\epsilon$ is 18.75 times faster than $\mathcal{A}$, for a threshold $T = 0.9$ and an error bound $\epsilon = 0.06$. Similarly, $\mathcal{A}_B$ is 8.34 times faster than $\mathcal{A}$, for a threshold $T = 0.9$. For the $DataCenter2$ dataset these numbers are 26.4 and 4.1.[5]

►**Absolute savings.** Table 3 shows the absolute CPU time of different algorithms on different datasets. This highlights that, in addition to significant relative speedups, our algorithms have significant absolute savings in execution time of different correlation queries.

In none of the experiments above, our algorithms result in a speedup less than 1. This highlights that our algorithms are never slower than $\mathcal{A}_T$ or $\mathcal{A}$ with our datasets.

7.3 Lagged Correlation

►**Error.** As mentioned in Section 6.2, our algorithm for computing lagged C^ϵ may violate ϵ-approximation guarantee. We now experimentally measure the effect for computing lagged C^ϵ with $\epsilon = 0.04$. After computing lagged C^ϵ, we count the number of signal pairs that violate the ϵ-approximation guarantee; i.e., for which the true maximum lagged correlation is more than ϵ away from our estimated maximum lagged correlation. We define the percentage of signal pairs violating the approximation guarantee as the error of our algorithm due to lag. Note that, without any lag, our algorithm has an error of 0 as it never violates the approximation guarantee.

Figure 7 shows the error of our algorithm as a function of maximum lag. Lags are shown as percentages of the entire signal lengths ($= m$). As shown, the error is very small for all datasets for reasonable lag values. In particular, $Chlorine$ and $RandomWalk$ incur close to zero error with a maximum lag of 5% of the entire signal length. All datasets have errors $< 2\%$ even for a large maximum lag of 20% of an entire signal. Lags are typically much smaller in practice; e.g., for $DataCenter1$, a signal represents the entire day, and hence 10% lag means a lag of 2.4 hours, which is extremely unlikely in a data center. Thus, for practical values of maximum lag, our algorithm incurs a very small error.

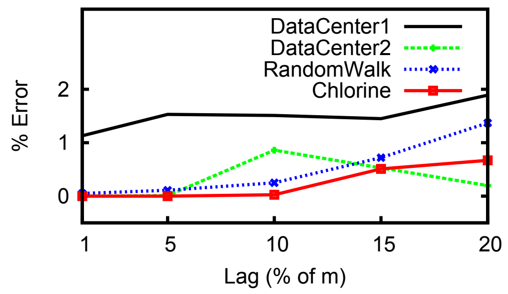


Figure 7: Errors in lagged correlation.

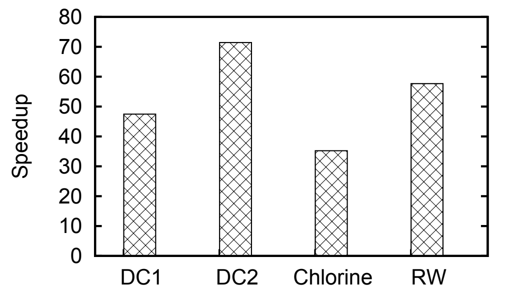


Figure 8: Speedup for lagged correlation (DC1=$DataCenter1$, DC2=$DataCenter2$, RW=$RandomWalk$).

►**Speedup.** The small error above comes with a significant benefit of speedup. In Figure 8, we report the speedup of our algorithm to compute a lagged C^ϵ for $\epsilon = 0.04$, with respect to compute C^T with BRAID [21], the state-of-the-art algorithm for computing lagged correlation. Both BRAID and our algorithm are configured to compute correlation coefficients for 16 different lags (recall that BRAID considers logarithmic number of lags). Figure 8 shows that our algorithm is $35\times$ to $71\times$ faster than BRAID for different

[5]We assume a large cache, so that I/O cost is small, to compute the speedup.

datasets. This huge speedup comes because our algorithm works in frequency domain and reuses DFT coefficients across different lags.

8. CONCLUSION

We have proposed novel algorithms, based on Discrete Fourier Transform (DFT) and graph partitioning, to reduce the end-to-end response time of an all-pair correlation query. To optimize I/O cost, we intelligently partition a massive input signal set into smaller batches such that caching the signals one batch at a time minimizes disk I/O. To optimize CPU cost, we have proposed two approximation algorithms. Our algorithms have strict error guarantees, which makes them as useful as corresponding exact solutions for many real applications. However, compared to the state-of-the-art exact solution, our algorithms are up to 17× faster for several real datasets.

9. REFERENCES

[1] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, 1993.

[2] R. Cole, D. Shasha, and X. Zhao. Fast window correlations over uncooperative time series. In *SIGKDD*, pages 743–749, 2005.

[3] G. Even. Fast approximate graph partitioning algorithms. *SIAM J. Comput.*, 28(6):2187–2214, 1999.

[4] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.

[5] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey. The node capacitated graph partitioning problem: a computational study. *Math. Program.*, 81(2):229–256, 1998.

[6] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82: Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[7] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD*, 2001.

[8] A. Hagen, L. Kahng. Fast spectral methods for ratio cut partitioning and clustering. In *Computer-Aided Design, 1991.*, pages 10–13, 1991.

[9] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[10] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *KAIS*, 3:263–286, 2000.

[11] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *SIGMOD Rec.*, 30:151–162, 2001.

[12] C.-S. Li, P. S. Yu, and V. Castelli. HierarchyScan: A hierarchical similarity search algorithm for databases of long sequences. In *ICDE*, 1996.

[13] X. Lian and L. Chen. Efficient similarity search over future stream time series. *TKDE*, 20(1):40–54, 2008.

[14] X. Lian, L. Chen, and B. Wang. Approximate similarity search over multiple stream time series. In *Advances in Databases: Concepts, Systems and Applications*, pages 962–968, 2008.

[15] X. Lian, L. Chen, J. X. Yu, J. Han, and J. Ma. Multiscale representations for fast pattern matching in stream time series. *TKDE*, 21(4):568–581, 2009.

[16] C. Loboz, S. Smyl, and S. Nath. Datagarage: Warehousing massive amounts of performance data on commodity servers. Technical Report MSR-TR-2010-22, Microsoft Research, March 2010.

[17] S. Papadimitriou, J. Sun, and C. Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB*, pages 697–708, 2005.

[18] D. Rafiei. On similarity-based queries for time series data. In *ICDE*, 1999.

[19] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *SIGMOD*, 1997.

[20] G. Reeves, J. Liu, S. Nath, and F. Zhao. Managing massive time series streams with multiscale compressed trickles. In *VLDB*, 2009.

[21] Y. Sakurai, S. Papadimitriou, and C. Faloutsos. BRAID: stream mining through group lag correlations. In *SIGMOD*, 2005.

[22] M. Vlachos, S. S. Kozat, and P. S. Yu. Optimal distance bounds on time-series data. In *SDM*, 2009.

[23] M. Vlachos, C. Meek, Z. Vagena, and D. Gunopulos. Identifying similarities, periodicities and bursts for online search queries. In *SIGMOD*, 2004.

[24] M. Wang, S. Lim, J. Cong, and M. Sarrafzadeh. Multi-way partitioning using bi-partition heuristics. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, 2000.

[25] H. Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 50–55, 1994.

[26] B.-K. Yi, N. Sidiropoulos, T. Johnson, A. Biliris, H. Jagadish, and C. Faloutsos. Online data mining for co-evolving time sequences. *ICDE*, 2000.

[27] Y. Zhu and D. Shasha. StatStream: statistical monitoring of thousands of data streams in real time. In *VLDB*, 2002.