Scalable Hybrid Similarity Join over Geolocated Time Series

Georgios Chatzigeorgakidis
University of Peloponnese, Greece
chgeorgakidis@uop.gr

Kostas Patroumpas
IMSI, Athena R.C., Greece
kpatro@imis.athena-innovation.gr

Dimitrios Skoutas
IMSI, Athena R.C., Greece
dskoutas@imis.athena-innovation.gr

Spiros Athanasiou
IMSI, Athena R.C., Greece
spathan@imis.athena-innovation.gr

Spiros Skiadopoulos
University of Peloponnese, Greece
spiros@uop.gr

ABSTRACT

A geolocated time series is a sequence of values associated with a geolocation, such as measurements provided by a sensor installed at a certain location. In this paper, we address the problem of *hybrid similarity joins* over such geolocated time series. This operation returns all pairs of geolocated time series that exhibit similar behavior in the time series domain while also being closely located in space. First, we propose algorithms for performing such join operations using different types of indices, including spatial-only, time series-only, and hybrid indices. Such centralized indexing schemes can cope well with moderate data volumes but they face scalability issues when the dataset size increases significantly. To overcome this problem, we present a MapReduce-based processing scheme with space-driven partitioning. Our parallel and distributed algorithm leverages our hybrid index for geolocated time series to efficiently execute similarity joins locally within each partition and minimize the amount of data that needs to be shuffled between processing nodes. An extensive experimental evaluation confirms that our approach can efficiently compute all matching pairs even for datasets containing millions of geolocated time series.

CCS CONCEPTS

• **Information systems → Data structures**; **Spatial-temporal systems**; • **Theory of computation → Distributed algorithms**; **MapReduce algorithms**;

KEYWORDS

time series, big data, spatial indices, similarity joins

ACM Reference Format:
Georgios Chatzigeorgakidis, Kostas Patroumpas, Dimitrios Skoutas, Spiros Athanasiou, and Spiros Skiadopoulos. 2018. Scalable Hybrid Similarity Join over Geolocated Time Series. In *26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '18), November 6–9, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3274895.3274949

1 INTRODUCTION

Time series data is a treasure trove for a variety of mining and monitoring applications both in industry (e.g., finance, public utilities) and in academia (e.g., astronomy, biology), while a rapidly increasing bulk of such data is also generated on the Web and the Internet of Things. Although indexing, analysis and exploration of time series data has attracted a lot of interest from the database and data mining communities [5, 8, 18], studying of *geolocated time series* only lately has come under focus [7]. This refers to time series that are produced at, or associated with, a specific geolocation. Analyzing such data can offer insights regarding trends and patterns in many applications. Indeed, they are often used to identify user check-in patterns in geosocial networks, weather or pollution measurements from a sensor network, resource consumption in households, fluctuations of house prices in real estate, and so on.

In this work, we focus on efficient evaluation of *hybrid similarity join queries* between large datasets of geolocated time series. Consider two such datasets containing time series of CO_2 emissions collected from two sensor networks R and S spread in different locations over a given spatial region. A hybrid similarity join query retrieves pairs of sensors (the first from R, the second from S) such that both the distance between the locations of the two sensors and the distance between the time series of their measurements do not exceed certain given thresholds. Then, an environmentalist may use the matching pairs to identify common patterns in nearby areas and get a better insight about the sources of pollution, its spread, etc. Similarly, check-ins in geosocial networks can also be modeled as geolocated time series and analyzed with hybrid similarity join queries. Results can indicate nearby venues with similar frequency patterns, which may be used for social recommendations according to time, place, activity, etc. Moreover, geolocated time series can indicate water or gas consumption in households. A utility company may identify nearby customers who have similar consumption profiles. Results may be used for customer segmentation, targeted marketing, planning future network upgrades, etc.

A hybrid similarity join query aims to identify *all pairs* between the two datasets qualifying to the criteria of *spatial proximity* and *time series similarity*. Clearly, performing a pairwise comparison among all pairs of objects in the two datasets is not an option when their size is large. Hence, *indexing* them is indispensable for efficient processing of such queries. Certainly, similarity search over indexed time series is a well-studied topic and several schemes have been proposed, like wavelet-based methods [6] or the family of *i*SAX trees [4, 5, 18, 21]. Likewise, efficient methods for distance joins in spatial databases also exist, usually over R-trees [3, 14].

In this paper, our starting point is to employ such indices either for *time series-only* (with *i*SAX) or *spatial-only* (using R-trees) filtering of candidate pairs during query evaluation. We also take advantage of the BTSR-tree index [7], which enables *combined search* over both the time series and the spatial information of candidates and thus excels in pruning power. These algorithms concurrently traverse those indices and identify subtrees that may contain candidate matches. However, this *centralized* approach has certain limitations, as it cannot sustain examination of large datasets. Hence, we further suggest a space-driven data partitioning scheme that enables a *parallel and distributed* approach for hybrid similarity joins. Following the MapReduce paradigm, our method leverages any of the aforementioned indices to efficiently handle similarity join queries locally within each partition. This is then combined with an optimization that minimizes the amount of data transferred between worker nodes at query time without false misses.

To the best of our knowledge, this is the first work to address hybrid similarity join queries over large datasets of geolocated time series. Our main contributions can be summarized as follows:

- We adapt state-of-the-art indices over such data in centralized settings and propose traversal methods that can prune the search space and return answers without false misses.
- We suggest a space-driven partitioning method to distribute large datasets in cluster infrastructures, thus enabling faster, in-parallel evaluation of smaller similarity join tasks.
- We conduct an extensive experimental evaluation against large data volumes of geolocated time series, confirming that our methods can efficiently and correctly process hybrid similarity join queries in these settings.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the problem. Section 4 provides background on two state-of-the-art index methods that we employ in alternative similarity join strategies presented in Section 5. Section 6 introduces a parallel and distributed approach for similarity join over large datasets of geolocated time series. Section 7 reports our experimental results, and Section 8 concludes the paper.

2 RELATED WORK

Earlier approaches on *time series indexing* have leveraged multi-resolution representations to gradually reduce time series dimensionality with Discrete Wavelet Transform and then index the resulting coefficients [6, 15]. Current state-of-the-art indexing over time series involves the *indexable Symbolic Aggregate Approximation* (*i*SAX) family of trees, which are based on the *Symbolic Aggregate Approximation* (SAX) representation of each time series [12]. After the original *i*SAX tree introduced in [18], several extensions have been proposed, including *i*SAX 2.0 [4] and *i*SAX2+ [5], which enable bulk loading of time series data and better handle the expensive I/O operations caused by aggressive node splitting during index construction. The ADS+ index [21] is an adaptive approach of *i*SAX, built progressively while processing query workloads, thus sparing much of the initial construction overhead. A comprehensive overview over time series indexing schemes based on the SAX representation is available in [13]. However, efficiently accommodating spatial information in any such scheme is not straightforward.

With respect to *spatial join queries*, several methods have been proposed, often based on the R-tree family of indices [1, 10]. In particular, the spatial join algorithms over R*-trees introduced in [3] can minimize the CPU and I/O cost in searching. *Multiway* spatial joins [14] generalize search over more than two R-trees. Top-k spatial distance joins [16] employ R-tree-based spatial joins in data blocks ordered by an objective score to retrieve k pairs of objects with highest score. However, all such algorithms are applied against spatial information only. Based on a similar observation for answering a variety of queries over geolocated time series, in [7] we proposed the BTSR-tree, a hybrid index based on the R-tree, but having nodes that also store bounds over the time series information in their underlying subtree. This index offers increased pruning capabilities for queries involving both time series similarity and spatial proximity. However, handling hybrid similarity joins is not addressed in [7]; we develop such a method next in this paper.

Our current work on geolocated time series data is reminiscent of related approaches in *spatio-textual search*. Spatio-textual joins identify objects that are both spatially and textually close. In particular, the algorithm proposed in [2] uses a spatial partitioning in conjunction with spatial joins over R-trees in order to batch process such queries. MapReduce-based methods in [20] resolve spatio-textual joins on spatially partitioned data. However, it should be stressed that time series information is quite distinct from documents or keywords used in those works and certainly requires a totally different processing paradigm. To the best of our knowledge, ours is the first approach for processing similarity joins on geolocated time series data.

3 PROBLEM DEFINITION

A *time series* is a time-ordered sequence of values $T = \{v_1, \ldots, v_n\}$, where v_i is the value at the i-th time point and n is the length of the series. Typically, as absolute values are usually less informative compared to the trends in a sequence, a *z-normalization* of the amplitudes is applied [9], so that the transformed time series approximately follow a standard Gaussian distribution $\mathcal{N}(0,1)$. We assume that this is done in a preprocessing step, applied similarly over all time series in a given dataset $\mathcal{T}$. As in other prior works like [18], we use the Euclidean distance to measure the *similarity* between a pair of time series T and T' of equal length n:

$$d_{ts}(T,T') = \sqrt{\sum_{i=1}^{n}(T.v_i - T'.v_i)^2}. \quad (1)$$

In this work, we specifically deal with time series that are additionally characterized by a *location*, denoted by $T.loc$. In the sequel, when it is clear from the context, we also refer to such geolocated time series as objects for brevity. Assuming a 2-dimensional space, we further use the notation $T.loc_x$, $T.loc_y$ to refer to the (x,y) coordinates of T's location. In the *spatial domain*, the *proximity* between two geolocated time series T and T' is calculated using the Euclidean distance of their respective locations:

$$d_{sp}(T,T') = \sqrt{(T.loc_x - T'.loc_x)^2 + (T.loc_y - T'.loc_y)^2}. \quad (2)$$

We can now formally introduce our problem:

Definition 1 (Hybrid Similarity Join over Geolocated Time Series). *Given two sets of geolocated times series $\mathcal{T}_R$ and $\mathcal{T}_S$, and two*

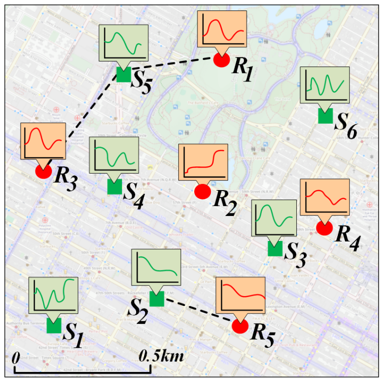


Figure 1: Hybrid similarity join over geolocated time series.

thresholds ϵ_{sp} *and* ϵ_{ts}*, the* hybrid similarity join *query returns all pairs qualifying w.r.t. to both criteria on spatial proximity and time series similarity, i.e.,*

$$\{(T_R, T_S) : T_R \in \mathcal{T}_R, T_S \in \mathcal{T}_S, d_{sp}(T_R, T_S) \leq \epsilon_{sp} \wedge d_{ts}(T_R, T_S) \leq \epsilon_{ts}\}.$$

That is, this query searches for pairs of objects that are within spatial distance at most ϵ_{sp}, while also their respective time series do not deviate by more than ϵ_{ts}. Spatial proximity is measured in distance units (e.g., meters). As mentioned before, since the (transformed) time series are z-normalized, values for parameter ϵ_{ts} are *unitless* and are typically expressed in standard deviations.

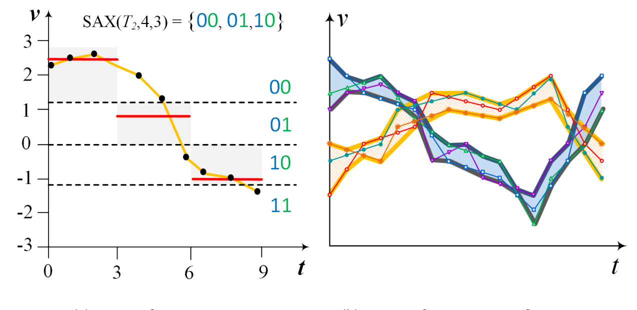
Example 1. *Figure 1 depicts two sets of geolocated time series,* $\{R_1, \ldots, R_5\}$ *(in red bullets) and* $\{S_1, \ldots, S_6\}$ *(in green squares) that represent* CO_2 *emissions collected by two sensor networks R and S in an urban area during a day. Suppose that a similarity join query over those two datasets specifies a distance radius* $\epsilon_{sp} = 500$ *meters to identify nearby sensors and a maximum deviation of* $\epsilon_{ts} = 0.4$ *to find similar* CO_2 *patterns. Qualifying pairs* $\{(R_1, S_5), (R_3, S_5), (R_5, S_2)\}$ *are shown connected with dashed lines. Note that other pairs, e.g.,* (R_4, S_3)*, may be even closer in space, but their time series deviate more than the given* ϵ_{ts}*, so they are filtered out. Besides, time series like those in rejected pair* (R_3, S_3) *may have almost the same pattern, but their locations are too far from each other to qualify for this query.* □

4 PRELIMINARIES

Next, we provide some background information used in our proposed methods for similarity joins over geolocated time series. Specifically, we present the *i*SAX family of trees [4, 5, 18], which can only index the *time series* information of each object and the *hybrid* BTSR-tree [7]. The latter is essentially an R-tree built on the spatial locations of the times series, but additionally summarizing in each node the time series contained in its subtree.

4.1 SAX Representation of Time Series

The *Symbolic Aggregate approXimation (SAX)* is a multi-resolution representation of a time series introduced in [18]. It can be derived from its *Piecewise Aggregate Approximation* (PAA) [11, 19] by quantizing the PAA segments on the v-axis. As exemplified in Figure 2(a), a time series T_2 is transformed to a PAA representation of w=3 words with real-valued coefficients (the horizontal red bars). To get a *SAX* representation for a time series, these coefficients are discretized along the v-axis using *breakpoints* (shown with dashed lines) assuming a $\mathcal{N}(0,1)$ Gaussian distribution that enables generation of equi-probable symbols for a given cardinality ($b = 4$ symbols are used in this example). Interestingly, by using bitwise representations for these symbols, coarser *SAX* values can be obtained from more refined ones by simply ignoring trailing bits. Importantly, the Euclidean distance between *SAX* representations of two time series is guaranteed to be a *lower bound* with respect to the Euclidean distance over the original time series. Formally, for two time series T, T' of equal length n using their respective *SAX* words T_w, T'_w of size w, it holds that:



(a) SAX of a time series (b) MBTS for two sets of time series

Figure 2: SAX and MBTS representations over time series.

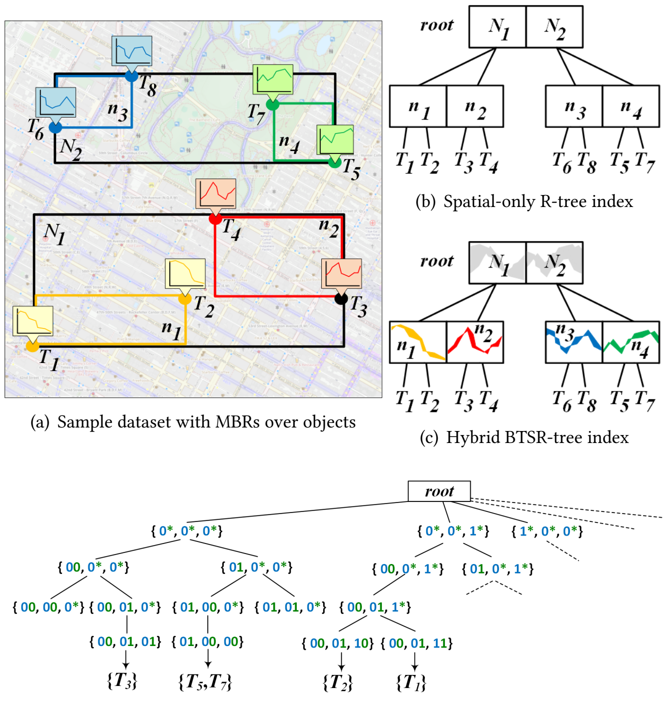
$$d_{SAX}(T_w, T'_w) = \sqrt{\frac{n}{w}} \sqrt{\sum_{j=1}^{w} d^2(t_j, t'_j)} \leq \sqrt{\sum_{i=1}^{n} (T.v_i - T'.v_i)^2} \tag{3}$$

where $d(t_j, t'_j)$ is the distance between symbols at the j-th position of each *SAX* word. Comparing *i*SAX words of different cardinality is possible by promoting the *i*SAX representation of lower cardinality to that of the larger, as the lower bound in Eq. 3 still holds.

4.2 The *i*SAX Family of Indices

Consider the dataset shown in Figure 3(a). By completely ignoring the spatial locations and using the SAX representations of all time series in this dataset, an *i*SAX index [18] can be built as illustrated in Figure 3(d). The root node captures the complete *i*SAX space. It does not contain any SAX words, it only points to its children nodes (in the worst case, their number is $2w$). Each leaf has a pointer to a disk file containing the raw time series that it represents. The leaf itself also stores the *i*SAX word of highest cardinality among these time series. An internal node designates a split in SAX space and is created when the number of time series contained by a leaf node exceeds a fixed capacity M. This split is binary and is made at a given position $j = 1..w$ of the SAX word using a round-robin policy, so it always yields two children that differ on their j-th symbol while replicating the rest from their parent node. In essence, the SAX space represented by every node fully contains the union of the SAX spaces of its subtree.

Searching for time series similar to a query q simply traverses the *i*SAX tree, looking for a leaf node having the same *i*SAX word as query q. The respective raw time series are fetched from disk and a sequential scan identifies those matching with q.



(a) Sample dataset with MBRs over objects

(b) Spatial-only R-tree index

(c) Hybrid BTSR-tree index

(d) *i*SAX index over time series only (subtrees under dash lines not shown)

Figure 3: Indexing schemes over geolocated time series.

4.3 Minimum Bounding Time Series

In [7], we introduced the notion of *Minimum Bounding Time Series* (MBTS), which abstracts *a set of time series* $\mathcal{T}$ using a pair of bounds that fully contain all of them. Figure 2(b) depicts an example of two MBTSs for two disjoint sets of time series. Formally, given a set of time series $\mathcal{T}$, its MBTS consists of an *upper bounding time series* $B^{\sqcap}$ and a *lower bounding time series* $B^{\sqcup}$, constructed by respectively selecting the maximum and minimum of values at each time point $i \in \{1, \ldots, n\}$ among all time series in set $\mathcal{T}$ as follows:

$$\begin{aligned} B^{\sqcap} &= \{\max_{T \in \mathcal{T}} T.v_1, \ldots, \max_{T \in \mathcal{T}} T.v_n\} \\ B^{\sqcup} &= \{\min_{T \in \mathcal{T}} T.v_1, \ldots, \min_{T \in \mathcal{T}} T.v_n\} \end{aligned} \tag{4}$$

Note that both bounding time series have the same length n as those enclosed within this MBTS.

4.4 The BTSR-tree Index

A BTSR-tree is constructed exactly as an R-tree [10] with respect to the spatial contents of a geolocated time series dataset, as depicted in the example of Figure 3. However, besides MBRs, nodes also store MBTSs, shown as colored strips per node in Figure 3(c). Thus, the search space can be efficiently pruned when evaluating hybrid queries combining time series similarity with spatial proximity.

As in R-trees, each node of the BTSR-tree has at least m and at most M entries and stores the MBRs of its children. Additionally, for each child, a node stores a pre-specified number of MBTSs, each one enclosing all the time series indexed in its subtree. Each MBTS is calculated according to Eq. 4. Construction and maintenance of the BTSR-tree follow the procedures of the R-tree for data insertion, deletion and node splitting. Objects (i.e., geolocated time series) are inserted into leaf nodes and any resulting changes are propagated upwards. Once the nodes have been populated, the MBTS of each node are calculated bottom-up, relying on k*-means clustering* according to their Euclidean distance in the time series domain. The example in Figure 2(b) depicts the $k = 2$ MBTSs (as two bands with a thick outline) obtained for a set of time series (shown as thin polylines). In a BTSR-tree, each parent node receives all the MBTSs of its children and computes its own k MBTSs. The process continues upwards, until reaching the root.

5 CENTRALIZED SIMILARITY JOIN

A naïve approach to answer a hybrid similarity join query over two geolocated time series $\mathcal{T}_R, \mathcal{T}_S$ would involve examination of all possible pairs, i.e., calculating their Cartesian product $\mathcal{T}_R \times \mathcal{T}_S$ and filtering each candidate pair with the two criteria. Clearly, such a technique has limitations due to its quadratic processing cost and cannot be realistically applied against datasets with more than a few thousand objects each. Hence, we propose three *index-based* techniques for answering hybrid similarity join queries:

- We describe a *spatial-only* filtering method that employs R-trees over the locations of objects so as to identify candidate pairs close enough in space. Afterwards, the time series of each such candidate pair should also be checked on their similarity to finally yield the exact answer (Section 5.1).
- We build *i*SAX indices *over the time series* information only per dataset and we introduce a traversal method to facilitate similarity search. Refinement over returned candidate pairs by their spatial distance issues the final results (Section 5.2).
- We employ BTSR-trees that can *jointly* index the positional and time series information of each object. We introduce a *hybrid similarity join* algorithm that descends these two BTSR-trees in tandem and can safely prune subtrees that cannot possibly contribute any valid results (Section 5.3).

In each of these methods, one global index is created per dataset. Hence, a *centralized* processor is responsible to maintain these indices and access them when evaluating similarity join queries.

5.1 Spatial-Only Filtering using R-Trees

One possible approach to similarity join search over two datasets $\mathcal{T}_R, \mathcal{T}_S$ of geolocated time series is to build an R-tree [10] per dataset by organizing its spatial locations into a hierarchy of nested d-dimensional rectangles. Each node corresponds to a disk page and represents the MBR of its children. A leaf holds the MBR of its contained geometries. The number of entries per node (excluding the root) is between a lower bound m and a maximum capacity M.

With respect to *hybrid similarity joins*, we search over R-trees using the spatial condition, exactly as in [3]. So, both R-trees are concurrently traversed starting from their roots and recursively examining their respective descendants only if the minimum distance *MINDIST* of their MBRs [17] does not exceed parameter ϵ_{sp}. Obviously, a pair of nodes breaking this spatial constraint cannot possibly contain any qualifying results, so their respective subtrees can be safely pruned. Once the leaf levels are reached, the candidate pairs of raw time series are accessed and refined according to both criteria in Definition 1 in order to issue the final results.

5.2 Time Series-Only Filtering with iSAX

This method makes use of available *i*SAX indices over two datasets $\mathcal{T}_R, \mathcal{T}_S$, each concerning their respective time series as discussed in Section 4.2. Each *i*SAX index solely indexes the time series part of a dataset; their leaf entries point to the raw time series including their location. Searching for similar objects starts from the root of each tree and progressively descends by visiting nodes that may contain candidate answers based on their similarity, strictly on the time series domain, as listed in Algorithm 1. Without loss of generality, we assume that either the trees have the same height, or the *i*SAX for $\mathcal{T}_R$ is less deep than the *i*SAX for $\mathcal{T}_S$.

Suppose that at a given iteration, node R in the first *i*SAX needs to be checked against node S in the second *i*SAX. If neither of them is leaf, the algorithm is recursively called against all combinations of their children entries, provided that these are within distance ϵ_{ts} as computed by Eq. 1 (Lines 1-5). Once the leaf level is reached in the first *i*SAX but not yet in the second *i*SAX (if they differ in height), recursive calls examine that specific leaf of the former against each of the children entries of the latter (Lines 6-8).

Eventually, when the leaf level is reached in both trees, we compare each combination of their respective contents (Lines 9-13). Each geolocated time series from leaf R of the first *i*SAX is checked with its counterparts in leaf S of the second *i*SAX. Since raw time series data is fully accessible at the leaf level (including locations), refinement of candidates is based not only on their distance d_{ts} in the time series domain, but also on their spatial distance d_{sp}. If both distances are below the respective constraints, then this specific pair qualifies for the final result Q to the query. The algorithm terminates once there are no remaining pairs of leaves to check.

Algorithm 1: *SimJoinSAX*$(R, S, \epsilon_{sp}, \epsilon_{ts})$

```
Input: Nodes R, S, spatial constraint $\epsilon_{sp}$, time series constraint $\epsilon_{ts}$
Output: Set Q of pairs of geolocated time series satisfying constraints
1  if R is not leaf ∧ S is not leaf then          ▹ internal nodes in both trees
2      foreach $N_R \in R.getChildren()$ do
3          foreach $N_S \in S.getChildren()$ do
4              if $d_{SAX}(N_R, N_S) \leq \epsilon_{ts}$ then      ▹ compare SAX words
5                  $SimJoinSAX(N_R, N_S, \epsilon_{sp}, \epsilon_{ts})$
6  else if R is leaf ∧ S is not leaf then         ▹ trees of different height
7      foreach $N_S \in S.getChildren()$ do
8          $SimJoinSAX(R, N_S, \epsilon_{sp}, \epsilon_{ts})$
9  else if R is leaf ∧ S is leaf then             ▹ leaf level in both trees
10     foreach $T_R \in R.getChildren()$ do
11         foreach $T_S \in S.getChildren()$ do
12             if $d_{sp}(T_R, T_S) \leq \epsilon_{sp} \wedge d_{ts}(T_R, T_S) \leq \epsilon_{ts}$ then
13                 $Q \leftarrow Q \cup \{(T_R, T_S)\}$     ▹ add pair to result set
```

5.3 Hybrid Filtering using BTSR-Trees

This method makes use of a hybrid BTSR-tree index per dataset $\mathcal{T}_R, \mathcal{T}_S$ of geolocated time series. Initially, let us assume that both BTSR-trees have the same height. Exactly like the *i*SAX-based method, search starts from the root of each tree and descends them in tandem by checking their nodes pairwise, as listed in Algorithm 2.

In case that the currently examined entries R and S are internal (directory) nodes, a nested-loop check finds which of their descendants may contain candidate results (Lines 1-6). In particular, for each child entry N_R of node R, we calculate a buffer rectangle by expanding its respective MBR by distance ϵ_{sp}. In case that this buffer intersects with the MBR of entry N_S from node S, whereas also their MBTS do not deviate by more than ϵ_{ts}, then search should be recursively applied against those two entries N_R, N_S. Clearly, if neither of these criteria is met, those candidate entries cannot possibly contain any matching time series.

Note that this step involves comparison between two MBTSs. Consider two MBTSs $B_1 = (B_1^{\sqcup}, B_1^{\sqcap})$ and $B_2 = (B_2^{\sqcup}, B_2^{\sqcap})$, each constructed according to Eq. 4 over two disjoint subsets of time series data. We compute their *deviation* d_{MBTS} by first comparing the lower bounding time series of the former with the upper bounding time series of the latter per time point $i \in \{1, \ldots, n\}$, depending on which of these two values is larger, i.e.:

$$\delta_i = \begin{cases} B_1^{\sqcup}.v_i - B_2^{\sqcap}.v_i, & \text{if } B_1^{\sqcup}.v_i \geq B_2^{\sqcap}.v_i \\ B_2^{\sqcup}.v_i - B_1^{\sqcap}.v_i, & \text{if } B_2^{\sqcup}.v_i \geq B_1^{\sqcap}.v_i \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

and we take the average Euclidean norm over these n differences:

$$d_{MBTS}(B_1, B_2) = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(\delta_i)^2} \tag{6}$$

Quite importantly, this measure is a *lower bound* of the Euclidean distance $d_{ts}(T_1, T_2)$ between two time series T_1, T_2 that are enclosed in MBTSs B_1, B_2, respectively. Consider the situation at a given time point $i \in \{1, \ldots, n\}$. In case that $B_1^{\sqcup}.v_i \geq B_2^{\sqcap}.v_i$, it is straighforward that $T_1.v_i \geq B_1^{\sqcup}.v_i \geq B_2^{\sqcap}.v_i \geq T_2.v_i$ by definition of the MBTS, hence $\delta_i' = T_1.v_i - T_2.v_i \geq \delta_i$. This also holds for the other branches of Eq. 5. But, taking the Euclidean norm of these δ_i' values over all time points expresses the time series similarity according to Eq. 1. Overall, this confirms that $d_{ts}(T_1, T_2) \geq d_{MBTS}(B_1, B_2)$, so checking with MBTSs does not cause any false misses.

If both R and S are leaves, the algorithm retrieves all time series from either leaf and checks every combination against both criteria (Lines 10-14). If a time series from $\mathcal{T}_R$ and a time series from $\mathcal{T}_S$ are close enough in space (i.e., less than ϵ_{sp}) and also similar in the time series domain by ϵ_{ts}, this pair is issued as result.

Algorithm 2: *SimJoinBTSR*$(R, S, \epsilon_{sp}, \epsilon_{ts})$

```
Input: Nodes R, S, spatial constraint $\epsilon_{sp}$, time series constraint $\epsilon_{ts}$
Output: Set Q of pairs of geolocated time series satisfying constraints
1  if R is not leaf ∧ S is not leaf then          ▹ internal nodes in both trees
2      foreach $N_R \in R.getChildren()$ do
3          $N_R.buf \leftarrow buffer(N_R.mbr, \epsilon_{sp})$     ▹ expand MBR by $\epsilon_{sp}$
4          foreach $N_S \in S.getChildren$ do
5              if $N_R.buf \cap N_S.mbr \neq \emptyset \wedge d_{MBTS}(N_R, N_S) \leq \epsilon_{ts}$ then
6                  $SimJoinBTSR(N_R, N_S, \epsilon_{sp}, \epsilon_{ts})$
7  else if R is leaf ∧S is not leaf then          ▹ trees of different height
8      foreach $N_S \in S.getChildren()$ do
9          $SimJoinBTSR(R, N_S, \epsilon_{sp}, \epsilon_{ts})$
10 else if R is leaf ∧ S is leaf then             ▹ leaf level in both trees
11     foreach $T_R \in R.getChildren()$ do
12         foreach $T_S \in S.getChildren()$ do
13             if $d_{sp}(T_R, T_S) \leq \epsilon_{sp} \wedge d_{ts}(T_R, T_S) \leq \epsilon_{ts}$ then
14                 $Q \leftarrow Q \cup \{(T_R, T_S)\}$     ▹ add pair to result set
```

Handling the case of BTSR-tree indices with different height is handled as in R-trees [3]. Without loss of generality, let the first BTSR-tree (over $\mathcal{T}_R$) be shorter than the second BTSR-tree (over $\mathcal{T}_S$). Then, once a leaf entry R is reached in the former, comparisons are made against any subtrees under nodes N_S in the latter (Lines 7-9). In case that both criteria are met, we descend this latter BTSR-tree and recursively check for similarity joins between its children entries with the same leaf entry R fixed in the first BTSR-tree. Eventually, the leaf level in the second BTSR-tree will be reached and refinement against the raw time series on both the spatial and time series criteria can yield the final results.

6 DISTRIBUTED SIMILARITY JOIN

As any join query over large datasets, computing hybrid similarity joins over millions of geolocated time series is a very demanding task. Building a global index per dataset and applying any of the methods in Section 5 still incurs excessive cost, as demonstrated in our empirical tests (Section 7). To tackle scalability, we present a *parallel and distributed* approach based on space-driven partitioning (Section 6.1). We also describe an *optimized*, index-guided variant to reduce the amount of data shuffled between workers (Section 6.2).

6.1 MapReduce Method with Spatial Partitioning

Typically, in MapReduce-based processing, both input datasets $\mathcal{T}_R$, $\mathcal{T}_S$ should be divided into smaller chunks that may be efficiently processed in a distributed fashion by a number of worker nodes. In our case, distributing geolocated time series data by their spatial location is straightforward and can be effectuated much faster as opposed to a times series-based subdivision that may need to examine long sequences. Our method employs a subdivision $\mathcal{P}$ into *disjoint partitions* over the spatial area covering all locations in either dataset $\mathcal{T}_R$, $\mathcal{T}_S$. Partitioning $\mathcal{P}$ is *identical* over both datasets. Without loss of generality, we consider $\mathcal{P}$ as a uniform grid tessellation into $g \times g$ square equi-sized cells, but our method can be easily adjusted to other space-driven subdivisions into disjoint regions (e.g., quadtrees). Choosing a suitable grid granularity g over each axis mostly depends on dataset size, but also on the number and processing power of available nodes in cluster infrastructures.

The pseudocode listed in Algorithm 3 outlines the entire process. It proceeds in two successive phases: (1) a *local search* per partition and (2) *cross-partition search* by shuffling subsets of data between neighboring partitions. In particular, we make use of distinct *tiers* of *blocks* with increasingly finer spatial resolution (Figure 4):

1) *Local search per partition* (Lines 1-6): The first tier concerns individual *partitions*, and the algorithm needs to check for similarity join between those objects from $\mathcal{T}_R$ and those from $\mathcal{T}_S$ contained in the same partition $p \in \mathcal{P}$. This is depicted for a given partition (cell) p enclosed with dashed line segments in Figure 4 over each of the two datasets.

2a) *Cross-partition search in pairs of adjacent bands* (Lines 7-12): A collection $\mathcal{B}$ of spatial *bands* of width ϵ_{sp} is created inwards along each side of every partition in $\mathcal{P}$. Objects of each dataset coming from adjacent bands across every pair of neighboring partitions need to be checked against the query criteria. For a given partition p, each of the four bands created over $\mathcal{T}_R$ must be compared with respective bands created not in the same partition p for $\mathcal{T}_S$, but in each of the four partitions sharing one common side with p. In Figure 4, the pairs of respective bands are shown hatched with the same colored pattern and are connected with curly arrows. A set $L_{\mathcal{B}}$ consisting of pairs of such adjacent bands indicates those that must be probed across all partitions.

2b) *Cross-partition search in pairs of boxes with one common corner* (Lines 13-18): The finest tier concerns a set C of square boxes of side ϵ_{sp} created at the corner of each partition in $\mathcal{P}$. Objects from either dataset contained in boxes having one common corner need further probing (i.e., *corner-wise* in $\mathcal{P}$), and all pairs of such boxes are collected in set L_C. As shown in Figure 4, each box c created at the four corners of partition p over $\mathcal{T}_R$ should be checked against one equi-sized box over $\mathcal{T}_S$; this latter box belongs to a neighboring partition $p' \neq p$, which has only one common corner with p.

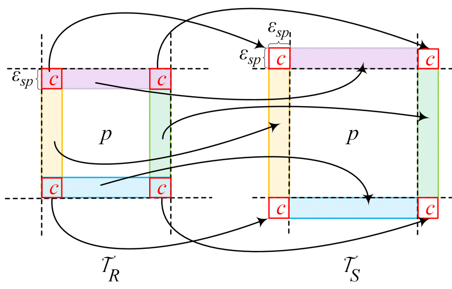


Figure 4: Blocks in cross-partition search for a partition p.

Since all blocks are purely space-driven, the rationale is that spatial filtering comes first, whilst the time series criterion is checked afterwards for any remaining candidate pairs. At each block level, our method creates disjoint data chunks for subsets of objects located in that block; this is applied against both datasets similarly for *partitions* (Lines 2-3), *bands* (Lines 8-9), and *boxes* (Lines 14-15).

Furthermore, at each block tier, a *local index* is built for every derived chunk. Interestingly, we may plug in any of the similarity join methods suggested in Section 5. The same indexing scheme must be used at each tier, i.e., either R-tree, *i*SAX, or BTSR-tree (hereafter referred to as $\mathcal{X}$-index). For partitions, such indices can be suitably built *in advance* with a predefined subdivision $\mathcal{P}$ and thus can be readily available for any similarity join query that may specify varying values on parameters ϵ_{sp} and ϵ_{ts}. In contrast, indices over data contained in each of the bands listed in $L_{\mathcal{B}}$ or each corner box in L_C have to be created *at query time*, since they clearly rely on distance threshold ϵ_{sp}, which may vary among queries.

Once pairs of blocks need be checked at each tier (either $L_{\mathcal{P}}$ for partitions, or $L_{\mathcal{B}}$ for bands, or L_C for boxes), *blockwiseSimJoin* (Lines 19-25) takes advantage of the created indices and applies the respective method from Section 5 to return their results. Each pair of blocks at any tier can be processed independently. Hence, for a given partitioning $\mathcal{P}$, once the query is submitted, the required subsets and their indices can be prepared in a *distributed* fashion and the respective block-wise checking can be evaluated *in parallel*. For a given partition p (first tier), subsets from both datasets are assigned to the same worker node. This policy is also applied in the case of blocks that need to be checked: the worker responsible for a given partition p receives the data and index concerning objects

Algorithm 3: $SimJoinMR(\mathcal{T}_R, \mathcal{T}_S, \epsilon_{sp}, \epsilon_{ts}, \mathcal{X})$

```
Input: dataset $\mathcal{T}_R$, dataset $\mathcal{T}_S$, spatial constraint $\epsilon_{sp}$, time series
       constraint $\epsilon_{ts}$, index method $\mathcal{X}$ (e.g., BTSR-tree or R-tree)
Output: Set $Q$ of pairs of geolocated time series satisfying constraints
/* PHASE #1: local search per partition */
$\mathcal{P} \leftarrow$ space partitioning common for both datasets $\mathcal{T}_R, \mathcal{T}_S$
$R_{\mathcal{P}} \leftarrow$ distribute $\mathcal{T}_R$ and build a local $\mathcal{X}$-index per partition $p \in \mathcal{P}$
$S_{\mathcal{P}} \leftarrow$ distribute $\mathcal{T}_S$ and build a local $\mathcal{X}$-index per partition $p \in \mathcal{P}$
$L_{\mathcal{P}} \leftarrow \{(p, p) : \forall p \in \mathcal{P}\}$        ▷ pairs of identical partitions
$Q_{\mathcal{P}} \leftarrow blockwiseSimJoin(L_{\mathcal{P}}, R_{\mathcal{P}}, S_{\mathcal{P}}, \epsilon_{sp}, \epsilon_{ts})$
$storeHDFS(Q_{\mathcal{P}})$        ▷ partial results over partitions
/* PHASE #2a: cross-partition search in pairs of adjacent bands */
$\mathcal{B} \leftarrow$ create bands of width $\epsilon_{sp}$ inwards each side of every $p \in \mathcal{P}$
$R_{\mathcal{B}} \leftarrow$ filter $R_{\mathcal{P}}$ by $\mathcal{B}$ and build a local $\mathcal{X}$-index per band $b \in \mathcal{B}$
$S_{\mathcal{B}} \leftarrow$ filter $S_{\mathcal{P}}$ by $\mathcal{B}$ and build a local $\mathcal{X}$-index per band $b \in \mathcal{B}$
$L_{\mathcal{B}} \leftarrow \{(r \in R_{\mathcal{B}}, s \in S_{\mathcal{B}})$ : bands $r.b, s.b$ share a side in partitioning $\mathcal{P}\}$
$Q_{\mathcal{B}} \leftarrow blockwiseSimJoin(L_{\mathcal{B}}, R_{\mathcal{B}}, S_{\mathcal{B}}, \epsilon_{sp}, \epsilon_{ts})$
$storeHDFS(Q_{\mathcal{B}})$        ▷ partial results over bands
/* PHASE #2b: cross-partition search in corner-wise pairs of boxes */
$C \leftarrow$ create boxes of side $\epsilon_{sp}$ at the corners of each partition $p \in \mathcal{P}$
$R_C \leftarrow$ filter $R_{\mathcal{P}}$ by $C$ and build a local $\mathcal{X}$-index per box $c \in C$
$S_C \leftarrow$ filter $S_{\mathcal{P}}$ by $C$ and build a local $\mathcal{X}$-index per box $c \in C$
$L_C \leftarrow \{(r \in R_C, s \in S_C)$ : boxes $r.c, s.c$ share a single corner in $\mathcal{P}\}$
$Q_C \leftarrow blockwiseSimJoin(L_C, R_C, S_C, \epsilon_{sp}, \epsilon_{ts})$
$storeHDFS(Q_C)$        ▷ partial results over boxes

Function $blockwiseSimJoin(L, R, S, \epsilon_{sp}, \epsilon_{ts})$
    $Q \leftarrow \emptyset$
    foreach block pair $(a, b) \in L$ do        ▷ local search
        $\mathcal{I}_a^R \leftarrow$ local $\mathcal{X}$-index available for dataset $R$ in block $a$
        $\mathcal{I}_b^S \leftarrow$ local $\mathcal{X}$-index available for dataset $S$ in block $b$
        $Q \leftarrow Q \cup SimJoin\mathcal{X}(\mathcal{I}_a^R.root, \mathcal{I}_b^S.root, \epsilon_{sp}, \epsilon_{ts})$
    return $Q$        ▷ results collected from all pairs of blocks
```

from the other dataset within an adjacent block. Such processing fits well under the *MapReduce* paradigm; mappers assign data subsets to workers according to partitioning scheme $\mathcal{P}$. Each worker employs reduce operations to generate the respective indices and store them on HDFS. At query time, a map procedure assigns the indices that reside within each partition to a reducer, which calculates the local results. Simultaneously, mappers shuffle data per block to workers responsible for their neighboring partitions. Finally, a reduce operation is carried out on each pair of such blocks to compute their similarity joins and to store results on HDFS.

Overall, this method manages to provide correct and complete results for any similarity join query over two datasets of geolocated time series. This is stated in the following:

Lemma 1. *Algorithm 3 issues all qualifying results of similarity join between two datasets $\mathcal{T}_R$, $\mathcal{T}_S$ of geolocated time series, without probing candidate pairs more than once and without any false misses.*

Proof. Regarding *correctness*, consider a given partition $p \in \mathcal{P}$, as depicted in Figure 4 and let R_p be the objects of $\mathcal{T}_R$ having their location contained therein. Obviously, any of their possibly qualifying pairs from dataset $\mathcal{T}_S$ must be within distance ϵ_{sp}. So, it suffices to examine similarity between objects in R_p with those objects of $\mathcal{T}_S$ topologically located within a *buffer* that expands partition p by distance ϵ_{sp}. Clearly, the area covered by the nine blocks (one partition, four bands, and four boxes) concerning $\mathcal{T}_S$ is exactly this buffer zone, so $Q_{cand}(p) = \{(T_R, T_S) : within(T_R.loc, p), within(T_S.loc, buffer(p, \epsilon_{sp}))\}$ provides all possible candidates in a given partition p. As partitions hold disjoint subsets of the raw data, iterating with the same logic over each partition $p \in \mathcal{P}$, confirms that all candidates are examined.

Regarding *completeness*, observe that the pairs of blocks involved at each stage include all possible candidates to probe from each dataset. At the first tier, searching in each partition p (common for either dataset) provides all qualifying pairs having their constituent objects both located in p. Cross-searching beyond the boundary of each partition p is meaningful only along the adjacent bands and boxes, each of them coming from a distinct neighboring partition to p. Clearly, in each of those nine blocks, a disjoint subset of candidate pairs from the two datasets is examined and their union is $Q_{cand}(p)$. Hence, each candidate pair is probed only once, and no qualifying results can ever be missing from the final answer. □

6.2 Minimizing Data Shuffling

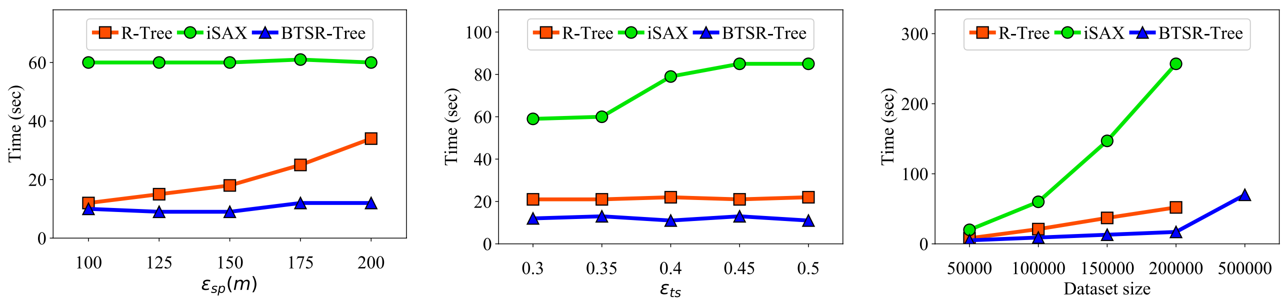
Recall that a worker w already has locally available all data for subsets of $\mathcal{T}_R, \mathcal{T}_S$ located in its assigned partition p. This is sufficient for its own local search, but w still needs to send raw data concerning its four bands and four boxes for the cross-partition search.

This evaluation strategy can be further optimized by minimizing the amount of data that needs to be shuffled during cross-partition search. We introduce an intermediate filtering step that takes advantage of the pruning power of our BTSR-tree index. Consider a given block a over dataset $\mathcal{T}_R$ held in worker w' that must be shipped to worker w responsible for partition p. Instead, w' builds a BTSR-tree $\mathcal{I}_a^R$ over this subset in a, and sends this index only to worker w, which builds its own BTSR-tree $\mathcal{I}_b^S$ over its local subset of $\mathcal{T}_S$ within its corresponding block b. Checking for similarity joins against those two indices can be carried out with Algorithm 2. This returns pairs $\{(mbr_i^a, mbr_j^b)\}$ of overlapping MBRs, where mbr_i^a is an MBR over block a and mbr_j^b is over block b, and each one contains candidate objects for refinement. The list $\{mbr_i^a\}$ of all identified MBRs concerning block a is returned to worker w' and the raw geolocated time series within each such MBR can be readily accessed thanks to the already available BTSR-tree $\mathcal{I}_a^R$. Those MBR-filtered time series are then shipped to worker w, which also retrieves its own raw data from BTSR-tree $\mathcal{I}_b^S$ concerning those MBRs $\{mbr_j^b\}$ identified for its own block b. Finally, those two MBR-filtered subsets of geolocated time series are each indexed with a new BTSR-tree and joined according to the similarity criteria to yield their matching results. As confirmed in our empirical tests, this index-guided shuffling can reduce the raw data transferred between workers by more than 50% without sacrificing performance.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Setup

We generated several *synthetic* datasets of various sizes, using a real-world water consumption dataset as a seed. This real dataset, provided by the DAIAD project (http://daiad.eu/), contained geolocated time series of hourly water consumption for 822 households in Alicante, Spain from 1/1/2015 to 20/1/2017. On this data, we first



(a) Varying ϵ_{sp} (b) Varying ϵ_{ts} (c) Scalability

Figure 5: Processing cost for *centralized* execution of similarity join queries employing different indices.

Table 1: Parameters tested in the experiments

Parameter	*Values*
Dataset size (*centralized*)	50K, 100K, 150K, **200K**, 500K, 1000K
Dataset size (*distributed*)	500K, **1000K**, 1500K, 2000K
Number of partitions $g \times g$ (*distributed*)	10^2, 20^2, $\mathbf{30^2}$,40^2,50^2,60^2,70^2
Distance radius ϵ_{sp} (meters) in queries	100, 125, **150**, 175, 200
Time series deviation ϵ_{ts} in queries	0.3, **0.35**, 0.4, 0.45, 0.5

calculated the weekly (24×7) time series per household by averaging corresponding hourly values over the entire period. Then, these weekly sequences were used as seeds to synthetically increase the size of the dataset up to 2 million geolocated time series, by introducing small random variations in their location and pattern.

In preliminary tests, we fine-tuned parameters for the various indices used against this data. For BTSR-trees and R-trees, the number of entries per node ranges between m=10 and M=50. In *i*SAX, up to M=250 time series can be stored per leaf and the length of each SAX word is w=8. Table 1 lists the range of values for the rest of parameters used in our tests; default values are in bold.

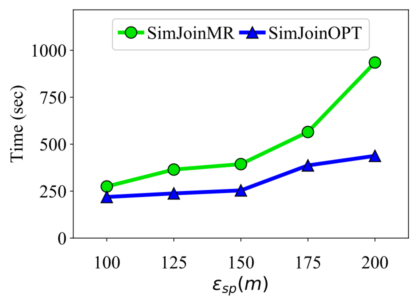
All algorithms were implemented in Java. Distributed methods were developed on Apache Spark 2.3.0. The centralized experiments were executed on a machine running MacOS 10.13.5 with a 2GHz CPU and 8GB of RAM. The distributed tests were conducted on a cluster with 7 virtual machines running Ubuntu 16.04.3 LTS, with 4 cores each, clocked at 2100MHz. Each node had a total of 5GB of RAM. Next, we report performance in terms of average response time per query. Each query runs against two instances of the same dataset (i.e., *self-join*), excluding identity matches from resulting pairs. In the distributed case, we also measure the amount of raw data transferred between workers during the cross-partition phase.

7.2 Evaluation Results

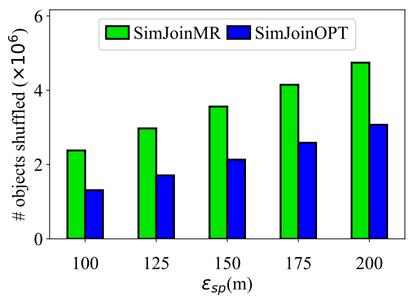
7.2.1 Centralized Methods. Figure 5 depicts performance for different parameter values and dataset sizes. The *i*SAX-based algorithm performs significantly worse than the rest, mostly because each node comparison involves reconversion of the *i*SAX symbols to the Euclidean space [18] and consequently, calculation of Euclidean distances over long sequences (up to 168 values in this data). BTSR-tree is superior in all cases, as it is able to prune in both time series and spatial domains. As shown in Figure 5(a), BTSR-tree and R-tree-based methods perform similarly for smaller ϵ_{sp} values, as fewer candidates are found and need refinement in the time series domain. However, as the distance radius ϵ_{sp} is relaxed, R-tree search worsens significantly, while BTSR-tree still copes well due to its hybrid pruning ability. *i*SAX-based search is immune to different values of ϵ_{sp}, as filtering with spatial distance is only involved at refinement. With varying ϵ_{ts} values (Figure 5(b)), BTSR-tree and R-tree approaches have no fluctuations in performance, as ϵ_{sp} is fixed and refinement of candidates involves a similar cost in the time series domain. However, using *i*SAX indexing is faster for lower ϵ_{ts} values and performance slowly degrades for larger ϵ_{ts}, as more candidates become eligible. In terms of scalability (Figure 5(c)), all algorithms are almost equally fast over small datasets. But, as dataset sizes grow, the BTSR-tree approach scales better thanks to its hybrid pruning, although the number of matching pairs escalates. Indicatively, for 100K input data, we get 2K qualifying pairs; in the 500K dataset, we get 40K results. For input data sizes larger than 500K, all centralized methods fail to finish execution, issuing an out-of-memory error. This manifests the necessity of distributed processing schemes for similarity joins over larger datasets.

7.2.2 Distributed Methods. First, we compare *SimJoinMR* (using R-trees for local indexing) with its *SimJoinOPT* variant (employing BTSR-trees) for varying ϵ_{sp} values. It is apparent from Figure 6(a) that query response times for the *SimJoinMR* method are increasing, since the underlying R-trees fare worse for larger distance radii. With larger ϵ_{sp} values, more raw data has to be shuffled between workers during the cross-partition checks, as the size of bands and boxes involved gets bigger and covers more candidate objects. Concerning exactly this shuffling overhead, Figure 6(b) reveals that this is indeed lower in the *SimJoinOPT* variant, which explains its processing cost advantage. Finally, Figure 6(c) illustrates the number of results produced from the two stages; first locally in each partition, and then after cross-partition checks in bands and boxes. As distance constraint ϵ_{sp} gets more relaxed, more pairs qualify as answers. For smaller ϵ_{sp}, the majority of results come locally from each partition. But as ϵ_{sp} is relaxed, many more pairs are found in neighboring partitions, as bands and boxes also become larger and increase their share in qualifying results much more.

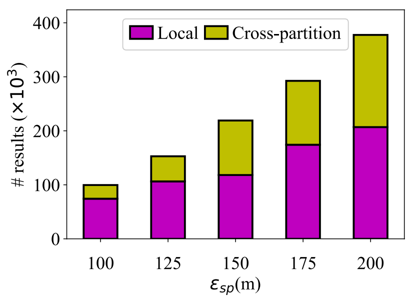
With regard to increasing ϵ_{ts} values, observe in Figure 7(a) that method *SimJoinMR* is consistently worse than *SimJoinOPT*, basically due to the different pruning power of their respective indices. The former relies on R-trees, which have no effect with varying ϵ_{ts}; in contrast, BTSR-trees employed by *SimJoinOPT* can effectively filter candidates also in the time series domain. With a more relaxed ϵ_{ts}, more results qualify, hence the linear increase in processing



(a) Query response time

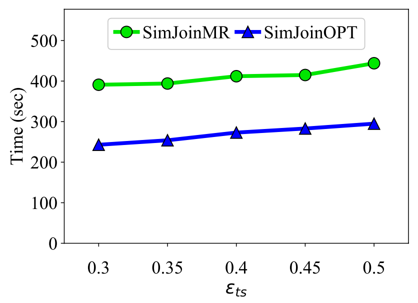


(b) Geolocated time series shuffled

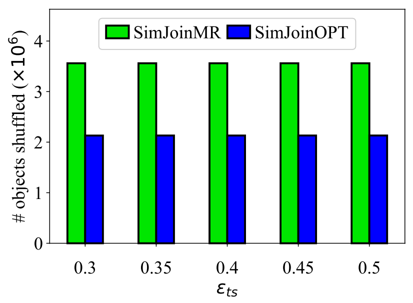


(c) Number of query results per phase

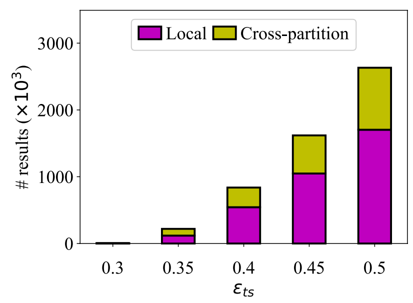
Figure 6: Performance results for the *distributed* methods with varying ϵ_{sp}.



(a) Query response time

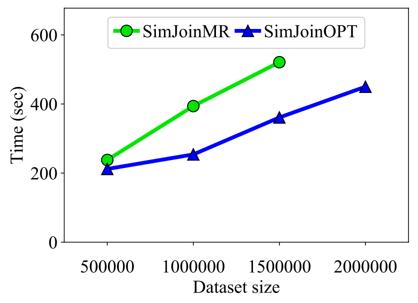


(b) Geolocated time series shuffled

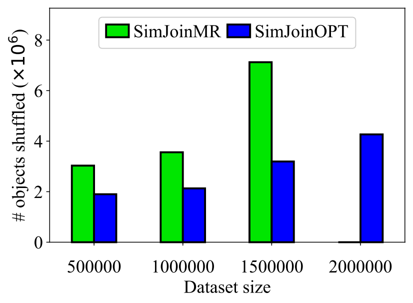


(c) Number of query results per phase

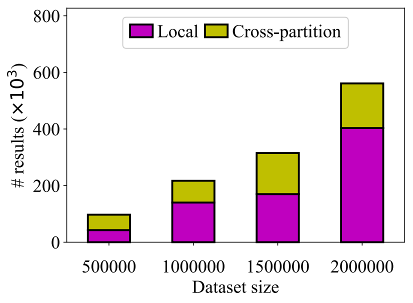
Figure 7: Performance results for the *distributed* methods with varying ϵ_{ts}.



(a) Query response time

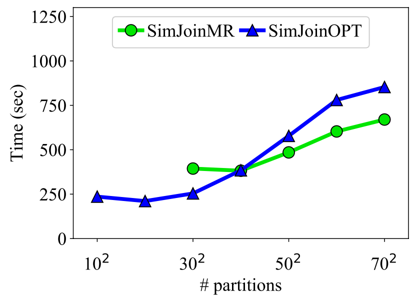


(b) Geolocated time series shuffled

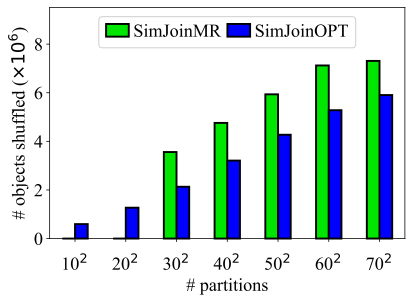


(c) Number of query results per phase

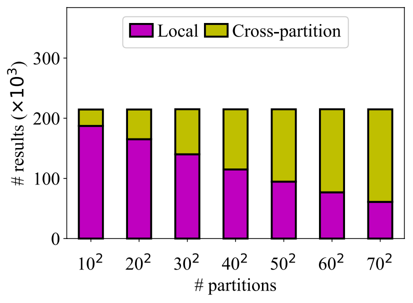
Figure 8: Scalability of the *distributed* methods.



(a) Query response time



(b) Geolocated time series shuffled



(c) Number of query results per phase

Figure 9: Effect of partitioning on the performance of *distributed* methods.

cost. Regarding the data shuffling overhead, this is practically stable for each method irrespective of the ϵ_{ts} constraint (Figure 7(b)). In *SimJoinMR*, selection of objects that should be transmitted is solely based on their spatial containment in the respective bands and corner-wise boxes. But *SimJoinOPT* spares transferring many irrelevant objects, as it also uses filtering with ϵ_{ts}; the amount of dispatched objects is only slightly increasing with ϵ_{ts}. Regarding the number of generated results, Figure 7(c) reveals a rather steep increase for small variations of ϵ_{ts}, which indicates that most time series are clustered within a small range of ϵ_{ts} deviations. As original data concern water consumption, this explains such highly correlated behavior, especially among neighboring households; of course, this pattern is replicated in the synthetic data as well. The percentage of results from cross-partition checks in each full answer is similar across various ϵ_{ts} values, as distance ϵ_{sp} is fixed and so are the respective bands and boxes involved in this phase.

Figure 8 concerns scalability of the distributed methods with increasing dataset sizes. For smaller datasets, both methods are competitive, but response times for *SimJoinMR* escalate with larger sizes. With 2 million objects as input, this method did not finish, as it required traversal of too many paths in its underlying R-trees per partition, exceeding the capabilities of the workers. Regarding communication (Figure 8(b)), *SimJoinMR* requires shuffling of more raw data, especially for input size of 1.5 million. In contrast, *SimJoinOPT* maintains lower communication overhead, as it uses light-weight indices to guide data shuffling. Figure 8(c) indicates that the number of results is growing according to the input size, as the spatial density also increases with larger synthetic datasets that still cover the same area (Alicante).

Last but not least, we conducted tests concerning partitioning, i.e., varying the grid granularity and distributing input data accordingly. As shown in Figure 9(a), *SimJoinMR* was not able to conclude its evaluation over coarser spatial subdivisions, as each resulting partition can hardly cope with the larger subsets of data held locally. For 30×30 partitions, *SimJoinOPT* performs better thanks to the superiority of BTSR-tree in pruning. But *SimJoinMR* overtakes *SimJoinOPT* when allowing finer partitioning (40×40 partitions or more), as the R-tree overhead diminishes. Each such index has to deal with smaller subsets, although it incurs higher communication overhead compared to *SimJoinOPT* (Figure 9(b)). Indeed, having more partitions forces *SimJoinOPT* to search for joins pairwise in many more bands and boxes, while also building the respective intermediate indices. So, such optimization really compensates with a coarser partitioning, achieving its best performance with a 20×20 grid as depicted in Figure 9(a). Finally, Figure 9(c) indicates that the majority of resulting pairs are derived locally under a coarser partitioning. However, this is reversed with finer partitioning, as the size of blocks (bands and boxes) during the cross-partition checks cover much more area per cell, hence many more qualifying pairs are found while searching across neighboring partitions.

8 CONCLUSIONS

In this paper, we addressed the problem of hybrid similarity joins over geolocated time series according to both their spatial proximity and time series similarity. Our approach takes advantage of different state-of-the-art indexing schemes to design an efficient algorithm for query evaluation. Given that scalability is a bottleneck in such centralized settings, we show how a space-driven partitioning can be employed to deal with much larger datasets in cluster environments. Our parallel and distributed method can efficiently execute similarity joins per partition locally, while also minimizing the amount of data shuffled between processing nodes. Our empirical results against synthetic datasets of varying sizes confirm the efficiency and effectiveness of our algorithms.

ACKNOWLEDGMENTS

This work was partially funded by the General Secretariat for Research and Technology (GSRT), the Hellenic Foundation for Research and Innovation (HFRI), the EU H2020 project SLIPO (731581) and the NSRF 2014-2020 project HELIX (5002781).

REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.
[2] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.
[3] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, pages 237–246, 1993.
[4] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. iSAX 2.0: Indexing and mining one billion time series. In *ICDM*, pages 58–67, 2010.
[5] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. *Knowl. Inf. Syst.*, 39(1):123–151, 2014.
[6] K. Chan and A. W. Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.
[7] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, S. Athanasiou, and S. Skiadopoulos. Indexing geolocated time series data. In *SIGSPATIAL*, pages 19:1–19:10, 2017.
[8] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. J. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.
[9] D. Q. Goldin and P. C. Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In *CP*, pages 137–153, 1995.
[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
[11] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowl. Inf. Syst.*, 3(3):263–286, 2001.
[12] J. Lin, E. J. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, 2007.
[13] T. Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In *SOFSEM*, pages 63–80, 2016.
[14] D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *SIGMOD*, pages 44–55, 1999.
[15] I. Popivanov and R. J. Miller. Similarity search over time-series data using wavelets. In *ICDE*, pages 212–221, 2002.
[16] S. Qi, P. Bouros, and N. Mamoulis. Efficient top-k spatial distance joins. In *SSTD*, pages 1–18, 2013.
[17] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.
[18] J. Shieh and E. J. Keogh. *i*SAX: indexing and mining terabyte sized time series. In *SIGKDD*, pages 623–631, 2008.
[19] B. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary Lp norms. In *VLDB*, pages 385–394, 2000.
[20] Y. Zhang, Y. Ma, and X. Meng. Efficient spatio-textual similarity join using mapreduce. In *WI*, pages 52–59, 2014.
[21] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD*, pages 1555–1566, 2014.