Correlation Joins over Time Series Data Streams Utilizing Complementary Dimension Reduction and Transformation

AMIRREZA ALIZADE NIKOO, University of Zurich, Switzerland
MICHAEL H. BÖHLEN, University of Zurich, Switzerland
SVEN HELMER, University of Zurich, Switzerland

A common analysis task over a stream of time series is to find all pairs of windows whose correlation is above a given threshold. For a large number of streams, doing so naively, i.e., checking the Cartesian product, is too expensive. In essence, finding correlated pairs in a non-naive way boils down to a high-dimensional similarity join in a Euclidean space. While there are similarity join algorithms, such as Quickjoin and ϵ-kdB tree, they are inefficient for high-dimensional data. We propose *CorrJoin*, short for Correlation Join, that combines a *complementary dimension reduction and transformation* step with a subsequent *double-filtering* step. In the first step, we reduce the dimensionality of data stream windows by combining a fast but inaccurate method, Piecewise Aggregate Approximation (PAA), with an accurate and slow one, Singular Value Decomposition (SVD). Not only does SVD compensate for the weaknesses of PAA, it also transforms the data to make the first filter based on bucketing more effective. The second filter, which uses Euclidean distances, reduces the number of false positives before computing exact correlations. Our experiments reveal that in common settings, *CorrJoin* is an order of magnitude faster than state-of-the-art approaches (up to 20 times faster than Quickjoin).

CCS Concepts: • **Information systems → Join algorithms**.

Additional Key Words and Phrases: correlation, join processing, time series, data streams

ACM Reference Format:
AmirReza Alizade Nikoo, Michael H. Böhlen, and Sven Helmer. 2023. Correlation Joins over Time Series Data Streams Utilizing Complementary Dimension Reduction and Transformation. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 235 (December 2023), 26 pages. https://doi.org/10.1145/3626722

1 INTRODUCTION

Due to the ubiquity of sensors and sensor networks, the analysis of time series data in a timely manner or (near) real-time has gained in importance. Time series data analysis can be found in areas such as running scientific experiments, keeping track of industrial equipment and vehicles, collecting economic statistics, and monitoring the health of patients. An increasingly important application is monitoring the performance of thousands of servers in a data center [44]. There is a trend in this area to move towards observability platforms, such as Prometheus and Dynatrace, which provide more functionality compared to monitoring [37]. These platforms help DevOps engineers in analyzing an incident by supporting the search for root causes. This analysis very often involves investigating time series data (for instance, Prometheus stores its data as time series[1]) and finding correlations between these time series. For example, keeping track of the latency of a

[1]https://prometheus.io/docs/concepts/data_model/

Authors' addresses: AmirReza Alizade Nikoo, University of Zurich, Zurich, Switzerland, nikoo@ifi.uzh.ch; Michael H. Böhlen, University of Zurich, Zurich, Switzerland, boehlen@ifi.uzh.ch; Sven Helmer, University of Zurich, Zurich, Switzerland, helmer@ifi.uzh.ch.

2836-6573/2023/12-ART235
https://doi.org/10.1145/3626722

web application we observe several peaks of high latency. Taking a closer look at other time series we find that the latency of the backend database system behaves very similarly, i.e., the two latency time series are highly correlated, indicating an issue with the database system. Now, correlation does not necessarily mean causation, but it helps DevOps engineers to find the cause of an incident much faster.[2] Currently, the analysis is triggered after an incident and then relevant time series data have to be found in log files and compared to each other to find correlated series. With the help of our correlation join, we can keep track of the correlations between time series as the data becomes available, making it possible to provide this information instantly when needed. We envision a system that can provide observability in near real-time.

On a technical level, our goal is to efficiently compute correlation joins over time series data streams. Thus, given a data stream consisting of m unbounded time series, we determine all pairs of time series that are locally correlated in terms of the Pearson correlation. Monitoring an unbounded time series is handled by a window of length n that is shifted over each time series and uses the most recent n data points (dimensions) to answer correlation queries. Two windows are considered correlated if their correlation is above a given threshold, and uncorrelated otherwise. Determining correlated pairs boils down to evaluating a self join among the windows of all streams with a join predicate checking for correlation.

Partition- or hash-based joins are among the fastest join algorithms because they eliminate many pairs from consideration, and only look at a small subset of the Cartesian product containing potential join partners. When applying these kinds of joins for finding pairs of correlated windows in data streams, we face two challenges. First, the join predicate is not based on equality, which makes it difficult to apply hash-based algorithms. Second, we are dealing with high-dimensional data, as typical data stream windows contain hundreds or thousands of values. In order to tackle the first challenge, we map the Pearson correlation to the Euclidean distance and determine pairs of windows with a distance less or equal to ϵ (which is the distance threshold derived from the correlation threshold (see Section 2.1)). By assigning the windows to buckets of width ϵ, we eliminate all pairs of windows from non-neighboring buckets. However, we cannot apply the bucketing approach directly as the high dimensionality of the data would lead to buckets with a large number of neighbors (this number increases exponentially with the number of dimensions). At the heart of our novel approach *CorrJoin*, short for *Correlation Join*, is a *complementary dimension reduction and transformation*. Using an innovative combination of Piecewise Aggregate Approximation (PAA), an efficient but not very accurate dimensionality reduction technique, with Singular Value Decomposition (SVD), an accurate but slow method, we are able to efficiently leverage bucketing for the computation of correlation joins.

In the following, we briefly describe the steps of our algorithm (see Figure 2 for a visualization of its overall structure). The first step of CorrJoin is responsible for preprocessing the input time series windows by reducing their dimensionality via PAA, which is able to do this very efficiently while retaining aggregated information. We actually produce two sets of dimensionality-reduced windows. The windows in the first set are reduced to k_s dimensions and are fed into SVD, while the windows in the second set are reduced to k_e dimensions and are used in a later filter step involving the Euclidean distance. SVD prepares the data for bucketing by transforming the dimensions and selecting the k_b dimensions that best spread the data (i.e., have the highest variances). The second step of CorrJoin utilizes *double-filtering*, passing the data from step one through two filters. Considering a quadratic number of pairs, i.e., all pairs of the Cartesian product, is clearly inefficient. So, the first filter, called *bucketing filter*, partitions the data into buckets of width ϵ and only compares windows located in the same or neighboring buckets. The earlier mentioned SVD transformation

[2]https://cloud.redhat.com/blog/observability-superpower-correlation

ensures that the data is spread across buckets in the most effective way. Since the bucketing filter is not perfect, i.e., the output includes pairs that are not correlated, the pairs that pass the bucketing filter are passed on to a second filter, the *Euclidean distance filter*, to check if the Euclidean distance is below distance threshold ϵ using the windows with k_e dimensions. This further decreases the number of pairs for which the exact correlation has to be computed.

In summary, CorrJoin strikes a good balance between performance (from PAA) and accuracy (from SVD). PAA-reduced windows are essential to speed up the SVD transformation and the Euclidean distance filter. SVD in turn is needed to make the bucketing work efficiently. In particular, our contributions are the following:

- We show how *dimensionality reduction*, *dimensionality transformation*, *bucket filtering*, and *distance filtering* must be combined to get a robust reduce-filter-and-refine approach for time series data that scales not only with the length but also with the number of time series. While each of these techniques is well understood, none of them leads to an efficient algorithm if used on its own. For a correlation join, these methods have to be combined in novel and very specific ways to reach a high level of efficiency.
- We introduce a novel approach for computing threshold-based correlations between windows of time series by combining Piecewise Aggregate Approximation (PAA), a fast but inaccurate reduction technique, with Singular Value Decomposition (SVD), a slow but accurate transformation and reduction technique.
- We decrease the number of pairwise comparisons by a *bucketing filter* that uses the most important dimensions for partitioning the windows into buckets. The bucketing guarantees that only windows in the same or in neighboring buckets must be compared. We present the first approach that leverages complementary dimension reductions and transformations to improve the efficiency of a bucketing filter.
- Using synthetic and real datasets, we conduct an extensive set of experiments to evaluate the performance of CorrJoin and compare it to the state-of-the-art approaches. Our experiments reveal that the proposed algorithm is at least an order of magnitude faster than the state-of-the-art approaches, and reduces the number of comparisons by at least 80% for widely used ranges of correlation thresholds.

2 DEFINITIONS AND TERMINOLOGY

We start out by defining the notation and terminology used in this paper. Throughout, we consider a dataset with m time series.

Definition 2.1. A *time series* $x_p = \{x_p[1], x_p[2], x_p[3], \dots\}$, $1 \leq p \leq m$, is an unbounded sequence of data points that are equally spaced in time. Each data point is a real number.

Definition 2.2. A *window* $w_p[i+1..i+n]$ of a time series x_p is a continuous subsequence of x_p of length n with starting time $i+1$, i.e., $w_p[i+1..i+n] = \{x_p[i+1], x_p[i+2], \dots, x_p[i+n]\}$.

As new data arrives windows are shifted by h newly arriving data points, called the *stride*. We write $w_p[\alpha h+1..\alpha h+n]$ to refer to window α of time series x_p with stride h. A window of length n is an n-dimensional object, in which each data point represents a dimension.

We abbreviate $w_p[i+1..i+n]$ and $w_q[i+1..i+n]$ by $x = \{x[1], x[2], \dots, x[n]\}$ and $y = \{y[1], y[2], \dots, y[n]\}$ with averages $\bar{x}$ and $\bar{y}$, respectively. The Pearson correlation is defined for

equal-length windows, x and y:

$$corr(x,y) = \frac{\sum_{i=1}^{n}(x[i]-\bar{x})(y[i]-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x[i]-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y[i]-\bar{y})^2}} \tag{1}$$

It is well known that the Pearson correlation can be computed incrementally over the windows [4]. We refer to this approach as incremental Pearson (IncP), which computes $s_1 = \sum_{i=1}^{n} x[i]$, $s_2 = \sum_{i=1}^{n} x[i]^2$, $s_3 = \sum_{i=1}^{n} y[i]$, $s_4 = \sum_{i=1}^{n} y[i]^2$, $s_5 = \sum_{i=1}^{n} x[i] \cdot y[i]$, and assembles these sums to

$$corr(x,y) = \frac{ns_5 - s_1 s_3}{\sqrt{(ns_2 - s_1^2)(ns_4 - s_3^2)}} \tag{2}$$

Ideally, we only calculate the exact correlation (according to Equation (1) or (2)) for a small number of potentially correlated pairs. Dimensionality reduction techniques achieve this by reducing the length of the windows and checking the correlation strength in a lower dimensional space, which can be done more cheaply. We use Piecewise Aggregate Approximation (PAA) as a dimensionality reduction technique since it is one of the fastest methods.[3] It guarantees that no correlated pairs are missed, i.e., there are no *false negatives*. *False positives*, i.e., non-qualifying pairs in the candidate set are eliminated afterward. Although PAA is very efficient, it is not very accurate, i.e., it produces a substantial number of false positives, especially if we drastically reduce the number of dimensions to very low values. In order to compensate for this, we employ Singular Value Decomposition (SVD) in combination with PAA. In the following, we briefly describe the two methods.

2.1 Piecewise Aggregate Approximation (PAA)

Piecewise Aggregate Approximation (PAA), also known as Piecewise Constant Approximation (PCA) [34, 49], reduces the dimensionality of a window from n to k dimensions by dividing it into k equi-length segments with the segment length equal to $\frac{n}{k}$ [24, 56]. The reduced representation is a vector containing the mean values of the segments.

Example ($k = 4$):

- $x = \{6, 2, 3, 1, 4, 5, 8, 7\}$
- $X = PAA(x, 4) = \{4, 2, 4.5, 7.5\}$

The time complexity for reducing the dimensionality of a window using PAA is $O(n)$ [30]. Whenever h new data points become available, the mean value vector is updated at cost $O(h + k)$.

Let $\hat{x} = \{\hat{x}[1], \hat{x}[2], \ldots, \hat{x}[n]\}$ be the normalization of window x such that $\hat{x}[i] = (x[i] - \bar{x})/\sqrt{\sum_{i=1}^{n}(x[i]-\bar{x})^2}$ where $\bar{x}$ is the average of x. The Euclidean distance d between two windows is the square root of the sum of the squares of the differences between the corresponding data points of the windows:

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x[i]-y[i])^2} \tag{3}$$

The Pearson correlation $corr$ can be expressed in terms of the Euclidean distance d [58], [43]:

$$corr(x,y) = 1 - \frac{1}{2}d^2(\hat{x},\hat{y}) \tag{4}$$

The higher the correlation between two pairs is, the smaller their distance becomes, as according to Equation 4 the correlation and the distance are inversely related. It has been shown that the

[3] Also, Keogh et al. [24] have shown that there is no big difference between various dimensionality reduction techniques for time series, but that PAA has a slight edge over the other methods.

Euclidean distance between windows X and Y whose dimensions have been reduced with PAA lower bounds the distance between the original windows x and y: this guarantees that there are no false negatives (but we still need to remove false positives afterward). More precisely, $d(X, Y) \leq \sqrt{k/n} \cdot d(x, y)$ [24, 56]. Thus, assuming the PAA-representations of the normalized windows are denoted by $\widehat{X}$ and $\widehat{Y}$, we get $d(\widehat{X}, \widehat{Y}) \leq \sqrt{k/n} \cdot d(\hat{x}, \hat{y})$. According to this property and Equation 4, we determine if $corr(x, y) \geq T$ by checking $d(\widehat{X}, \widehat{Y}) \leq \epsilon$ where $\epsilon = \sqrt{2k(1-T)/n}$, since

$$corr(x, y) \geq T \Rightarrow 1 - 1/2d^2(\hat{x}, \hat{y}) \geq T \Rightarrow d(\hat{x}, \hat{y}) \leq \sqrt{2(1-T)}$$

Thus, $d(\widehat{X}, \widehat{Y}) \leq \sqrt{k/n} \cdot d(\hat{x}, \hat{y}) \leq \sqrt{k/n} \cdot \sqrt{2(1-T)}$.

2.2 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a matrix decomposition technique that decomposes a matrix M into three matrices U, D, $V^\intercal$ (transpose of V) such that $M = UDV^\intercal$ [25, 28, 51]. The matrices are defined as follows:

- Matrix M is an $m \times n$ matrix that consists of m n-dimensional windows.
- Matrix U is an $m \times p$ matrix where $p = min(m, n)$. The columns of U are the orthonormal eigenvectors of $MM^\intercal$.
- Matrix D is a $p \times n$ diagonal matrix with nonnegative elements $\sigma_1, \sigma_2, \ldots, \sigma_n$ along its diagonal. These elements are the square roots of the eigenvalues of $M^\intercal M$, and are ranked in decreasing order such that $\sigma_1 \geq \sigma_2 \geq \ldots, \geq \sigma_n \geq 0$.
- Matrix V is an $n \times n$ matrix whose columns are the orthonormal eigenvectors of $M^\intercal M$.

As an example, consider three windows: $a = \{3, 1\}$, $b = \{-1, 2\}$ and $c = \{2, 4\}$. Then, we have

$$\begin{matrix} a \leftarrow \\ b \leftarrow \\ c \leftarrow \end{matrix} \overbrace{\begin{bmatrix} 3 & 1 \\ -1 & 2 \\ 2 & 4 \end{bmatrix}}^{\mathbf{M}(3\times2)} = \overbrace{\begin{bmatrix} 0.5 & -0.7 \\ 0.2 & 0.7 \\ 0.9 & 0.2 \end{bmatrix}}^{\mathbf{U}(3\times2)} \times \overbrace{\begin{bmatrix} 5.2 & 0 \\ 0 & 2.8 \end{bmatrix}}^{\mathbf{D}(2\times2)} \times \overbrace{\begin{bmatrix} 0.6 & 0.8 \\ -0.8 & 0.6 \end{bmatrix}}^{\mathbf{V}^\intercal(2\times2)}$$

The product UD contains a set of windows that are rotated from the original windows in matrix M. As they are merely rotated, their Euclidean distances are preserved. The dimensionality reduction of the windows from n to k is performed by discarding (or zeroing-out) the $n - k$ smallest σ_i from matrix D (yielding the truncated version D_k) and also discarding the $n - k$ corresponding entries in matrix U (yielding the truncated version U_k). The product $U_k D_k$ consists of k-dimensional windows in the reduced dimensionality space.

SVD must examine the entire dataset and collects the m windows of all series in a matrix M. SVD has a computational complexity of $O(m(np + k^2))$ for computing matrices U and D and reducing the dimensionality of windows.

3 RELATED WORK

We first look at related work that uses dimensionality reduction to implement approaches for *lowering the costs of distance (or correlation) computations*. These approaches do not decrease the number of pairwise comparisons, they only speed up the distance computations.

Agrawal et al. [2] authored a seminal work about efficiently comparing two time series at scale. They measure the similarity of two sequences using the Euclidean distance, reducing the dimensionality of the time series via DFT. Additionally, they demonstrate how to map the Pearson correlation to the Euclidean distance (with the help of Parseval's theorem [47]), making their approach applicable for finding highly correlated time series. The idea of speeding up the correlation computation between time series by reducing their dimensionality can also be found in the following

approaches: Popivanov and Miller [41] use wavelet transformation instead of DFT, Chen et al. [8] propose the use of linear approximation, and Qiu et al. [42] combine DFT with a neural network. Mueen et al. [36] show how to improve the performance by optimizing the I/O cost of the DFT-based approach. Xu et al. propose TSUBASA [53], which focuses on storing historical time series data in a preprocessed form to accelerate the search for correlated pairs later on. The main differences to our approach are that the data is queried an arbitrary number of times (which makes the preprocessing worthwhile) and that all pairs are compared to each other during query evaluation. While TSUBASA can be applied in an online setting, it cannot play out its strength in such an environment and behaves similarly to the naive *IncP* approach. Several other methods have been used in the past to reduce the dimensionality of windows and use a filter-and-refine approach. These range from Piecewise Aggregate Approximation (PAA) [34, 49], Adaptive Piecewise Constant Approximation (APCA) [7, 14, 21], Multiscale Segment Mean (MSM) [32], Discrete Haar Wavelet Transform (DHWT) [19, 26], Piecewise Linear Approximation (PLA) [8, 23], Singular Value Decomposition (SVD) [28, 51], all the way to Chebyshev Polynomials [5]. Additionally, there are approaches that approximate the answer, i.e., they produce false negatives. There is an approach based on neural networks [42], one on hierarchical domain transformation [29], another DFT-based approach [58], and two computing dot products with random vectors [9, 54].

Next, we look at approaches that propose solutions to *reduce the number of pairs that have to be compared*. These approaches avoid checking a quadratic number of pairs but their performance deteriorates once we have significant costs for comparing two windows, which is the case for time series data. Generally, we have the following alternatives to avoid all pairwise comparisons: index-based approaches, in which matching objects are looked up via an index, sort-based approaches, in which objects are first sorted on a dimension and then scanned in an interleaved fashion, and partitioning/hash-based approaches, in which objects are partitioned and only compatible partitions are considered. From these alternatives, we do not further pursue sort-based approaches, as they have been shown to be outperformed by partitioning algorithms [20].

We first look at index structures that support queries searching for all time series that are similar to a given query time series [6, 11, 15, 27, 33, 39, 40, 46, 50, 59]. The two major issues with these indexes are that they are expensive to construct and they struggle with high-dimensional data [12, 13, 17, 18, 52]. For our purposes, we need light-weight methods allowing us to build the indexes efficiently on the fly, as we throw them away after the current batch of windows has been processed and need to build new ones for the next batch. Additionally, we are dealing with high-dimensional windows, a use case indexes generally struggle with. The only suitable index we found is the ϵ-kdB tree by Shim et al. [46], with which we compare our approach.

Next, we discuss partition-based approaches. Jacox and Samet [20] apply the idea of Quicksort to find the most similar data objects, creating their Quickjoin algorithm. They use a reference point in space and the distance of the points in the dataset to this reference point to partition the data. This approach is used recursively to divide the dataset into smaller and smaller subsets. Once these subsets reach a size that is small enough, finding potential matches is started inside the subsets and in the neighboring subsets. This approach has been shown to be highly effective, outperforming sort-based approaches, which is why we compare our approach to it. Sarma et al. [10] have generalized this approach to make it more suitable for parallel processing. There are approximate methods for identifying candidate pairs in high-dimensional spaces: one based on locality sensitive hashing (LSH) [1] and another based on randomly shifted grids [3]. Since these methods produce false negatives, we do not consider them here.

Finally, complementary to our work are approaches using correlation to improve the accuracy of predictions: Yi et al. [55], Gao et al. [16], and Lian et al. [31]. They rely on utilizing correlation for

their predictions, but do not propose new algorithms for the efficient computation of threshold-based correlations.

4 CORRELATION JOIN

Our goal is to determine all pairs of windows whose Pearson correlation is above threshold T. As we map the correlation to the Euclidean distance (see Equation (4)), this boils down to returning pairs whose distance is less or equal to ϵ. We start by illustrating how the bucketing filter works in principle and why it should not be applied naively to a correlation join. The difficulties we run into serve as a motivation for the solution we propose: a carefully crafted pipeline with several steps (cf. Figure 2). We show how every single one of the difficulties can be overcome by combining complementary dimensionality reduction and transformation techniques with a two-filter approach for maximal efficiency.

4.1 Bucketing Filter

We avoid comparing all m^2 pairs by partitioning the input windows of the m data streams into buckets. To keep things simple, we first show the principle for one-dimensional data points. We divide the value domain of the data points into buckets of width ϵ, i.e., bucket b contains the data points whose value w falls into the interval $[(b-1)\epsilon, b\epsilon)$, i.e., the first bucket is labeled $b = 1$. Thus, given a value w, $\lfloor \frac{w}{\epsilon} \rfloor + 1$ is the label of the bucket w belongs to.

Example 4.1. Let $m = 6$, $\epsilon = 0.5$, and windows $w_1 = \{0.1\}, w_2 = \{0.2\}, w_3 = \{0.3\}, w_4 = \{1.2\}, w_5 = \{1.4\}, w_6 = \{1.8\}$. Window w_4 falls into bucket $\lfloor \frac{1.2}{0.5} \rfloor + 1 = 3$, which means that w_4 is located in bucket 3. Figure 1 shows the final bucketing (Bkt is used as an abbreviation of bucket).

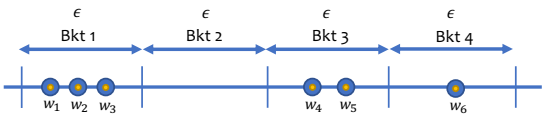

Fig. 1. An example for window bucketing

Bucketing facilitates the search for pairs of data points within a distance of ϵ of each other, since it allows us to distinguish three different cases. First, the distance between two data points in the same bucket is definitely less than ϵ. Second, two data points located in non-neighboring buckets cannot possibly be within distance ϵ of each other. Finally, for data points from neighboring buckets we have to check if their distance is less or equal to ϵ.

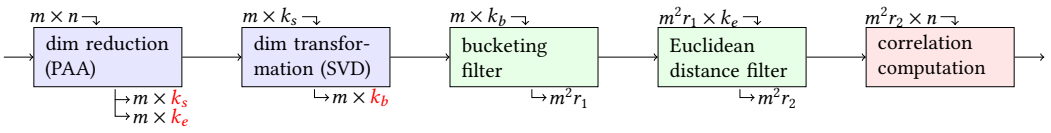

Fig. 2. CorrJoin computation pipeline: input dimensionality is stated above each box, output dimensionality and candidate set sizes (for filtering steps) below each box; parameters to be tuned are k_s, k_e and k_b.

4.2 Moving to Higher Dimensions

Bucketing also works for high-dimensional data points. However, applying the approach in a straightforward fashion leads to performance issues. Assuming that we have n-dimensional data points and that we need to divide the value domain of each dimension into B partitions of width ϵ, yields B^n hypercubes for the bucketing scheme and each hypercube has up to $3^n - 1$ neighbors. Applying bucketing to the windows of our data streams directly is infeasible since the windows have hundreds or thousands of dimensions. Instead, we need to reduce the number of dimensions before bucketing. However, reducing the number of dimensions negatively impacts the quality of the bucketing approach. For example, if we reduce the number of dimensions by either picking them randomly or using PAA, the distance between the data points in the reduced space is only a lower bound of their true distance. As a consequence, the dimensionality reduction moves data points closer together, meaning that we now have more data points assigned to the same bucket or neighboring buckets as before, even though their true distance is greater than ϵ. Data points located in the same or neighboring buckets must be compared. Reducing the dimensionality to low values to make bucketing feasible means that we get a rougher approximation of the original data. For the datasets we used in the experimental evaluation, we typically had to go down to three dimensions (based on the investigations shown in Figures 4 and 7), leading to inaccurate distance estimations using PAA.

In order to counteract the negative impact of dimensionality reduction, we apply Singular Value Decomposition (SVD) to the m windows computed by PAA. After applying SVD, we pick the dimensions with the highest variance, which leads to a better spread of data points among the buckets, increasing the likelihood of data points ending up in non-neighboring buckets. In the end, we have low-dimensional data (for our datasets, we used three dimensions), but the data created by applying PAA and then SVD reflects the original distances much better than picking dimensions randomly or using only PAA.

4.3 Euclidean Distance Filter

Before computing the correlation between pairs of windows from the same or neighboring buckets, we use a Euclidean distance filter to further reduce the number of candidate pairs, but this time using a higher number of dimensions compared to the dimensionality used by SVD before bucketing. We denote the number of dimensions used for SVD by k_s and the number of dimensions used for the Euclidean distance filter by k_e. By using k_e dimensions we obtain a better accuracy and, thus, fewer false positives. We can afford to use a higher number of dimensions, as we apply the comparison to a smaller number of pairs. Only the pairs passing the bucketing filter will go through the Euclidean distance filter.

4.4 Complete Algorithm

The complete CorrJoin algorithm assembles the different components into the pipeline shown in Figure 2. The input to the pipeline are m windows, each with n dimensions. In a first step, we reduce the dimensionality of the windows using PAA. We generate representations with k_s and k_e dimensions as depicted in Figure 3 (the original windows are preserved in an $m \times n$ matrix). We generate the k_e-dimensional representation at this point already to save another scan through the windows later on. The concrete values of k_s and k_e have a significant impact on the performance, and we discuss the tuning of these parameters in Section 6.2. The k_s-dimensional windows are fed into the algorithm computing the SVD. From the output of SVD we choose the first k_b dimensions for the bucketing step (the calibration of parameter k_b is discussed later). This produces a set of candidate pairs C_1 that contains r_1 percent of the pairs in the Cartesian product. We call the pruning

rate for this step the *join pruning rate* and it is equal to $1 - r_1$. Note that we do not pass the actual windows through the pipeline but their matrix indices (red numbers in Figure 3) for lookup during the processing. The values for r_1 observed in our experimental evaluation are typically below 40%. We apply a Euclidean filter step on this candidate set, this time using the k_e-dimensional windows. This yields candidate set C_2 with r_2 percent of the Cartesian product. Consequently, r_2 determines the *overall pruning rate*, which is equal to $1 - r_2$. Typical values for r_2 are below 10%. Finally, we compute the true value of the Pearson correlation for all pairs in the second candidate set, resulting in the overall answer set.

Algorithm 1 shows the pseudocode for our CorrJoin algorithm. For better readability, the bucketing is shown in Algorithm 2. Matrices W, W_s, W_b, and W_e with m rows and n, k_s, k_b and k_e columns, store the values of the windows in the original and reduced dimensionality spaces, respectively. Additionally, there is an input buffer B_p of size h, which holds the next batch of elements from time series x_p, i.e., in each step, h new values become available in the data streams. Operators $\ominus$ and $\oplus$ drop previous and add new data points to a window, respectively. In Algorithm 2, *BKT* includes all the buckets of windows after bucketing according to k_b dimensions. We note that in our implementation, we store the windows indices in the buckets rather than the windows itself.

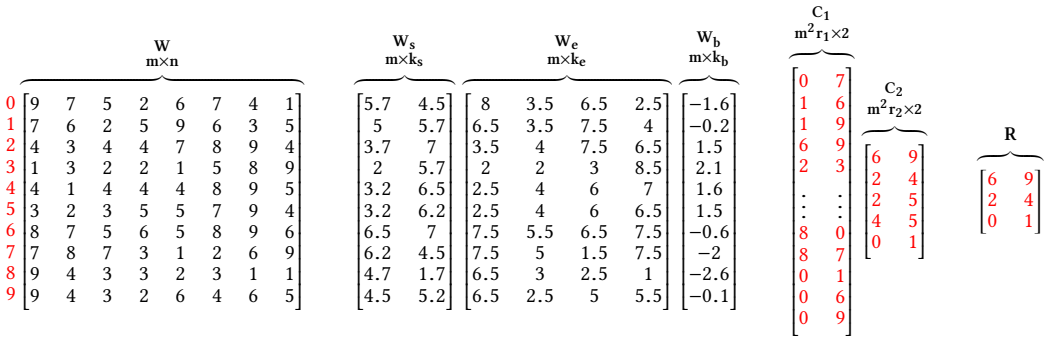

Fig. 3. Example for utilized data structures: matrices with m rows and n, k_s, k_b and k_e columns store windows in the original and reduced dimensionality spaces. Candidate sets C_1 and C_2 contain window pairs that have to be checked.

5 COST MODEL AND ANALYSIS

We now determine the run-time of the pipeline illustrated in Figure 2 and compare it to the cost of the baseline algorithm. The total costs of the pipeline are comprised of five components, one for each phase: computing the PAA representation, computation of SVD, bucketing filter, a Euclidean distance filter, and the computation of the true correlation. If the time series data has not been normalized yet, we need to first do the normalization. Altogether, the total costs c_{total} are made up of

$$(c_{norm} +)\ c_{PAA} + c_{SVD} + c_{bkt} + c_{filter} + c_{true} \tag{5}$$

We look at all these subcosts in turn and compare them to the costs of the brute-force baseline approach, which computes the Pearson correlation for all m^2 pairs on the full-sized windows, resulting in $c_{base} = O(m^2 n)$.

The normalization of the time series data can be done in time mn, so

$$c_{norm} = O(mn) \tag{6}$$

Algorithm 1: CorrJoin($x_1, ..., x_m, n, h, T, k_s, k_e, k_b$)

Input: Stream with m series $x_1, x_2, \ldots, x_m$ (each with an input buffer B_p), window size n, stride h, correlation threshold T, number of dimensions k_s, k_e, k_b
Output: Report threshold-based correlations

$\epsilon_1 \leftarrow \sqrt{2k_s(1-T)/n}$, $\epsilon_2 \leftarrow \sqrt{2k_e(1-T)/n}$
$\alpha \leftarrow 1$
while *input buffers are arriving* **do**
  for $p \leftarrow 1$ **to** m **do**
    $w_p[\alpha] \leftarrow w_p[\alpha-1][1..h] \ominus w_p[\alpha-1] \oplus B_p$
    $W[p] \leftarrow$ normalization of $w_p[\alpha]$ (see Section 2.1)
    $W_s[p], W_e[p] \leftarrow \text{PAA}(W[p], k_s, k_e)$
  $W_b \leftarrow \text{SVD}(W_s)$
  $C_1 \leftarrow \text{BucketingFilter}(W_b, k_b, \epsilon_1)$
  $C_2 \leftarrow \emptyset$
  for $\langle p, q \rangle \in C_1$ **do**
    if $d(W_e[p], W_e[q]) \leq \epsilon_2$ **then** $C_2 \leftarrow C_2 \cup \langle p, q \rangle$
  for $\langle p, q \rangle \in C_2$ **do**
    if $|corr(W[p], W[q])| \geq T$ **then** Report (p, q, α)
  $\alpha \leftarrow \alpha + 1$

Algorithm 2: BucketingFilter(W_b, k_b, ϵ)

Input: Matrix W_b of windows, number of dimensions k_b, distance threshold ϵ
Output: Candidate set of likely correlated pairs of windows

$C_1 \leftarrow \emptyset$
$m \leftarrow$ number of windows in W_b
$BKT \leftarrow$ initialize k_b-dimensional bucketing scheme
for $p \leftarrow 1$ **to** m **do**
  assign $W_b[p]$ according to k_b columns to $Bkt_i \in BKT$
for $Bkt \in BKT$ **do**
  for $W_b[i] \in Bkt$ **do**
    for $W_b[j] \in Bkt$ **do**
      if $i < j$ **and** $d(W_b[i], W_b[j]) \leq \epsilon$ **then** $C_1 = C_1 \cup \langle i, j \rangle$
  for *each neighbor* Bkt' *of* Bkt **do**
    for $W_b[i] \in Bkt$ **do**
      for $W_b[j] \in Bkt'$ **do**
        if $d(W_b[i], W_b[j]) \leq \epsilon$ **then** $C_1 = C_1 \cup \langle i, j \rangle$
return C_1

which is cheaper than c_{base} by a factor of m.

Computing the PAA representation has the same time complexity, so its costs are

$$c_{PAA} = O(mn) \tag{7}$$

Again, this is much faster than the baseline approach. These steps take little time (together less than 5%) and are preparation steps to speed up the following steps.

SVD's costs are quadratic in the number of dimensions. This is the reason why it requires low-dimensional data to perform efficiently. As the input data has k_s dimensions, we get

$$c_{SVD} = O(mk_s^2) \tag{8}$$

Clearly, k_s is always smaller than n, since we are reducing the dimensionality with PAA. However, we also have to make sure that k_s^2 is smaller than m, otherwise the computation of SVD will dominate the computation time.

Next we turn to the costs for the bucketing and joining. The cost for the bucketing is mk_b as assigning each window to a bucket is done considering k_b dimensions. Let m_i be the number of elements in the i-th bucket and let m_{ij} be the number of elements in the j-th neighbor of bucket i. Basically, we need to determine the costs for joining the elements within a bucket (the summation) and the costs for joining these elements with elements in neighboring buckets (the double-summation):

$$c_{bkt} = O(mk_b + \sum_{i=1}^{B^{k_b}} m_i^2 k_b + \sum_{i=1}^{B^{k_b}} \sum_{j=1}^{\frac{3^{k_b}-1}{2}} m_i m_{ij} k_b) \tag{9}$$

Clearly, choosing a value for k_b that is too large leads to a prohibitively expensive run time, due to the exponential growth of the summation limits. The reason we use bucketing is to prune many non-matching pairs without having to check them. Note that while the costs increase exponentially in k_b, the join pruning rate does not. On the contrary, increasing k_b leads to diminishing returns, i.e., the improvement of the pruning rate becomes smaller and smaller. On top of that, using SVD allows us to pick the dimensions in order of importance, i.e., every additional dimension we choose is worse than the preceding ones. We investigate this empirically in Section 6.2. Note that the costs c_{bkt} are impacted by the distribution of the data over the buckets. If the elements are clustered in a few buckets, we compute the join of a considerable number of pairs in a nested-loop fashion. This is the reason why it is important to spread out the elements over the buckets with the help of SVD.

The costs of the Euclidean distance filter applied next depend on the size $m^2 r_1$ of the first candidate set (r_1 is the ratio of pairs passing the bucketing filter, i.e., the join pruning rate is equal to $1 - r_1$):

$$c_{filter} = O(m^2 r_1 k_e) \tag{10}$$

Clearly, k_e is much smaller than n and $m^2 r_1$ is also smaller than m^2 (the candidate set contains a subset of the Cartesian product) as we see in the experimental evaluation. The smaller r_1, the faster this step will run. As $m^2 r_1$ is lower-bounded by the number of pairs in the final result set, this has implications on the use of our algorithm. If we expect a very large result set, we may be better off running the baseline approach. This is not surprising and true for all join algorithms, e.g., for a large result set, a hash join cannot play out its strengths compared to a nested-loop join (see also Section 6.3.3).

Finally, we have the costs for computing the answer set given the second candidate set produced by the filter step, which has size $m^2 r_2$. Thus,

$$c_{true} = O(m^2 r_2 n) \tag{11}$$

As in the second candidate set the number of candidate pairs is further reduced, r_2 is smaller than r_1, and the arguments we made for c_{filter} also hold for c_{true}. We achieve an overall pruning rate of $1 - r_2$.

6 EXPERIMENTAL EVALUATION

We evaluate our algorithm and compare it to the state-of-the-art and baseline approaches empirically. By doing so, we back up claims made earlier and demonstrate the effectiveness and efficiency of CorrJoin. We consider the following approaches:

- ϵ-kdB tree: partitions the data hierarchically into several tiles by randomly selecting dimensions. It reduces the number of comparisons by checking the windows inside a tile and neighboring tiles, and computes the correlations for qualifying pairs.
- TSUBASA: finds correlated pairs by considering the correlation of all pairs. It divides each window into multiple basic windows and obtains the statistics of basic windows to identify pairwise correlations among the windows.
- *Quickjoin*: uses a reference window and the distance of the windows in the dataset to this reference window to partition the data. It reduces the number of comparisons by recursively partitioning the data into subsets. Once the subsets reach a size that is small enough, finding potential matches is started inside the subsets and in the neighboring subsets. Finally, the correlations are computed for qualifying matches.
- $IncP^{PAA}$: computes correlated pairs by checking the correlation of all pairs. It speeds up the calculation of the correlation by using the PAA filter-and-refine approach. Note that we consider $IncP^{PAA}$ as our baseline approach, but it has been shown to be an efficient approach, performing an order of magnitude faster than naive $IncP$ [30].

6.1 Setup and Datasets

First, we describe our methodology, i.e., the setup of our experiments and the time series dataset we used. We implemented the algorithms from scratch in R without using additional libraries, as we want to measure their performance and not that of any libraries. We conducted the experiments on a 2.2 GHz 6-Core Mac with 16 GB of memory. We used the following time series datasets[4]:

- **Synthetic dataset** is a collection of time series generated by the random walk model [2]. Thus, each time series $x_p = \{x_p[1], x_p[2], \ldots, x_p[n']\}$ is generated as

$$x_p[i] = x_p[i-1] + z_i, \qquad i = 2, 3, \ldots, n' \tag{12}$$

  where x_1 and z_i are independent, identically distributed (IID) variables uniformly distributed in the range $(-1, 1)$. For each experiment with synthetic datasets, we generate as many time series as we need.
- **Random dataset** is a collection of time series where each time series $x_p = \{x_p[1], x_p[2], \ldots, x_p[n']\}$ is generated as $x_p[i] = z_i$, $i = 1, 2, \ldots, n'$, where z_i is an independent, identically distributed variable uniformly distributed in the range $(-1, 1)$.
- **Stock dataset** consists of daily closing prices for $m = 3878$ companies traded on the NASDAQ stock exchange from 2016 to 2020 collected by Yahoo Finance[5].
- **Gas dataset** contains 128 time series collected between 2007 and 2011 from a gas delivery platform situated at the ChemoSignals Laboratory at the University of California in San Diego (one measurement for every 6 hours) [45, 48]. We replicated every series in the dataset 40 times, creating one with $m = 5120$ series (as we are particularly interested in scaling issues).

[4]The code and datasets are available at https://drive.google.com/drive/folders/1skrE2x2DMgIms5lZR04kiOlLvvna2zNs
[5]https://finance.yahoo.com/

To each replica we added an IID variable uniformly distributed in the range $(-0.05, 0.05)$, so that all time series are unique and the correlation rate of the dataset roughly stayed constant.

- **Chlorine dataset** consists of 161 time series (pipe junctions) of a water distribution network and measurement of the Chlorine concentration level at all these junctions during 15 days (one measurement for every 5 minutes) [36, 38]. Every series in the dataset is replicated 30 times with small additive noise to keep the correlation rate constant, creating a dataset with $m = 4830$ series.

When we selected the datasets, we took care to consider the oscillation (or variability) of the time series. Basically, oscillation is the speed and scale of changes in amplitude. For instance, a high-oscillation dataset contains time series with many high peaks and low valleys. The oscillation has an impact on the performance of dimensionality reduction approaches, such as PAA: the lower the oscillation, the better these approaches can preserve the general shape of a time series. We selected datasets with low, medium, and high oscillation for our experiments: these are the chlorine, stock, and gas datasets, respectively.

6.2 Tuning our approach

Before comparing our approach to its competitors, we show how to tune parameters k_s, k_e, and k_b to achieve the best performance. We also demonstrate the impact of the different steps of the CorrJoin pipeline on the performance, illustrating why the particular setup we have chosen for the pipeline makes sense.

6.2.1 Dimensionality of Bucketing. We start by looking at the dimensionality k_b used for the bucketing, as this parameter drives the whole setup and determines the requirements for the dimensionality reduction step. Figure 4 shows the runtime for different values of k_b for the different parts of the pipeline. In each subfigure the costs for computing the exact correlations (red part) is the same, since the candidate set produced by the Euclidean distance filter is independent of k_b. We are particularly interested in the bucketing filter step (green part) and the Euclidean distance filter step (orange part). Increasing k_b makes the distance filter more effective due to the decreasing number of candidate pairs created by the bucketing filter, at the cost of increased computational overhead for bucketing. However, the gains become smaller and smaller, meaning that these are diminishing returns. The overall runtime first decreases, but when going beyond $k_b = 3$, it increases again, as a larger number of dimensions slows down the bucketing filter due to checking a growing number of buckets with an increasing number of neighbors.

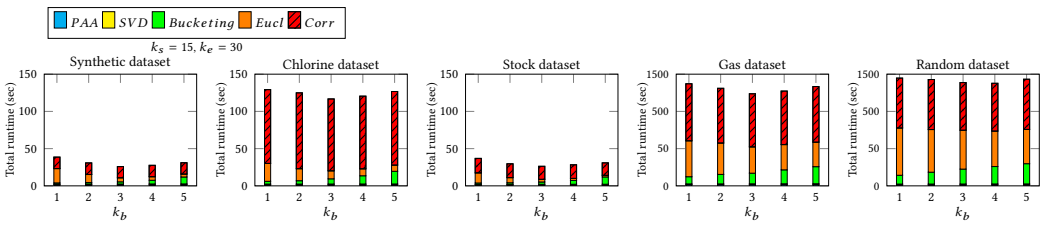

Fig. 4. Breakdown of runtime of CorrJoin for increasing k_b

Having to use a small value for k_b has a significant impact on the steps occurring before the bucketing. We have to be able to bring the number of original dimensions, n, down to $k_b = 3$ without losing too much accuracy. This is the topic of the following Section 6.2.2.

6.2.2 Evaluation of PAA and SVD. One of the easiest and fastest ways to reduce the dimensionality of a window down to $k_b = 3$ dimensions is to select the dimensions randomly. We call this approach CJ^{Rand}.

The left-hand side of Figure 5) shows that this leads to a very poor runtime, since randomly selected dimensions produce a lot of intermediate pairs that do not meet the correlation threshold. The right-hand side of Figure 5 confirms that randomly picking 3 dimensions in a window results in a terrible accuracy. The join pruning rate is almost zero for a correlation threshold of up to .95 and in the end merely reaches 20%. We compare the random selection of dimensions with the CJ^{PAA} approach, which uses PAA to reduce the dimensionality. PAA aggregates all available information in a window into mean values. This results in a much better overall runtime since the pruning rate of CJ^{PAA} is much higher (see Figure 5).

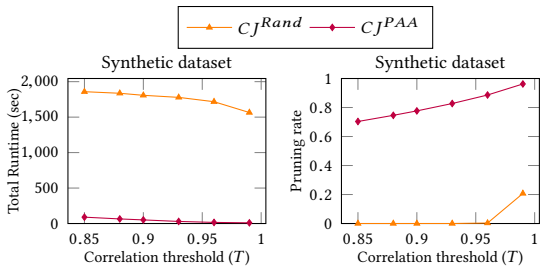

Fig. 5. Total runtime and join pruning rate of CJ^{Rand} and CJ^{PAA}

We can do better by leveraging SVD, a technique that transforms the data and allows us to pick dimensions with a higher variance than any individual dimensions in the original window. Running SVD directly on the original windows is out of the question, though. We ran some experiments and it took SVD more than an hour to compute the decomposition for windows with 1000 dimensions. Consequently, we must reduce the dimensionality of the original windows before using SVD. Our experiments show that 15-dimensional data as input for SVD gives good results (we will come back to this in Section 6.2.3). Thus, we apply a two-staged approach: in the first stage the dimensionality is brought down to 15 dimensions and in the second stage we apply SVD to transform and select dimensions. Figure 6 shows the results for using a random selection ($CJ^{Rand+SVD}$) and PAA ($CJ^{PAA+SVD}$) in the first stage, respectively. Again, the PAA variant performs much better than the random one. Although $CJ^{Rand+SVD}$ is better than CJ^{Rand}, it is still much worse than $CJ^{PAA+SVD}$ or CJ^{PAA}. The clear winner is $CJ^{PAA+SVD}$, which outperforms CJ^{PAA} by a factor of 2.5.

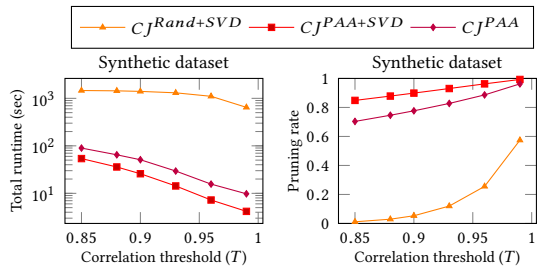

Fig. 6. Total runtime and join pruning rate of our approach with and without PAA

It is also interesting to note that we get diminishing returns out of SVD, i.e., the first couple of dimensions of a transformed window are the most important ones, as they exhibit the highest variability. With every additional dimension, the variability drops, see Figure 7 for an illustration. Thus, in general there is little reason to go beyond three dimensions for the bucketing, since the additional dimensions do not help in distinguishing windows from each other.

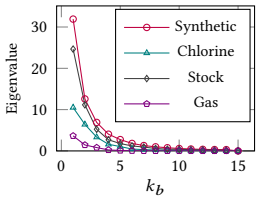

Fig. 7. Quickly falling eigenvalues for increasing k_b

In summary, by combining PAA and SVD we increase the efficiency of bucketing. PAA first reduces the dimensionality of a window to a level that allows us to make use of SVD, which in turn prepares the data for the bucketing by allowing us to transform dimensions and select dimensions with high variability.

6.2.3 Filtering and Refining. Before computing the true correlation and returning the answer set, we apply a Euclidean distance filter to check the pairs coming out of the bucketing filter. Figure 8 shows a clear difference in runtime when running our pipeline with and without this distance filter. Since computing the true correlation on the full-dimensional windows is more costly than doing so on windows with a reduced dimensionality, it is beneficial to first filter out more false positives. Since the bucketing phase has already removed many non-qualifying pairs, we can use a dimension count of $k_e = 30$ to obtain a better accuracy.

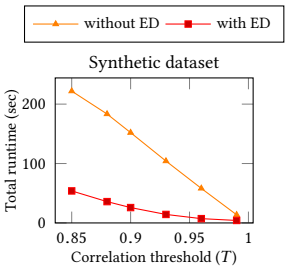

Fig. 8. Total runtime of CorrJoin with and without Euclidean distance computations

We now turn to optimizing parameters k_s and k_e, the dimensionality used for SVD and the Euclidean distance filter, respectively. This involves various trade-offs. Increasing k_s leads to a more accurate approximation of the windows at the cost of increased computational overhead for SVD. On the other hand, increasing k_e results in fewer candidate pairs at the cost of increased computational overhead for the distance filtering. Figure 9 shows a heatmap of the runtime of our approach for a range of values for k_s and k_e. We observe the best performance of CorrJoin for $k_s = 15$ and $k_e = 30$. These are the values we use for the remainder of our experiments. As an extreme case, we generate a random dataset whose series exhibit a random behavior, for which there is a change in the optimal values for k_s and k_e. However, such a dataset is rarely seen in a real world application and even if it does appear, it only changes the optimal parameter values slightly.

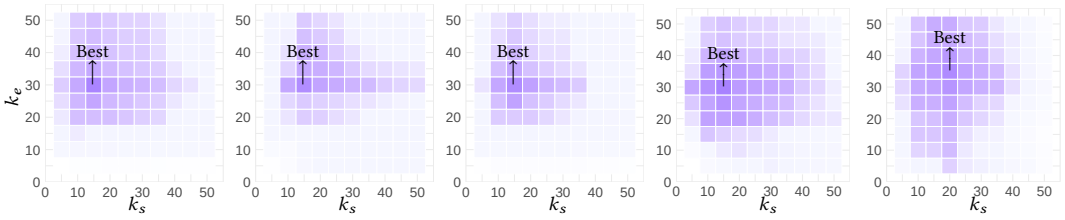

Fig. 9. Heatmap of the runtime of CorrJoin with varying k_s and k_e

6.3 Performance

We compare the performance of CorrJoin with the baseline approach $IncP^{PAA}$ and its competitors ϵ-kdB tree and Quickjoin. We look at the total runtime and pruning rates (join and overall), investigate the impact of low correlation thresholds, and follow up by showing the effects of increasing m, the number of time series, and increasing n, the size of the windows. Finally, we demonstrate how to improve the performance of Quickjoin by applying dimensionality-reduction techniques to it.

6.3.1 Total Runtime and Pruning Rate. Figure 10 compares the runtime and pruning rates of our approach to the ϵ-kdB tree, TSUBASA and the baseline approach (the join pruning rate is indicated with a dashed line, the overall pruning rate with a solid line). The ϵ-kdB tree executes the join by choosing the dimensions for partitioning the data sequentially one by one until no further partitioning is possible. Essentially, this means that each dimension is chosen randomly and the results in Figure 10 reflect this: they look very similar to the results for CJ^{Rand} in Figure 5 and confirm that the random selection of dimensions does not work for high-dimensional data. On the other hand, TSUBASA has a similar time complexity as the naive *IncP* since it does not reduce the number of pairwise comparisons. Since the ϵ-kdB tree and TSUBASA are an order of magnitude worse than the baseline approach, we drop them from further investigation and focus on the comparison of CorrJoin with Quickjoin (to better see the comparison between CorrJoin and $IncP^{PAA}$, look at the first plot in Figure 19).

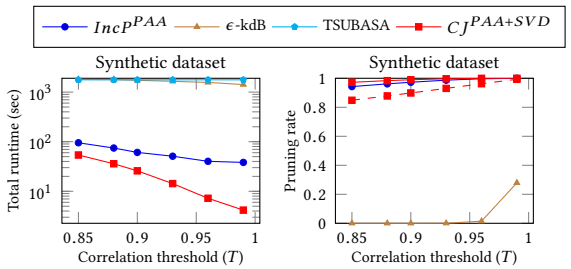

Fig. 10. Total runtime and pruning rate of CorrJoin, ϵ-kdB tree and TSUBASA

We also look into the performance of ϵ-kdB tree for different numbers of dimensions (n) in Figure 11. For a very low number of dimensions ($n \leq 30$), the ϵ-kdB tree shows a better performance than CorrJoin, but it quickly starts to deteriorate as the number of dimensions grows.

We continue by running CorrJoin and Quickjoin on three different real-world datasets: chlorine, stock, and gas, which exhibit a low, medium, and high variability, respectively. Figure 12 shows the total runtime and pruning rates for the chlorine, stock, and gas dataset. CorrJoin performs very

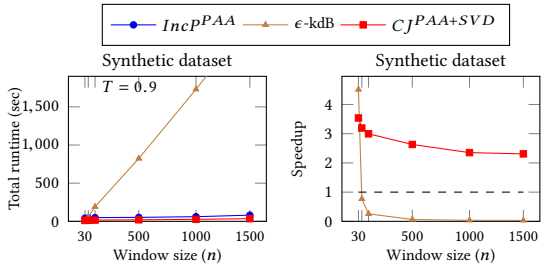

Fig. 11. The ϵ-kdB tree performs better for very small n, but quickly starts to deteriorate as n increases

well on the datasets exhibiting a low or medium variability (chlorine and stock), since in these cases PAA provides a more accurate approximation of the windows in the reduced dimensionality space. PAA is able to preserve the overall shape of a time series more accurately, which in turn makes the output of SVD in the following step more accurate as well. In terms of the runtime, CorrJoin outperforms Quickjoin by a factor of up to 22 and $IncP^{PAA}$ by up to an order of magnitude ($T \geq 0.95$). Dimensionality reduction techniques do not work as well for datasets with a high variability (this is a worst case for our technique). This is why for gas dataset, the pruning rate of CorrJoin is not better than the pruning rate of Quickjoin, but in terms of performance, CorrJoin still performs better than all the other algorithms. It is interesting to note that Quickjoin's performance for all three datasets is worse than that of the baseline approach.

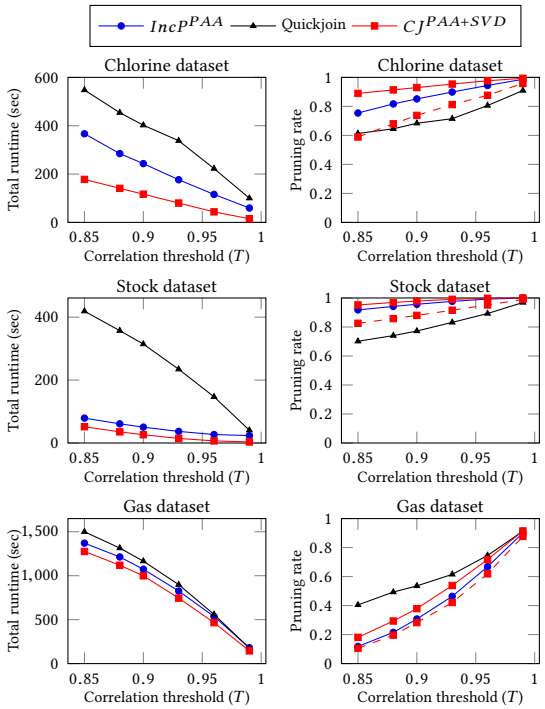

Fig. 12. Total runtime and pruning rate of the approaches for different datasets

Overall, the pruning rate of CorrJoin is better than that of Quickjoin, explaining part of the difference in runtime. Due to the application of dimensionality-reduction techniques, CorrJoin can also improve the runtime of the correlation computation compared to Quickjoin.

6.3.2 Memory Usage. Figure 13 shows the memory usage of the different approaches. $IncP^{PAA}$ uses less memory than our approach and Quickjoin, since it does not reduce the number of comparisons and, thus, does not need additional space for doing so. Nevertheless, CorrJoin only uses around 18 Megabytes more than $IncP^{PAA}$.

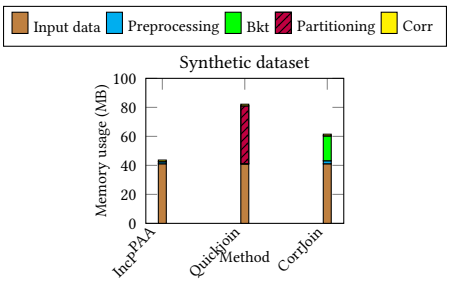

Fig. 13. Memory usage of different approaches

6.3.3 Low Correlation Thresholds. We now investigate the impact of lower correlation rates. Previous work on determining pairwise correlation [9, 30, 35, 42, 57, 58] looked almost exclusively at high or very high correlation thresholds (0.9 or higher). While this covers an important category of applications, there are application domains, such as the social sciences, in which correlated pairs reach a correlation threshold of at most 0.5 or 0.6 [22]. The correlation threshold has a direct impact on the parameter ϵ used for the Euclidean distance: ϵ determines the width of the buckets. The higher the correlation threshold, the lower ϵ, which spreads out the windows among more buckets, resulting in a smaller number of windows found in neighboring buckets. Figure 14 shows the effects of varying the correlation threshold for the different real-world datasets. For the chlorine dataset with a low variability, although CorrJoin also experiences a slowdown and lower join pruning rate of the bucketing filter due to a larger answer set, it is still much faster than the other approaches. For the medium and high variability datasets, stock and gas, CorrJoin still has an edge over the other approaches in terms of performance.

The reason for the performance degradation of the algorithms is the growing size of the answer set for decreasing correlation thresholds, leading to higher costs for generating this set. Even the performance of the baseline approach, which uses a nested-loop strategy, suffers, as more pairs meet the thresholds and have to be checked. For join-based approaches, which avoid computing the Cartesian product, this also has another impact, as the techniques used to filter out non-correlated pairs will pay off less and less. We generated synthetic datasets with different correlation rates, i.e., the proportion of correlated pairs compared to the Cartesian product, to measure the effect of a growing answer set. Figure 15 illustrates the speedup of CorrJoin and Quickjoin compared to the baseline approach. For datasets with low variability (such as the synthetic dataset), in absolute numbers the performance of Quickjoin and $IncP^{PAA}$ deteriorates very similarly when increasing the size of the answer set (cf. top left diagram of Figure 14). For the speedup factor, this means that it will actually increase, since Quickjoin's performance is worse than the baseline approach (which also causes it to stay below one). For CorrJoin, once the rate reaches around 20%, the difference in performance between CorrJoin and $IncP^{PAA}$ is not discernible anymore.

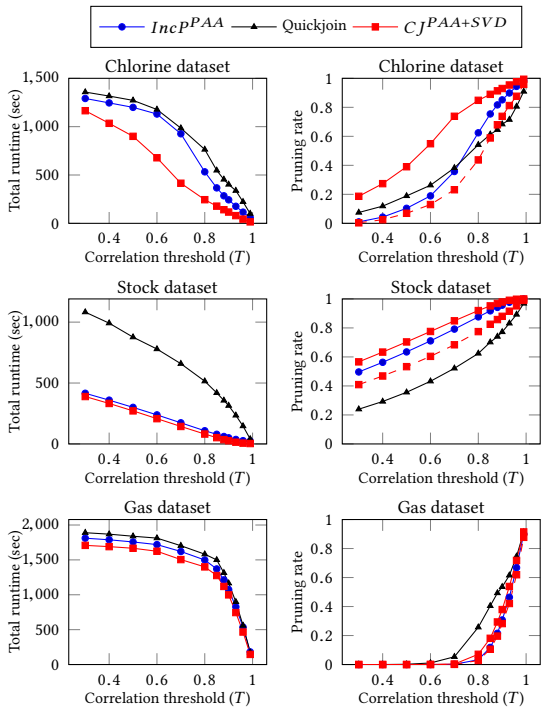

Fig. 14. Total runtime and pruning rate for low correlation thresholds.

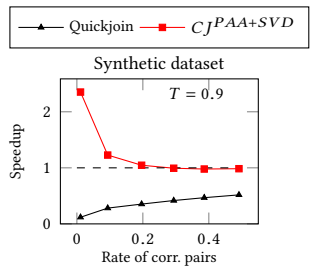

Fig. 15. Speedup for different correlation rates

6.3.4 Stride. One effect of utilizing sliding windows is that we do not have to recompute everything but are able to check for correlation incrementally. However, the impact of this effect is rather small, as updating the mean value vectors of PAA can be done in linear time. In Figure 16a, we compare the runtime of each method for different values of stride h. We report the results for the synthetic dataset (similar results were obtained for the other datasets). The runtime of all methods only increases (very) slightly.

Another, more important, effect of h is the tradeoff between throughput and latency. The larger the value h, the higher the latency (as we have to wait for more values to arrive before processing them) and the better the throughput (because we can processe larger batches). Figure 16b shows the maximum number of series we are able to process for different values of h. For larger strides, we can handle more data streams, but have to wait longer for all h values to arrive.

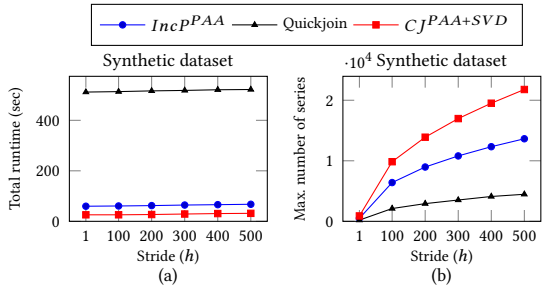

Fig. 16. Runtime and maximum number of series which can be handled by each method when increasing stride h

6.3.5 Number of Time Series. Next, we look at the scalability of the different approaches in terms of the number of processed time series. Figure 17 shows the speedup of CorrJoin and Quickjoin compared to the baseline approach. Quickjoin does not perform better than the baseline approach, which results in a speedup of less than 1. For small values of m, i.e., $m \leq 100$, the speedup of CorrJoin is below one, since the overhead of the join operation (i.e., the preprocessing and partitioning of the input data) does not pay off and $IncP^{PAA}$ is faster. As m increases, i.e., $m > 100$, CorrJoin quickly starts to outperform $IncP^{PAA}$ and eventually levels off, asymptotically approaching a limit. The primary factor determining the speedup of CorrJoin over $IncP^{PAA}$ is that $IncP^{PAA}$ has to compute the Cartesian product, while CorrJoin eliminates a lot of pairs via the bucketing. The other factors are roughly comparable (cf. Section 5). Calculating the exact costs c_{bkt} (as defined in Section 5) is difficult to do analytically. Nevertheless, we can estimate an upper bound for the best possible speedup CorrJoin can hope to achieve. The baseline approach $IncP^{PAA}$ has to compare m^2 pairs. The size of the candidate set produced by CorrJoin is $m^2 r_1$, which means we can lower-bound the number of pairs that were compared in the bucket filter (every pair showing up in the answer set was checked, otherwise it would not have appeared). In turn, this means that the ratio of comparisons between $IncP^{PAA}$ and CorrJoin can be upper-bounded, it is $\frac{m^2}{m^2 r_1} = \frac{1}{r_1}$. Consequently, the speedup of CorrJoin over $IncP^{PAA}$ is at most $\frac{1}{r_1}$. For the setup in Figure 17, the join pruning rate was 0.89, so $\frac{1}{r_1} = \frac{1}{0.11} \approx 9.09$ and the speedup can never go beyond this. It actually leveled off sooner, as this upper bound is quite loose. The insights on scalability re-emphasize the findings depicted in Figure 15. A lower correlation rate among windows leads to a smaller value for r_1, which in turn results in a higher speedup of CorrJoin compared to $IncP^{PAA}$.

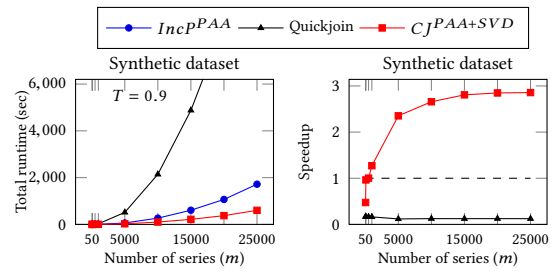

Fig. 17. The benefit of our approach becomes more pronounced for larger m

6.3.6 Window Size. Finally, we consider the impact of the size of the original windows, n, on the performance of the join algorithms (see Figure 18 for an illustration). As the window size increases, the runtime of CorrJoin and Quickjoin goes up. For CorrJoin this does not happen entirely evenly, but accelerates slightly with larger window sizes: for $n = 5000$, CorrJoin has a runtime of 105 seconds, for $n = 10000$, one of 235 seconds. There are multiple causes for the increasing runtime and its unevenness. Clearly, larger windows take more time to process: this concerns the reduction of the dimensionality at the very beginning of the pipeline and the final filter step, which is run on the original windows again. As we can see on the right-hand side of Figure 18, n also has an effect on the pruning rate. Since the dimensionality of the windows in the intermediate steps (k_s, k_e, and k_b) does not change, we are reducing larger and larger windows to the same (small) dimensionalities, which means that we lose more and more accuracy. Then there is also the effect of distance concentration for high-dimensional data, i.e., the higher the dimension, the closer the distance of a point to its closest neighbor becomes to the distance to its farthest neighbor. Thus, the effectiveness of separating windows into different buckets based on their distances from each other diminishes for very high dimensions. In contrast, Quickjoin's runtime increases steeply right from the start, as the window size n directly leads to higher costs for comparing window pairs.

Thus, while the costs for CorrJoin rise with increasing window size, our method is feasible in practice, especially in the range of window sizes typical for applications.

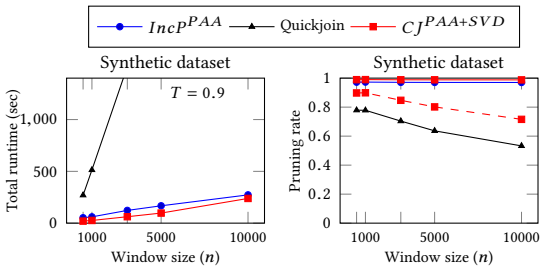

Fig. 18. Total runtime and pruning rate of the approaches with increasing window size n

6.3.7 Improving Quickjoin. Implementing Quickjoin as described by Jacox and Samet [20] leads to an algorithm that spends a substantial amount of its computations for checking distances between window pairs. This becomes apparent when increasing the window size (cf. Figure 18). One way to improve Quickjoin is by leveraging the first part of our approach and reducing the dimensionalities of the windows using PAA and filtering via the dimensionality-reduced windows. In Figure 19, we show the results of comparing the improved Quickjoin (Quickjoin+) with CorrJoin and $IncP^{PAA}$. While applying PAA to Quickjoin certainly boosts its performance it is still clearly outperformed by CorrJoin (see left-hand column of Figure 19). Only for very high correlation thresholds does Quickjoin+ reach the same performance level as CorrJoin. Note that for very high correlation thresholds, algorithms generally show a good performance, i.e., this case is easy to handle. For low correlation thresholds, Quickjoin+ is even slightly slower than $IncP^{PAA}$. When looking at the pruning rate of Quickjoin+ and CorrJoin, we see that the first pruning done by CorrJoin via the bucketing already results in a better rate than Quickjoin+'s partitioning.

As a final improvement, we apply both PAA and SVD to Quickjoin and then run Quickjoin's partitioning on windows whose dimensionality is the same as that of CorrJoin's windows, i.e., the preprocessing now mirrors that of CorrJoin. While this improvement (shown as Quickjoin++ in Figure 20) improves the pruning rate of Quickjoin further and, in turn, its runtime, it is still

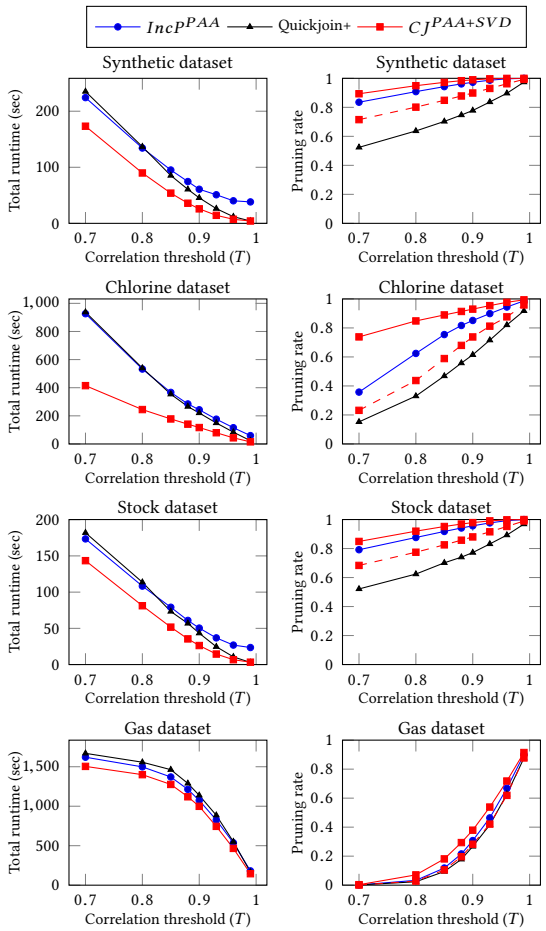

Fig. 19. Total runtime and pruning rate of $IncP^{PAA}$, CorrJoin, and the improved Quickjoin+ (which is still worse than CorrJoin)

slower than CorrJoin. We identified two aspects in which Quickjoin still differs from CorrJoin. First, in contrast to CorrJoin, which can directly assign the dimensionality-reduced windows to buckets, Quickjoin has to compare all windows in the current partition to a pivot. This results in a higher total number of comparisons between windows (see Figure 20). Second, CorrJoin achieves a more accurate partitioning for the filtering: it uses buckets with a width of exactly ϵ. Using a smaller value would mean we could not rule out pairs from non-neighboring buckets, as buckets of a distance of up to two positions could still contain matching pairs. On the other hand, using a larger value would lead to checking additional pairs in wider neighboring buckets that could have been ruled out with narrow buckets. Since Quickjoin partitions the data on the fly with pivots, it is highly unlikely that it ends up with partitions that have a width of exactly ϵ. Therefore, compared to CorrJoin, it has to check additional pairs.

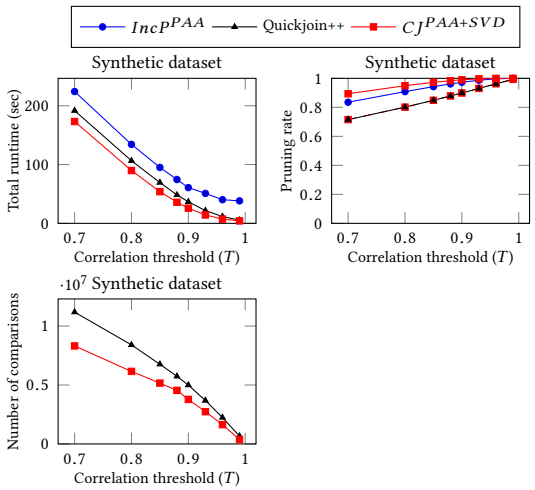

Fig. 20. Total runtime and pruning rate of $IncP^{PAA}$, CorrJoin, and Quickjoin++ (employing both PAA and SVD)

7 CONCLUSION AND FUTURE WORK

We propose *CorrJoin*, an efficient and effective algorithm for determining all pairs of data stream windows whose correlation is above a certain threshold. We carefully combine PAA, SVD, bucketing, and Euclidean distance filters to a novel and powerful reduce-filter-refine algorithm that leverages the complementary properties of dimension reduction and transformation to build a highly effective bucketing filter that outperforms state-of-the-art solution for this task. We illustrate its effectiveness and efficiency in an empirical evaluation using several real-world datasets, showing that CorrJoin outperforms state-of-the-art algorithms. It reduces the number of needed comparisons by at least 80% and is several times faster than state-of-the-art algorithms and the baseline approach. Additionally, we exhibit how parts of our technique can even be applied to Quickjoin, one of our competitors, to boost its performance. While it is still (slightly) slower than our algorithm, it can now consistently outperform the baseline. This demonstrates that our approach has the potential to be applicable on a more general level.

For future work, we plan to investigate the following aspects. The bucketing approach makes it possible to adapt our algorithm to distributed processing, for instance executing it in a map-reduce framework. The preprocessing involving PAA and SVD can be done independently for each window in the map step, while the bucketing determines in the shuffle step which reducers receive which potential window pairs to check for correlation. We also believe that our approach is applicable to high-dimensional similarity joins using dimensionality reduction techniques that lower-bound the true distance and plan to take a closer look at this setting.

REFERENCES

[1] Pankaj K. Agarwal, Xiao Hu, Stavros Sintos, and Jun Yang. 2021. Dynamic Enumeration of Similarity Joins. In *48th Int. Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference) (LIPIcs).* https://doi.org/10.4230/LIPIcs.ICALP.2021.11

[2] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. 1993. Efficient Similarity Search In Sequence Databases. In *Proceedings of the FODO Conference.* 69–84.

[3] Dror Aiger, Haim Kaplan, and Micha Sharir. 2014. Reporting Neighbors in High-Dimensional Euclidean Space. *SIAM J. Comput.* 43, 4 (jan 2014), 1363–1395. https://doi.org/10.1137/12089867X

[4] Paul Bottinelli and Joppe Bos. 2016. Computational aspects of correlation power analysis. *Journal of Cryptographic Engineering* 7 (February 2016). https://doi.org/10.1007/s13389-016-0122-9

[5] Yuhan Cai and Raymond Ng. 2004. Indexing Spatio-Temporal Trajectories with Chebyshev Polynomials. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 599–610. https://doi.org/10.1145/1007568.1007636

[6] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. 2014. Beyond one billion time series: Indexing and mining very large time series collections with iSAX2+. *Knowledge and Information Systems* (2014).

[7] Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. 2002. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM Trans. Database Syst.* 27, 2 (June 2002), 188–228. https://doi.org/10.1145/568518.568520

[8] Qiuxia Chen, Lei Chen, Xiang Lian, Yunhao Liu, and Jeffrey Xu Yu. 2007. Indexable PLA for Efficient Similarity Search. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 435–446.

[9] Richard Cole, Dennis Shasha, and Xiaojian Zhao. 2005. Fast Window Correlations over Uncooperative Time Series. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*. Association for Computing Machinery, New York, NY, USA, 743–749. https://doi.org/10.1145/1081870.1081966

[10] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. ClusterJoin: A Similarity Joins Framework Using Map-Reduce. *Proc. VLDB Endow.* (aug 2014), 1059–1070.

[11] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. 2008. Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures. *Proc. VLDB Endow.* (aug 2008), 1542–1552.

[12] Karima Echihabi. 2019. Truly Scalable Data Series Similarity Search. In *Proceedings of the VLDB 2019 PhD Workshop, co-located with the 45th International Conference on Very Large Databases (VLDB 2019), Los Angeles, California, USA, August 26-30, 2019 (CEUR Workshop Proceedings)*. CEUR-WS.org.

[13] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *Proc. VLDB Endow.* (oct 2018), 112–127.

[14] C. Faloutsos, H. Jagadish, A. Mendelzon, and T. Milo. 1997. A Signature Technique for Similarity-Based Queries. In *Proceedings of the Compression and Complexity of Sequences 1997 (SEQUENCES '97)*. IEEE Computer Society, USA.

[15] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. 1994. Fast Subsequence Matching in Time-Series Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 419–429. https://doi.org/10.1145/191839.191925

[16] Like Gao and X. Sean Wang. 2002. Continually Evaluating Similarity-Based Pattern Queries on a Streaming Time Series. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 370–381. https://doi.org/10.1145/564691.564734

[17] Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Anastasia Bezerianos, and Themis Palpanas. 2020. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, 1857–1873.

[18] Tian Guo, Jean-Paul Calbimonte, H. Zhuang, and Karl Aberer. 2015. SigCO: Mining Significant Correlations via a Distributed Real-time Computation Engine. In *2015 IEEE International Conference on Big Data, Big Data 2015*. 747–756.

[19] Alfred Haar. [n. d.]. On the Theory of Orthogonal Function Systems. https://pdfs.semanticscholar.org/3b08/b61ba914626db518b6add5b73ac21d62f0c1.pdf.

[20] Edwin H. Jacox and Hanan Samet. 2008. Metric Space Similarity Joins. *ACM Trans. Database Syst.* (jun 2008), 38 pages.

[21] Li Junkui and Wang Yuanzhen. 2007. APCAS: An Approximate Approach to Adaptively Segment Time Series Stream. In *Proceedings of the Joint 9th Asia-Pacific Web and 8th International Conference on Web-Age Information Management Conference on Advances in Data and Web Management* (Huang Shan, China) *(APWeb/WAIM'07)*. Springer-Verlag, Berlin, Heidelberg, 554–565.

[22] Daniel Kahnemann, Olivier Sibony, and Cass Sunstein. 2021. *Noise: A Flaw in Human Judgment.* Little, Brown Spark, New York.

[23] E. Keogh. 1997. Fast similarity search in the presence of longitudinal scaling in time series databases. In *Proceedings Ninth IEEE International Conference on Tools with Artificial Intelligence*. 578–584.

[24] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and Information Systems* 3 (August 2001), 263–286.

[25] Mourad Khayati. 2015. *Recovery of Missing Values using Matrix Decomposition Techniques.* Ph. D. Dissertation. University of Zurich.

[26] Kin-Pong Chan and Ada Wai-Chee Fu. 1999. Efficient Time Series Matching by Wavelets. In *Proceedings of the 15th International Conference on Data Engineering (ICDE '99)*. IEEE Computer Society, USA, 126.

[27] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: a scalable bottom-up approach for building data series indexes. *Proceedings of the VLDB Endowment* (02 2018), 677–690.

[28] Flip Korn, H. V. Jagadish, and Christos Faloutsos. 1997. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (Tucson, Arizona, USA) *(SIGMOD '97)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/253260.253332

[29] Chung-Sheng Li, Philip S. Yu, and Vittorio Castelli. 1996. HierarchyScan: A Hierarchical Similarity Search Algorithm for Databases of Long Sequences. In *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE Computer Society, 546–553. https://doi.org/10.1109/ICDE.1996.492205

[30] Yuhong Li, Leong Hou U, Man Lung Yiu, and Zhiguo Gong. 2013. Discovering Longest-Lasting Correlation in Sequence Databases. *Proc. VLDB Endow.* 6, 14 (September 2013), 1666–1677. https://doi.org/10.14778/2556549.2556552

[31] X. Lian and L. Chen. 2008. Efficient Similarity Search over Future Stream Time Series. *IEEE Transactions on Knowledge & Data Engineering* 20, 01 (January 2008), 40–54. https://doi.org/10.1109/TKDE.2007.190666

[32] Xiang Lian, Lei Chen, Jeffrey Xu Yu, Jinsong Han, and Jian Ma. 2009. Multiscale Representations for Fast Pattern Matching in Stream Time Series. *IEEE Trans. on Knowl. and Data Eng.* 21, 4 (April 2009), 568–581. https://doi.org/10.1109/TKDE.2008.184

[33] Michele Linardi and Themis Palpanas. 2018. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *Proc. VLDB Endow.* (sep 2018), 2236–2248.

[34] Vasileios Megalooikonomou, Guo Li, and Qiang Wang. 2004. A Dimensionality Reduction Technique for Efficient Similarity Analysis of Time Series Databases *(CIKM '04)*. Association for Computing Machinery, New York, NY, USA, 160–161. https://doi.org/10.1145/1031171.1031203

[35] Abdullah Mueen, Hossein Hamooni, and Trilce Estrada. 2014. Time Series Join on Subsequence Correlation. In *2014 IEEE International Conference on Data Mining*. 450–459. https://doi.org/10.1109/ICDM.2014.52

[36] Abdullah Mueen, Suman Nath, and Jie Liu. 2010. Fast approximate correlation for massive time-series data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 171–182. https://doi.org/10.1145/1807167.1807188

[37] Sina Niedermaier, Falko Koetter, Andreas Freymann, and Stefan Wagner. 2019. On Observability and Monitoring of Distributed Systems – An Industry Interview Study. In *17th Int. Conf. on Service-Oriented Computing (ICSOC'19)*. Toulouse, France, 36–52.

[38] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. 2005. Streaming Pattern Discovery in Multiple Time-Series. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 697–708.

[39] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. MESSI: In-Memory Data Series Indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 337–348.

[40] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. ParIS+: Data Series Indexing on Multi-Core Architectures. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2021), 2151–2164.

[41] I. Popivanov and R.J. Miller. 2002. Similarity search over time-series data using wavelets. In *Proceedings 18th International Conference on Data Engineering*. 212–221. https://doi.org/10.1109/ICDE.2002.994711

[42] Han Qiu, Hoang Thanh Lam, Francesco Fusco, and Mathieu Sinn. 2018. Learning Correlation Space for Time Series. https://arxiv.org/pdf/1802.03628.pdf. arXiv:1802.03628 [cs.LG]

[43] Davood Rafiei. 1999. On similarity-based queries for time series data. *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)* (1999), 410–417.

[44] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (jul 2010), 65–79. https://doi.org/10.1109/MM.2010.68

[45] Irene Rodriguez-Lujan, Jordi Fonollosa, Alexander Vergara, M.L. Homer, and Ramón Huerta. 2013. On the calibration of sensor arrays for pattern recognition using the minimal number of experiments. *Chemometrics and Intelligent Laboratory Systems* 130 (January 2013). https://doi.org/10.1016/j.chemolab.2013.10.012

[46] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. 1997. High-Dimensional Similarity Joins. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE '97)*. IEEE Computer Society, 301–311.

[47] A. V. Oppenheim und R. W. Schafer. 1975. *Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey.

[48] Alexander Vergara, Shankar Vembu, Tuba Ayhan, Margaret A. Ryan, Margie L. Homer, and Ramón Huerta. 2012. Chemical gas sensor drift compensation using classifier ensembles. *Sensors and Actuators B: Chemical* 166-167 (2012), 320–329. https://doi.org/10.1016/j.snb.2012.01.074

[49] Qiang Wang and Vasileios Megalooikonomou. 2008. A Dimensionality Reduction Technique for Efficient Time Series Similarity Analysis. *Information systems* 33, 1 (March 2008), 115–132. https://doi.org/10.1016/j.is.2007.07.002

[50] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-Adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *Proc. VLDB Endow.* (aug 2013), 793–804.

[51] Daniel Wu, Ambuj Singh, Divyakant Agrawal, Amr El Abbadi, and Terence R. Smith. 1996. Efficient Retrieval for Browsing Large Image Databases. In *Proceedings of the Fifth International Conference on Information and Knowledge Management* (Rockville, Maryland, USA) *(CIKM '96)*. Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/238355.238365

[52] Qing Xie, Shuo Shang, Bo Yuan, Chaoyi Pang, and Xiangliang Zhang. 2013. Local Correlation Detection with Linearity Enhancement in Streaming Data. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management (CIKM '13)*. Association for Computing Machinery, 309–318.

[53] Yunlong Xu, Jinshu Liu, and Fatemeh Nargesian. 2022. TSUBASA: Climate Network Construction on Historical and Real-Time Data. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 286–295.

[54] Djamel Edine Yagoubi, Reza Akbarinia, Boyan Kolev, Oleksandra Levchenko, Florent Masseglia, Patrick Valduriez, and Dennis Shasha. 2018. ParCorr: efficient parallel methods to identify similar time series pairs across sliding windows. *Data Mining and Knowledge Discovery* (2018). https://doi.org/10.1007/s10618-018-0580-z

[55] B. . Yi, N. D. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. 2000. Online data mining for co-evolving time sequences. In *Proceedings of 16th International Conference on Data Engineering*. 13–22.

[56] Byoung-Kee Yi and Christos Faloutsos. 2000. Fast Time Sequence Indexing for Arbitrary Lp Norms. *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00* (January 2000), 385–394.

[57] Sheng Zhong, Vinicius M.A. Souza, and Abdullah Mueen. 2020. FilCorr: Filtered and Lagged Correlation on Streaming Time Series. In *2020 IEEE International Conference on Data Mining (ICDM)*. 1436–1441. https://doi.org/10.1109/ICDM50108.2020.00190

[58] Yunyue Zhu and Dennis E. Shasha. 2002. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. 358–369. https://doi.org/10.1016/B978-155860869-6/50039-1

[59] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: The Adaptive Data Series Index. *The VLDB Journal* (dec 2016), 843–866.

Received April 2023; revised July 2023; accepted August 2023