De Francisci Morales, Gianmarco; Gionis, Aristides

Streaming similarity self-join

Published in:
Proceedings of the VLDB Endowment

Published: 01/01/2016

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
De Francisci Morales, G., & Gionis, A. (2016). Streaming similarity self-join. In *Proceedings of the VLDB Endowment* (pp. 792-803). (Proceedings of the VLDB endowment; Vol. 9, No. 10). ACM.

Streaming Similarity Self-Join

Gianmarco De Francisci Morales
Qatar Computing Research Institute
gdfm@acm.org

Aristides Gionis
Aalto University
aristides.gionis@aalto.fi

ABSTRACT

We introduce and study the problem of computing the similarity self-join in a streaming context (SSSJ), where the input is an unbounded stream of items arriving continuously. The goal is to find all pairs of items in the stream whose similarity is greater than a given threshold. The simplest formulation of the problem requires unbounded memory, and thus, it is intractable. To make the problem feasible, we introduce the notion of *time-dependent similarity:* the similarity of two items decreases with the difference in their arrival time.

By leveraging the properties of this time-dependent similarity function, we design two algorithmic frameworks to solve the SSSJ problem. The first one, MiniBatch (`MB`), uses existing index-based filtering techniques for the static version of the problem, and combines them in a pipeline. The second framework, Streaming (`STR`), adds *time filtering* to the existing indexes, and integrates new time-based bounds deeply in the working of the algorithms. We also introduce a new indexing technique (`L2`), which is based on an existing state-of-the-art indexing technique (`L2AP`), but is optimized for the streaming case.

Extensive experiments show that the `STR` algorithm, when instantiated with the `L2` index, is the most scalable option across a wide array of datasets and parameters.

1. INTRODUCTION

Similarity self-join is the problem of finding *all pairs* of similar objects in a given dataset. The problem is related to the *similarity join* operator [4, 10], which has been studied extensively in the database and data-mining communities. The similarity self-join is an essential component in several applications, including plagiarism detection [11, 16], query refinement [21], document clustering [8, 9, 15], data cleaning [10], community mining [22], near-duplicate record detection [28], and collaborative filtering [12].

The similarity self-join problem is inherently quadratic. In fact, the brute-force $\mathcal{O}(n^2)$ algorithm that computes the similarity between all pairs is the best one can hope for, in the worst case, when exact similarity is required and when

Proceedings of the VLDB Endowment, Vol. 9, No. 10
Copyright 2016 VLDB Endowment 2150-8097/16/06.

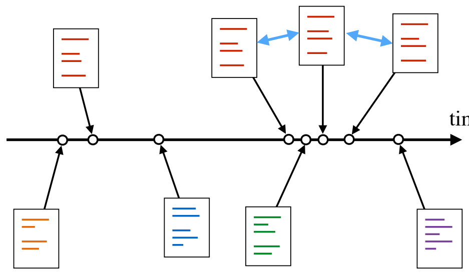

Figure 1: Timestamped documents arrive as stream. The documents on top (in red) have similar content. Among all pairs of similar documents, we are interested in those that arrive close in time. In the example, out of all 4-choose-2 pairs only two pairs are reported (shown with blue arrows).

dealing with arbitrary objects. In practice, it is possible to obtain scalable algorithms by leveraging structural properties of specific problem instances. A typical desiderata here is to design *output-sensitive* algorithms.[1]

In many real-world applications objects are represented as *sparse* vectors in a high-dimensional Euclidean space, and similarity is measured as the dot-product of unit-normalized vectors — or equivalently, cosine similarity. The similarity self-join problem asks for all pairs of objects whose similarity is above a predefined threshold θ. Efficient algorithms for this setting are well-developed, and rely on pruning based on inverted indexes, as well as a number of different geometric bounds [3, 7, 10, 28]; details on those methods are presented in the following sections.

Computing similarity self-join is a problem that is germane not only for static data collections, but also for data streams. Here we describe two examples of real-world applications that motivate the problem of similarity self-join in the streaming setting.

Trend detection: existing algorithms for trend detection in microblogging platforms, such as Twitter, rely on identifying hashtags or other terms whose frequency suddenly increases. A more focused and more granular trend-detection approach would be to identify a set of posts, whose frequency increases, and which share a certain *fraction* of hashtags or terms. For such a trend-detection method it is essential to be able to find similar pairs of posts in a data stream.

Near-duplicate item filtering: consider again a microblogging platform, such as Twitter. When an event occurs, users may receive multiple near-copies of posts related to the event. Such posts often appear consecutively in the feed of

[1]`https://en.wikipedia.org/wiki/Output-sensitive_algorithm`

users, thus cluttering their information stream and degrading their experience. Grouping these near-copies or filtering them out is a simple way to improve the user experience.

Surprisingly, the problem of streaming similarity self-join has not been considered in the literature so far. Possibly, because the similarity self-join operator in principle requires *unbounded* memory: one can never forget a data item, as it may be similar to another one that comes far in the future.

To the best of our knowledge, this paper is the first to address the similarity self-join problem in the streaming setting. We overcome the inherent unbounded-memory bottleneck by introducing a temporal factor to the similarity operator. Namely, we consider two data items similar, only if they have arrived within a *short time span.* More precisely, we define the *time-dependent similarity* of two data items to be their content-based cosine similarity multiplied by a factor that decays exponentially with the difference in their arrival times. This time-dependent factor allows us to drop old items, as they cannot be similar to any item that arrives after a certain time horizon τ. The concept is illustrated in Figure 1.

The time-dependent similarity outlined above is very well-suited for both our motivating applications — trend detection and near-duplicate filtering: in both cases we are interested in identifying items that not only are content-wise similar, but have also arrived within a short time interval.

Akin to previous approaches for the similarity self-join problem, our method relies on indexing techniques. Previous approaches use different types of *index filtering* to reduce the number of candidates returned by the index for full similarity evaluation. Following this terminology, we use the term *time filtering* to refer to the property of the time-dependent similarity that allows to drop old items from the index.

We present two different algorithmic frameworks for the streaming similarity self-join problem, both of which rely on the time-filtering property. Both frameworks can be instantiated with indexing schemes based on existing ones for the batch version of the problem. The first framework, named MiniBatch (`MB`), uses existing indexing schemes off-the-shelf. In particular, it uses two indexes in a pipeline, and it drops the older one when it becomes old enough. The second framework we propose, named Streaming (`STR`), modifies existing indexing schemes so that time filtering is incorporated internally in the index.

One of our contributions is a new index, `L2`, which combines state-of-the art bounds for index pruning from `L2AP` [3] in a way that is optimized for stream data. The superior performance of `L2` stems from the fact that it uses bounds that (*i*) are effective in reducing the number of candidate pairs, (*ii*) do not require collecting statistics over the data stream (which need to be updated as the stream evolves), (*iii*) lead to very lightweight index maintenance, while (*iv*) allowing to drop data as soon as they become too old to be similar to any item currently read. Our experimental evaluation demonstrates that the `L2` index, incorporated in the `STR` framework, is the method of choice for all datasets we try across a wide range of parameters.

In brief, our contributions can be summarized as follows.

- We introduce the similarity self-join problem in data streams.
- We propose a novel time-dependent similarity measure, which allows to forget data items when they become old.
- We show how to solve the proposed problem within two different algorithmic frameworks, `MB` and `STR`: For `MB`, any state-of-the-art indexing scheme can be used as a black box. For `STR`, we adapt and extend the existing schemes to be well-suited for streaming data.
- We perform an extensive experimental evaluation, and test the proposed algorithms on datasets with different characteristics and for a large range of parameters.

Finally, we note that all the code[2] we developed and all datasets[3] used for validation are publicly available.

2. RELATED WORK

The problem of *streaming similarity self-join* has not been studied previously. Work related to this problem can be classified into two main areas: similarity self-join (also known as all-pairs similarity), and streaming similarity.

Similarity self-join. The similarity self-join problem has been studied extensively. It was introduced by Chaudhuri et al. [10], and many works have followed up [4, 5, 7, 10, 27, 28], including approaches that use parallel computation [1, 2, 6, 13, 24]. Discussing all these related papers is out of the scope of this work. The most relevant approaches are those by Bayardo et al. [7], which significantly improved the scalability of the previous indexing-based methods, and by Anastasiu and Karypis [3], which represents the current state-of-the-art for sequential batch algorithms. Our work relies heavily on these two papers, and we summarize their common *filtering framework* in Section 5. For a good overview, refer to the work of Anastasiu and Karypis [3].

Streaming similarity. The problem of computing similarities in a stream of vectors has received surprisingly little attention so far. Most of the work on streaming similarity focuses on time series [14, 17, 19].

Lian and Chen [20] study the similarity join problem for *uncertain* streams of vectors in the sliding-window model. They focus on uncertain objects moving in a low-dimensional Euclidean space ($d \in [2, 5]$). Similarity is measured by the Euclidean distance, and therefore the pruning techniques are very different than the ones employed in our work; e.g., they use space-partition techniques, and use temporal correlation of samples to improve the efficiency of their algorithm.

Wang and Rundensteiner [26] study the more general problem of multi-way join with expensive user-defined functions (UDFs) in the sliding-window model. They present a time-slicing approach, which has some resemblance to the time-based pruning presented in this paper. However, their focus in on distributing the computation on a cluster via pipelined parallelism, and ensuring that the multiple windows of the multi-way join align correctly. Differently from this paper, their method treats UDFs as a black box.

Valari and Papadopoulos [23] focus on graphs, rather than vectors, and study the case of Jaccard similarity of nodes on a sliding window over an edge stream. They apply a count-based pruning on edges by keeping a fixed-size window, and leveraging an upper bound on the similarity within the next window. Instead, in this work we consider time-based pruning, without any assumption on the frequency of arrival of items in the stream. Furthermore, the arrival of edges changes the similarity of the nodes, and therefore different indexing techniques need to be applied; e.g., nodes can never be pruned.

[2]Code: `http://github.com/gdfm/sssj`

[3]Data: `http://research.ics.aalto.fi/dmg/sssj-datasets/`

3. PROBLEM STATEMENT

We consider data items represented as vectors in a d-dimensional Euclidean space $\mathbb{R}^d$. In real-world applications, the dimensionality d is typically high, while the data items are sparse vectors. Given two vectors $\mathbf{x} = \langle \mathrm{x}_1 \dots \mathrm{x}_d \rangle$ and $\mathbf{y} = \langle \mathrm{y}_1 \dots \mathrm{y}_d \rangle$, their similarity $\mathrm{sim}(\mathbf{x}, \mathbf{y})$ is the dot-product

$$\mathrm{sim}(\mathbf{x}, \mathbf{y}) = \mathrm{dot}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} = \sum_{j=1}^{d} \mathrm{x}_j \, \mathrm{y}_j.$$

We assume that all vectors $\mathbf{x}$ are normalized to unit length, i.e., $||\mathbf{x}||_2 = 1$, so that the dot-product of two vectors is equal to their cosine similarity; $\cos(\mathbf{x}, \mathbf{y}) = \mathrm{dot}(\mathbf{x}, \mathbf{y})$.

In the standard *all-pairs similarity search* problem (APSS), also known as *similarity self-join*, we are given a set of vectors and a similarity threshold θ, and the goal is to find all pairs of vectors $(\mathbf{x}, \mathbf{y})$ for which $\mathrm{sim}(\mathbf{x}, \mathbf{y}) \geq \theta$.

In this paper we assume that the input items arrive as a data stream. Each vector $\mathbf{x}$ in the input stream is timestamped with the time of its arrival $t(\mathbf{x})$, and the stream is denoted by $\mathcal{S} = \langle \dots, (\mathbf{x}_i, t(\mathbf{x}_i)), (\mathbf{x}_{i+1}, t(\mathbf{x}_{i+1}), \dots \rangle$.

We define the similarity of two vectors $\mathbf{x}$ and $\mathbf{y}$ by considering not only their coordinates $(\mathrm{x}_j, \mathrm{y}_j)$, but also the difference in their arrival times in the input stream $\Delta t_{\mathbf{xy}} = |t(\mathbf{x}) - t(\mathbf{y})|$. For fixed coordinates, the larger the arrival time difference of two vectors, the smaller their similarity. In particular, given two timestamped vectors $\mathbf{x}$ and $\mathbf{y}$ we define their *time-dependent similarity* $\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y})$ as

$$\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) = \mathrm{dot}(\mathbf{x}, \mathbf{y}) \, e^{-\lambda |t(\mathbf{x}) - t(\mathbf{y})|},$$

where λ is a *time-decay* parameter.

This time-dependent similarity reverts to the standard dot-product (or cosine) similarity when $\Delta t_{\mathbf{xy}} = 0$ or $\lambda = 0$, and it goes to zero as $\Delta t_{\mathbf{xy}}$ approaches infinity, at an exponential rate modulated by λ. As in the standard APSS problem, given a similarity threshold θ, two vectors $\mathbf{x}$ and $\mathbf{y}$ are called *similar* if their time-dependent similarity is above the threshold, i.e., $\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) \geq \theta$.

We can now define the streaming version of the APSS problem, called the *streaming similarity self-join* problem (SSSJ).

PROBLEM 1 (SSSJ). *Given a stream of timestamped vectors $\mathcal{S}$, a similarity threshold θ, and a time-decay factor λ, output all pairs of vectors $(\mathbf{x}, \mathbf{y})$ in the stream such that $\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) \geq \theta$.*

Time filtering. The main challenge of computing a self-join in a data stream is that, in principle, unbounded memory is required, as an item can be similar to any other item that will arrive arbitrarily far in the future. By adopting the time-dependent similarity measure $\mathrm{sim}_{\Delta t}$, we introduce a forgetting mechanism, which not only is intuitive from an application point-of-view, but also allows to overcome the unbounded-memory requirement. In particular, since for ℓ_2-normalized vectors $\mathrm{dot}(\mathbf{x}, \mathbf{y}) \leq 1$, we have

$$\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) = \mathrm{dot}(\mathbf{x}, \mathbf{y}) \, e^{-\lambda \Delta t_{\mathbf{xy}}} \leq e^{-\lambda \Delta t_{\mathbf{xy}}}.$$

Thus, $\Delta t_{\mathbf{xy}} > \lambda^{-1} \log \theta^{-1}$ implies $\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) < \theta$, and as a result a given vector cannot be similar to any vector that arrived more than

$$\tau = \frac{1}{\lambda} \log \frac{1}{\theta}$$

time units earlier. Consequently, we can safely prune vectors that are older than τ. We call τ the *time horizon*.

Parameter setting. The SSSJ problem, defined in Problem 1, requires two parameters: a similarity threshold θ and a time-decay factor λ. The time-filtering property suggests a simple methodology to set these two parameters:

1. Select θ as the lowest value of the similarity between two *simultaneously-arriving* vectors that are deemed *similar*.
2. Select τ as the smallest difference in arrival times between two *identical* vectors that are deemed *dissimilar*.
3. Set $\lambda = \tau^{-1} \log \theta^{-1}$.

Setting θ and τ in steps 1 and 2 depends on the application.

Additional notation. When referring to the non-streaming setting, we consider a dataset $\mathcal{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ consisting of n vectors in $\mathbb{R}^d$.

Following Anastasiu and Karypis, we use the notation $\mathbf{x}' = \mathbf{x}'_p = \langle \mathrm{x}_1, .., \mathrm{x}_{p-1}, 0, .., 0 \rangle$ and to denote the *prefix* of a vector $\mathbf{x}$. For a vector $\mathbf{x}$, we denote by $\mathrm{vm}_{\mathbf{x}}$ its maximum coordinate, by $\Sigma_{\mathbf{x}} = \sum_j \mathrm{x}_j$ the sum of its coordinates, and by $|\mathbf{x}|$ the number of its non-zero coordinates.

Given a static dataset $\mathcal{D}$ we use m_j to refer to the maximum value along the j-th coordinate over all vectors in $\mathcal{D}$. All values of m_j together compose the vector $\mathbf{m}$. Our methods use indexing schemes which build an index incrementally, vector-by-vector. We use $\widehat{\mathbf{m}}$ to refer to the vector $\mathbf{m}$, restricted to the dataset that is already indexed, and we write $\widehat{\mathrm{m}}_j$ for its j-th coordinate. Furthermore, we use $\widehat{\mathbf{m}}^{\lambda}$ (and $\widehat{\mathrm{m}}_j^{\lambda}$ for its j-th coordinate) to denote a time-decayed variant of $\widehat{\mathbf{m}}$, whose precise definition is given in Section 5.3.

4. OVERVIEW OF THE APPROACH

In this section we present a high-level overview of our main algorithms for the SSSJ problem.

We present two different algorithmic frameworks: `MB-IDX` (MiniBatch-`IDX`) and `STR-IDX` (Streaming-`IDX`), where `IDX` is an indexing method for the APSS problem on static data. To make the presentation of our algorithms more clear, we first give an overview of the indexing methods for the static APSS problem, as introduced in earlier papers [3, 7, 10].

All indexing schemes are based on building an *inverted index*. The index is a collection of d *posting lists*, $\mathcal{I} = \{I_1, \dots, I_d\}$, where the list I_j contains all pairs $(\iota(\mathbf{x}), \mathrm{x}_j)$ such that the j-th coordinate of vector $\mathbf{x}$ is non-zero, i.e., $\mathrm{x}_j \neq 0$. Here $\iota(\mathbf{x})$ denotes a reference to vector $\mathbf{x}$.

Some methods optimize computation by not indexing the whole dataset. In these cases, the un-indexed part is still needed in order to compute exact similarities. Thus, we assume that a separate part of the index $\mathcal{I}$ contains the un-indexed part of the dataset, called *residual direct index* $\mathcal{R}$.

All schemes build the index incrementally, while also computing similar pairs. In particular, we start with an empty index, and we iteratively process the vectors in $\mathcal{D}$. For each newly-processed vector $\mathbf{x}$, we compute similar pairs $(\mathbf{x}, \mathbf{y})$ for all vectors $\mathbf{y}$ that are already in the index. Thereafter, we add (some of) the non-zero coordinates of $\mathbf{x}$ to the index.

Building the index and computing similar pairs can be seen as a three-phase process:

index construction (`IC`): adds new vectors to the index $\mathcal{I}$.

candidate generation (`CG`): uses the index $\mathcal{I}$ to generate candidate similar pairs. The candidate pairs may contain false positives but no false negatives.

Algorithm 1: MB-IDX (MiniBatch with index IDX)

```
input : Data stream $\mathcal{S}$, threshold $\theta$, decay $\lambda$
output: All pairs $\mathbf{x}, \mathbf{y} \in \mathcal{S}$ s.t. $\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) \geq \theta$
1  $\mathcal{I} \leftarrow \varnothing$
2  $t_0 \leftarrow 0; t \leftarrow 0$
3  $\tau \leftarrow \lambda^{-1} \log \theta^{-1}$
4  while true do
5      $W \leftarrow \varnothing$
6      $t_0 \leftarrow t_0 + \tau$
7      while $t \leq t_0 + \tau$ do
8          $\mathbf{x} \leftarrow$ read($\mathcal{S}$)
9          $t \leftarrow t(\mathbf{x})$
10         $C \leftarrow$ CandGen-IDX($\mathcal{I}, \mathbf{x}, \theta$)
11         $P \leftarrow$ CandVer-IDX($\mathcal{I}, \mathbf{x}, C, \theta$)
12         report ApplyDecay($P, \lambda$)
13         $W \leftarrow W \cup \{\mathbf{x}\}$
14     $(\mathcal{I}, P) \leftarrow$ IndConstr-IDX($W, \theta$)
15     report ApplyDecay($P, \lambda$)
```

candidate verification (CV): computes true similarities between candidate pairs, and reports true similar pairs, while dismissing false positives.

More concretely, for an indexing scheme IDX we assume that the following three primitives are available, which correspond to the three phases outlined above:

$(\mathcal{I}, P) \leftarrow$ IndConstr-IDX($\mathcal{D}, \theta$): Given a dataset $\mathcal{D}$ consisting of n vectors, and a similarity threshold θ, the function IndConstr-IDX returns in $P = \{(\mathbf{x}, \mathbf{y})\}$ all similar pairs $(\mathbf{x}, \mathbf{y})$, with $\mathbf{x}, \mathbf{y} \in \mathcal{D}$. Additionally, IndConstr-IDX builds an index $\mathcal{I}$, which can be used to find similar pairs between the vectors in $\mathcal{D}$ and another query vector $\mathbf{z}$.

$C \leftarrow$ CandGen-IDX($\mathcal{I}, \mathbf{x}, \theta$): Given an index $\mathcal{I}$, built on a dataset $\mathcal{D}$, a vector $\mathbf{x}$, and a similarity threshold θ, the function CandGen-IDX returns a set of candidate vectors $C = \{\mathbf{y}\}$, which is a superset of all vectors that are similar to $\mathbf{x}$.

$P \leftarrow$ CandVer-IDX($\mathcal{I}, \mathbf{x}, C, \theta$): Given an index $\mathcal{I}$, built on a dataset $\mathcal{D}$, a vector $\mathbf{x}$, a set of candidate vectors C, and a similarity threshold θ, the function CandVer-IDX returns the set $P = \{(\mathbf{x}, \mathbf{y})\}$ of true similar pairs.

Both proposed frameworks, MB-IDX and STR-IDX, rely on an indexing scheme IDX, and adapt it to the streaming setting by augmenting it with the time-filtering property. The difference between the two frameworks is in how this adaptation is done. MB-IDX uses IDX as a "black box": it uses time filtering in order to build independent instances of IDX indexes, and drops them when they become obsolete. Conversely, for STR-IDX we opportunely modify the indexing method IDX by directly applying time filtering.

The presentation of the STR-IDX framework is deferred to the next section, following the details of the specific indexing methods IDX we use. The MB-IDX framework is presented below, as we only need to know the specifications of an index IDX, not its internal workings. In particular, we only assume the three primitives, IndConstr-IDX, CandGen-IDX, and CandVer-IDX, are available.

The MB-IDX framework works in time intervals of duration τ. During the k-th time interval it reads all vectors from the stream, and stores them in a buffer W. At the end of the time interval, it invokes IndConstr-IDX to report all similar pairs in W, and to build an index $\mathcal{I}$ on W. During the $(k+1)$-th time interval, the buffer W is reset and used to store the new vectors from the stream. At the same time, MB-IDX queries $\mathcal{I}$ with each vector $\mathbf{x}$ read from the stream, to find similar pairs between $\mathbf{x}$ and vectors in the previous time interval. Similar pairs are computed by invoking the functions CandGen-IDX and CandVer-IDX. At the end of the $(k+1)$-th time interval, the index $\mathcal{I}$ is replaced with a new index on the vectors in the $(k+1)$-th time interval.

MB-IDX guarantees that all pairs of vectors whose time difference is smaller than τ are tested for similarity. Thus, due to the time-filtering property, it returns the complete set of similar pairs. Algorithm 1 shows pseudocode for MB-IDX.

One drawback of MB-IDX is that it reports some similar pairs with a delay. In particular, all similar pairs that span across two time intervals are reported after the end of the first interval. In applications that require to report similar pairs as soon as both vectors are present, this behavior is undesirable. Moreover, to guarantee correctness, MB-IDX needs to tests pairs of vectors whose time difference is as large as 2τ, which can be pruned only after they are reported by the indexing scheme IDX, thus wasting computational power.

5. FILTERING FRAMEWORK

We now review the main indexing schemes used for the APSS problem. For each indexing scheme IDX, we describe its three phases (IC, CG, CV), and discuss how to adapt it to the streaming setting, giving the STR-IDX algorithm. One of our main contributions is the adaptation of well-known algorithms for the batch case to the streaming setting. To make the paper self-contained, in each of the following subsections we first present the indexing schemes in the static case (state of the art) and how they are used in the MB framework, and then we describe their adaptation to the STR framework (contribution of this paper). Furthermore, section 5.4 describes our improved ℓ_2-based indexing scheme.

For all the indexing schemes we present, except INV, we provide pseudocode. Due to lack of space, and also to highlight the differences among the indexing schemes, we present our pseudocode using a color convention:

– L2AP index: all lines are included (black, red, and green).
– AP index: red lines are included; green lines excluded.
– L2 index: green lines are included; red lines excluded.

5.1 Inverted index

The first scheme is a simple *inverted index* (INV) with no index-pruning optimizations. As with all indexing schemes, the main observation is that for two vectors to be similar, they need to have at least one common coordinate. Thus, two similar vectors can be found together in some list I_j.

In the IC phase, for each new vector $\mathbf{x}$, all its coordinates x_j are added to the index. In the CG phase, given the query vector $\mathbf{x}$, we use the index $\mathcal{I}$ to retrieve candidate vectors similar to $\mathbf{x}$. In particular, the candidates $\mathbf{y}$ are all vectors in the posting lists where $\mathbf{x}$ has non-zero coordinates.

The result of the CG phase is the exact similarity score between $\mathbf{x}$ and each candidate vector $\mathbf{y}$. Therefore, the CV phase simply applies the similarity threshold θ and reports the true similar pairs.

MB **framework (**MB-INV**):** The functions IndConstr-INV, CandGen-INV, and CandVer-INV, needed to specify the MB-INV algorithm follow directly from the discussion above. Since they are rather straightforward, we omit further details.

Algorithm 2: `IndConstr-L2AP`

```
input : Dataset $\mathcal{D}$, threshold $\theta$
output: Similarity index $\mathcal{I}$ on $\mathcal{D}$, and
        set of pairs $P = \{(\mathbf{x},\mathbf{y}) \mid \mathrm{sim}(\mathbf{x},\mathbf{y}) \geq \theta\}$
1  $\mathcal{I} \leftarrow \varnothing$
2  $P \leftarrow \varnothing$
3  foreach $\mathbf{x} \in \mathcal{D}$ do
4      $C \leftarrow$ CandGen-L2AP$(\mathcal{I}, \mathbf{x}, \theta)$
5      $P \leftarrow P \cup$ CandVer-L2AP$(\mathcal{I}, \mathbf{x}, C, \theta)$
6      $\mathrm{b}_1 \leftarrow 0$
7      $\mathrm{b}_2 \leftarrow 0$; $\mathrm{b}_t \leftarrow 0$
8      foreach $j = 1 \ldots d$ s.t. $\mathrm{x}_j > 0$ do
9          pscore $\leftarrow \min\{\mathrm{b}_1, \mathrm{b}_2\}$
10         $\mathrm{b}_1 \leftarrow \mathrm{b}_1 + \mathrm{x}_j \, \min\{\mathrm{m}_j, \mathrm{vm}_\mathbf{x}\}$
11         $\mathrm{b}_t \leftarrow \mathrm{b}_t + {\mathrm{x}_j}^2$; $\mathrm{b}_2 \leftarrow \sqrt{\mathrm{b}_t}$
12         if $\min\{\mathrm{b}_1, \mathrm{b}_2\} \geq \theta$ then
13             if $\mathcal{R}[\iota(\mathbf{x})] = \varnothing$ then
14                 $\mathcal{R}[\iota(\mathbf{x})] \leftarrow \mathbf{x}'_j$
15                 $Q[\iota(\mathbf{x})] \leftarrow$ pscore
16             $I_j \leftarrow I_j \cup \{(\iota(\mathbf{x}), \mathrm{x}_j, ||\mathbf{x}'_j||)\}$
17 return $(\mathcal{I}, P)$
```

STR framework (STR-INV): The description of INV given above considers the APSS problem, i.e., a static dataset $\mathcal{D}$. We now consider a data stream $\mathcal{S}$, where each vector $\mathbf{x}$ in the stream is associated with a timestamp $t(\mathbf{x})$. We apply the INV scheme by adding the data items in the index in the order they appear in the stream. The lists I_j of the index keep pairs $(\iota(\mathbf{x}), \mathrm{x}_j)$ ordered by timestamps $t(\mathbf{x})$. Maintaining a time-respecting order inside the lists is easy. We process the data in the same order, so we can can just append new vector coordinates at the end of the lists.

Before adding a new item $\mathbf{x}$ to the index, we use the index to retrieve all the earlier vectors $\mathbf{y}$ that are Δt-similar to $\mathbf{x}$. Due to the time-filtering property, vectors that are older than τ cannot be similar to $\mathbf{x}$. This observation has two implications: (i) we can stop retrieving candidate vectors from a list I_j upon encountering a vector that is older than τ; and (ii) when encountering such a vector, the part of the list preceding it can be pruned away.

5.2 All-pairs indexing scheme

The AP [7] scheme improves over the simple INV method by reducing the size of the index. When using AP, not all the coordinates of a vector $\mathbf{x}$ need to be indexed, as long as it can be guaranteed that $\mathbf{x}$ and all its similar vectors $\mathbf{y}$ share at least one common coordinate in the index $\mathcal{I}$.

Similarly to the INV scheme, AP incrementally builds an index by processing one vector at a time.

In the IC phase (function IndConstr-AP, shown as Algorithm 2, including red lines and excluding green), for each new vector $\mathbf{x}$, the algorithm scans its coordinates in a predefined order. It keeps a score pscore, which represents an upper bound on the similarity between a prefix of $\mathbf{x}$ and any other vector in the dataset. To compute the upper bound, AP uses the vector $\mathbf{m}$, which keeps the maximum of each coordinate in the dataset. As long as pscore is smaller than the threshold θ, given that the similarity of $\mathbf{x}$ to any other vector cannot exceed θ, the coordinates of $\mathbf{x}$ scanned so far can be omitted from the index without the danger of missing any similar pair. As soon as pscore exceeds θ, the remain-

Algorithm 3: `CandGen-L2AP`

```
input : Index $\mathcal{I}$, vector $\mathbf{x}$, threshold $\theta$
output: Accum. score array $C$ for candidate vectors
        (candidate set is $C = \{\mathbf{y} \in \mathcal{D} \mid C[\iota(\mathbf{y})] > 0\}$)
1  $C \leftarrow \varnothing$
2  $\mathrm{sz}_1 \leftarrow \theta / \mathrm{vm}_\mathbf{x}$
3  $\mathrm{rs}_1 \leftarrow \mathrm{dot}(\mathbf{x}, \widehat{\mathbf{m}})$
4  $\mathrm{rs}_2 \leftarrow 1$; $\mathrm{rs}_t \leftarrow 1$
5  foreach $j = d \ldots 1$ s.t. $\mathrm{x}_j > 0$ do //reverse order
6      foreach $(\iota(\mathbf{y}), \mathrm{y}_j, ||\mathbf{y}'_j||) \in I_j$ do
7          remscore $\leftarrow \min\{\mathrm{rs}_1, \mathrm{rs}_2\}$
8          if $|\mathbf{y}| \, \mathrm{vm}_\mathbf{y} \geq \mathrm{sz}_1$ then
9              if $C[\iota(\mathbf{y})] > 0$ or remscore $\geq \theta$ then
10                 $C[\iota(\mathbf{y})] \leftarrow C[\iota(\mathbf{y})] + \mathrm{x}_j \, \mathrm{y}_j$
11                 l2bound $\leftarrow C[\iota(\mathbf{y})] + ||\mathbf{x}'_j|| \, ||\mathbf{y}'_j||$
12                 if l2bound $< \theta$ then
13                     $C[\iota(\mathbf{y})] \leftarrow 0$
14     $\mathrm{rs}_1 \leftarrow \mathrm{rs}_1 - \mathrm{x}_j \, \widehat{\mathrm{m}}_j$
15     $\mathrm{rs}_t \leftarrow \mathrm{rs}_t - {\mathrm{x}_j}^2$; $\mathrm{rs}_2 \leftarrow \sqrt{\mathrm{rs}_t}$
16 return $C$;
```

Algorithm 4: `CandVer-L2AP`

```
input : Index $\mathcal{I}$, vector $\mathbf{x}$, candidate vector array $C$,
        threshold $\theta$
output: Set of pairs $P = \{(\mathbf{x},\mathbf{y}) \mid \mathrm{sim}(\mathbf{x},\mathbf{y}) \geq \theta\}$
1  $P \leftarrow \varnothing$
2  foreach $\mathbf{y}$ s.t. $C[\iota(\mathbf{y})] > 0$ do
3      $\mathrm{ps}_1 \leftarrow C[\iota(\mathbf{y})] + Q[\iota(\mathbf{y})]$
4      $\mathrm{ds}_1 \leftarrow C[\iota(\mathbf{y})] + \min\{\mathrm{vm}_\mathbf{x} \, \Sigma_{\mathbf{y}'}, \mathrm{vm}_{\mathbf{y}'} \, \Sigma_\mathbf{x}\}$
5      $\mathrm{sz}_2 \leftarrow C[\iota(\mathbf{y})] + \min\{|\mathbf{x}|, |\mathbf{y}'|\} \, \mathrm{vm}_\mathbf{x} \, \mathrm{vm}_{\mathbf{y}'}$
6      if $\mathrm{ps}_1 \geq \theta$ and $\mathrm{ds}_1 \geq \theta$ and $\mathrm{sz}_2 \geq \theta$) then
7          $s \leftarrow C[\iota(\mathbf{y})] + \mathrm{dot}(\mathbf{x}, \mathbf{y}')$
8          if $(s \geq \theta)$ then
9              $P \leftarrow P \cup \{(\mathbf{x}, \mathbf{y}, s)\}$
10 return $P$
```

ing suffix of $\mathbf{x}$ is added to the index, and the prefix $\mathbf{x}'$ is saved in the residual direct index $\mathcal{R}$.

In the CG phase (function CandGen-AP, shown as Algorithm 3, including red lines and excluding green), AP uses a lower bound sz_1 for the size of any vector $\mathbf{y}$ that is similar to $\mathbf{x}$, so that vectors that have too few non-zero entries can be ignored (line 8). Additionally, it uses a variable rs_1 to keep an upper bound on the similarity between $\mathbf{x}$ and any other vector $\mathbf{y}$. This upper bound is computed by using the residual direct index $\mathcal{R}$ and the already accumulated dot-product, and is updated as the algorithm processes the different posting lists. When the upper bound becomes smaller than θ, vectors that have not already been added to the set of candidates can be ignored (line 9). The array C holds the candidate vectors, together with the partial dot-product that is due to the indexed part of the vectors.

Finally, in the CV phase (CandVer-AP function, shown as Algorithm 4), we compute the final similarities by using the residual index $\mathcal{R}$, and report the true similar pairs.

The streaming versions of AP, in both MB and STR frameworks, are not efficient in practice, and thus, we omit further details. Instead, we proceed presenting the next scheme, the L2AP index, which is a generalization of AP.

5.3 L2-based indexing scheme

L2AP [3] is the state-of-the-art for computing similarity self-join. The scheme uses tighter ℓ_2-bounds, which reduce the size of the index, but also the number of generated candidates and the number of fully-computed similarities.

The L2AP scheme primarily leverages the Cauchy-Schwarz inequality, which states that $\mathrm{dot}(\mathbf{x}, \mathbf{y}) \le ||\mathbf{x}||\,||\mathbf{y}||$. The same bound applies when considering a prefix of a query vector $\mathbf{x}$. Since we consider unit-normalized vectors ($||\mathbf{y}|| = 1$),

$$\mathrm{dot}(\mathbf{x}', \mathbf{y}) \le ||\mathbf{x}'||\,||\mathbf{y}|| \le ||\mathbf{x}'||.$$

The previous bound produces a tighter value for pscore, which is used in the IC phase to bound the similarity of the vector currently being indexed, to the rest of the dataset. In particular, L2AP sets $\texttt{pscore} = \min\{\texttt{pscore}_{\texttt{AP}}, ||\mathbf{x}'||\}$.

Additionally, the L2AP index stores the value of pscore computed when $\mathbf{x}$ is indexed. L2AP keeps these values in an array Q, index by $\iota(\mathbf{x})$, as shown in line 15 of Algorithm 2. The L2AP index also stores in the index the magnitude of the prefix of each vector $\mathbf{x}$, up to coordinate j. That is, the entries of the posting lists are now triples of the type $(\iota(\mathbf{x}), \mathrm{x}_j, ||\mathbf{x}'_j||)$.

Both pieces of additional information, the array Q and the values $||\mathbf{x}_j||$ stored in the posting lists, are used during the CG phase to reduce the number of candidates.

In the CG phase, for a given query vector $\mathbf{x}$, we scan its coordinates backwards, i.e., in reverse order with respect to the one used during indexing, and we accumulate similarity scores for suffixes of $\mathbf{x}$. We keep a remscore bound on the remaining similarity score, which combines the rs_1 bound used in AP and a new ℓ_2-based rs_2 bound that uses the prefix magnitude values ($||\mathbf{y}'_j||$) stored in the posting lists. The remscore bound is an upper bound on the similarity of the prefix of the current query vector and any other vector in the index, thus as long as remscore is smaller than θ, the algorithm can prune the candidate.

Pseudocode for the three phases of L2AP, IC, CG, and CV, is shown in Algorithms 2, 3, and 4, respectively, including both red and green lines. More details on the scheme can be found in the original paper of Anastasiu and Karypis [3].

MB framework (MB-L2AP): As before, the MB-L2AP algorithm is a direct instantiation of the generic Algorithm 1, using the functions that implement the three different phases of L2AP.

STR framework (STR-L2AP): To describe the modifications required for adapting the L2AP scheme in the streaming framework, we need to introduce some additional notation.

First, we assume that the input is a stream $\mathcal{S}$ of vectors. The main difference is that now both $\mathbf{m}$ and $\widehat{\mathbf{m}}$ are function of time, as the maximum values in the stream evolve over time. We adapt the use of the two vectors to the streaming case differently.

The vector $\widehat{\mathbf{m}}$ is used in the CG phase to generate candidate vectors. Given that we are looking at vectors that are already indexed (i.e., in the *past*), it is possible to apply the definition of $\mathrm{sim}_{\Delta t}$ between the query vector $\mathbf{x}$ and $\widehat{\mathbf{m}}$. In particular, given that the coordinates of $\widehat{\mathbf{m}}$ originate from different vectors in the index, we can apply the decay factor to each coordinate to $\widehat{\mathbf{m}}$ separately.

More formally, let $\widehat{\mathbf{m}}^{\lambda}(t)$ be the *worst case indexed vector* at time t, that is, $\widehat{\mathbf{m}}^{\lambda}(t)$ is the representation of the vector in the index that is most Δt-similar to any vector in $\mathcal{S}$ arriving at time t. Its j-th coordinate $\widehat{\mathrm{m}}_j^{\lambda}(t)$ is given by

$$\widehat{\mathrm{m}}_j^{\lambda}(t) = \max_{\substack{\mathbf{x}\in\mathcal{S}\\ t(\mathbf{x})\le t}} \left\{ \mathrm{x}_j\, e^{-\lambda|t-t(\mathbf{x})|} \right\}.$$

To simplify the notation, we omit the dependency from time when obvious from the context, and write simply $\widehat{\mathbf{m}}^{\lambda}$.

An upper bound on the time-dependent cosine similarity of any newly arrived vector $\mathbf{x}$ at time t can be obtained by

$$\texttt{remscore}(t) = \mathrm{dot}(\mathbf{x}, \widehat{\mathbf{m}}^{\lambda}) = \sum_{j=1}^{d} \mathrm{x}_j\, \widehat{\mathrm{m}}_j^{\lambda}.$$

Conversely, the vector $\mathbf{m}$ is used in the IC phase to decide which coordinates to add to the index. Its purpose is to guarantee that any two similar vectors in the stream share at least a coordinate. As such, it needs to look at the *future*. In a static setting, and in MB, maximum values in the whole dataset can easily be accumulated beforehand. However, in a streaming setting these maximum values need to be kept online. This difference implies that when the vector $\mathbf{m}$ changes, the invariant on which the AP index is built is lost, and we need to restore it. We call this process *re-indexing*.

At a first glance, it might seem straightforward to use $\mathrm{sim}_{\Delta t}$ to decide what to index, i.e., adding the decay factor to line 10 in Algorithm 2. However, the decay factor causes $\mathbf{m}$ to change rapidly, which in turn leads to a larger number of re-indexings. Re-indexing, as explained next, is an expensive operation. Therefore, we opt to avoid applying the time decay when computing the b_1 bound with $\mathbf{m}$.

Algorithm 5: STR-IDX (Streaming with index IDX)

```
input : Data stream $\mathcal{S}$, threshold $\theta$, decay $\lambda$
output: All pairs $\mathbf{x}, \mathbf{y} \in \mathcal{S}$ s.t. $\mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) \ge \theta$
1 $\mathcal{I} \leftarrow \varnothing$
2 $P \leftarrow \varnothing$
3 while true do
4     $\mathbf{x} \leftarrow$ read($\mathcal{S}$)
5     $(\mathcal{I}, P) \leftarrow$ IndConstr-IDX-STR($\mathcal{I}, \mathbf{x}, \theta, \lambda$)
6     report $P$
```

Re-indexing (IC). For each coordinate x_j of a newly arrived vector $\mathbf{x}$, one of three cases may occurr.

– $\mathrm{x}_j = 0$: no action is needed.

– $0 < \mathrm{x}_j \le \mathrm{m}_j$: the posting list I_j can be pruned via time filtering, as explained next.

– $\mathrm{x}_j > \mathrm{m}_j$: the upper bound vector $\mathbf{m}$ needs to be updated with the new value just read, i.e., $\mathrm{m}_j \leftarrow \mathrm{x}_j$. In this case, parts of the pruned vectors may need to be re-indexed.

When $\mathbf{m}$ is updated ($\mathrm{x}_j > \mathrm{m}_j$) the invariant of prefix filtering does not hold anymore. That is, there may be vectors similar to $\mathbf{x}$ that are not matched in $\mathcal{I}$. To restore the invariant, we re-scan $\mathcal{R}$ (un-indexed part of vectors within horizon τ). If the similarity of a vector $\mathbf{y} \in \mathcal{R}$ and $\mathbf{m}$ has increased, we will reach the threshold θ while scanning the prefix $\mathbf{y}'_p$ (before reaching its end). Therefore, we just need to index those coordinates $\mathrm{y}_{p'} < \mathrm{y}_j \le \mathrm{y}_p$, where p' is the newly computed boundary.

Given that the similarity can increase only for vectors that have non-zero value in the dimensions of $\mathbf{m}$ that got updated, we can keep an inverted index of $\mathcal{R}$ to avoid scanning every vector. We use the updated dimensions of the max

Algorithm 6: `IndConstr-L2AP-STR`

```
input : Index $\mathcal{I}$, vector $\mathbf{x}$, threshold $\theta$, decay $\lambda$
output: Updated similarity index $\mathcal{I}$ including $\mathbf{x}$, and
        set of pairs $P = \{(\mathbf{x}, \mathbf{y}) \mid \mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) \geq \theta\}$
$\mathrm{b}_1 \leftarrow +\infty$; $\mathrm{b}_2 \leftarrow +\infty$
$\mathrm{b}_1 \leftarrow 0$
$\mathrm{b}_2 \leftarrow 0; \mathrm{b}_t \leftarrow 0$
$C \leftarrow$ CandGen-L2AP-STR$(\mathcal{I}, \mathbf{x}, \theta, \lambda)$
$P \leftarrow$ CandVer-L2AP-STR$((\mathcal{I}, \mathbf{x}, C, \theta, \lambda)$
foreach $j = 1 \ldots d$ s.t. $\mathrm{x}_j > 0$ do
    pscore $\leftarrow \min\{\mathrm{b}_1, \mathrm{b}_2\}$
    $\mathrm{b}_1 \leftarrow \mathrm{b}_1 + \mathrm{x}_j \min\{\mathrm{m}_j, \mathrm{vm}_{\mathbf{x}}\}$
    $\mathrm{b}_t \leftarrow \mathrm{b}_t + {\mathrm{x}_j}^2; \mathrm{b}_2 \leftarrow \sqrt{\mathrm{b}_t}$
    if $\min\{\mathrm{b}_1, \mathrm{b}_2\} \geq \theta$ then
        if $\mathcal{R}[\iota(\mathbf{x})] = \varnothing$ then
            $\mathcal{R}[\iota(\mathbf{x})] \leftarrow \mathbf{x}'_j$
            $Q[\iota(\mathbf{x})] \leftarrow$ pscore
        $I_j \leftarrow I_j \cup \{(\iota(\mathbf{x}), \mathrm{x}_j, ||\mathbf{x}'_j||)\}$
return $(\mathcal{I}, P)$
```

vector m_j to select the possible candidates for re-indexing, and then scan only those from the residual index $\mathcal{R}$.

Note that re-indexing inserts older vectors in the index. As the lists always append items at the end, re-indexing introduces out-of-order items in the list. As explained in Section 6, the loss of the time-ordered property hinders one of the optimizations in pruning the index based on time filtering (explained next).

Time filtering (`CG`). To avoid continuously scanning the index, we adopt a lazy approach. The posting lists I_j are (partially) sorted by time, i.e., newest items are always appended to the tail of the lists. Thus, a simple linear scan from the head of the lists is able to prune the lists lazily while accumulating the similarity in the `CG` phase. More specifically, when a new vector $\mathbf{x}$ arrives, we scan all the posting lists I_j such that $\mathrm{x}_j \neq 0$, in order to generate its candidates. While scanning the lists, we drop any item $(\iota(\mathbf{y}), \mathrm{y}_j, ||\mathbf{y}'_j||)$ such that $|t(\mathbf{x}) - t(\mathbf{y})| > \tau$.

The main loop of `STR-L2AP` is shown in Algorithm 5. The algorithm simply reads each item $\mathbf{x}$ in the stream, adds $\mathbf{x}$ to the index, and computes the vectors that are similar to it, by calling `IndConstr-L2AP-STR` (Algorithm 6, including both red and green lines).

5.4 Improved L2-based indexing scheme

The last indexing scheme we present, `L2`, is an adaptation of `L2AP`, optimized for stream data.

The idea for the improved index is based on the observation that `L2AP` combines a number of different bounds: the bounds inherited from the `AP` scheme (e.g., b_1 for index construction, and rs_1 for candidate generation) and the new ℓ_2-based bounds introduced by Anastasiu and Karypis [3] (e.g., b_2 for index construction, and rs_2 and `l2bound` for candidate generation).

We have observed that in general the ℓ_2-based bounds are more effective than the `AP`-based bounds. In almost all cases the ℓ_2-based bounds are the ones that trigger. This observation is also verified by the results of Anastasiu and Karypis [3]. Furthermore, as can be easily seen (by inspecting the red and green lines of Algorithms 2, 3, and 4), while the `AP` bounds use statistics of the data in the index, the ℓ_2-based

Algorithm 7: `CandGen-L2AP-STR`

```
input : Index $\mathcal{I}$, vector $\mathbf{x}$, threshold $\theta$, decay $\lambda$
output: Accum. score array $C$ for candidate vectors
        (candidate set is $C = \{\mathbf{y} \in \mathcal{D} \mid C[\iota(\mathbf{y})] > 0\}$)
$C \leftarrow \varnothing$
$\mathrm{rs}_1 \leftarrow +\infty$; $\mathrm{rs}_2 \leftarrow +\infty$
$\mathrm{rs}_1 \leftarrow \mathrm{dot}(\mathbf{x}, \widehat{\mathbf{m}}^{\lambda})$
$\mathrm{rs}_2 \leftarrow 1; \mathrm{rs}_t \leftarrow 1$
foreach $j = d \ldots 1$ s.t. $\mathrm{x}_j > 0$ do //reverse order
    foreach $(\iota(\mathbf{y}), \mathrm{y}_j, ||\mathbf{y}'_j||) \in I_j$ s.t. $\Delta t_{\mathbf{xy}} \leq \tau$ do
        remscore $\leftarrow \min\{\mathrm{rs}_1, \mathrm{rs}_2\, e^{-\lambda \Delta t_{\mathbf{xy}}}\}$
        if $(C[\iota(\mathbf{y})] > 0$ or remscore $\geq \theta)$ then
            $C[\iota(\mathbf{y})] \leftarrow C[\iota(\mathbf{y})] + \mathrm{x}_j\, \mathrm{y}_j$
            l2bound $\leftarrow (C[\iota(\mathbf{y})] + ||\mathbf{x}'_j||\, ||\mathbf{y}'_j||)\, e^{-\lambda \Delta t_{\mathbf{xy}}}$
            if l2bound $< \theta$ then
                $C[\iota(\mathbf{y})] \leftarrow 0$
    $\mathrm{rs}_1 \leftarrow \mathrm{rs}_1 - \mathrm{x}_j\, \widehat{\mathrm{m}}_j^{\lambda}$
    $\mathrm{rs}_t \leftarrow \mathrm{rs}_t - {\mathrm{x}_j}^2; \mathrm{rs}_2 \leftarrow \sqrt{\mathrm{rs}_t}$
return $C$;
```

Algorithm 8: `CandVer-L2AP-STR`

```
input : Index $\mathcal{I}$, vector $\mathbf{x}$, candidate vector array $C$,
        threshold $\theta$, decay $\lambda$
output: Set of pairs $P = \{(\mathbf{x}, \mathbf{y}) \mid \mathrm{sim}_{\Delta t}(\mathbf{x}, \mathbf{y}) \geq \theta\}$
$P \leftarrow \varnothing$
foreach $\mathbf{y}$ s.t. $C[\iota(\mathbf{y})] > 0$ do
    $\mathrm{ps}_1 \leftarrow (C[\iota(\mathbf{y})] + Q[\iota(\mathbf{y})])\, e^{-\lambda \Delta t_{\mathbf{xy}}}$
    $\mathrm{ds}_1 \leftarrow (C[\iota(\mathbf{y})] + \min\{\mathrm{vm}_{\mathbf{x}}\, \Sigma_{\mathbf{y}'}, \mathrm{vm}_{\mathbf{y}'}\, \Sigma_{\mathbf{x}}\})\, e^{-\lambda \Delta t_{\mathbf{xy}}}$
    $\mathrm{sz}_2 \leftarrow (C[\iota(\mathbf{y})] + \min\{|\mathbf{x}|, |\mathbf{y}'|\}\, \mathrm{vm}_{\mathbf{x}}\, \mathrm{vm}_{\mathbf{y}'})\, e^{-\lambda \Delta t_{\mathbf{xy}}}$
    if $\mathrm{ps}_1 \geq \theta$ and $\mathrm{ds}_1 \geq \theta$ and $\mathrm{sz}_2 \geq \theta)$ then
        $s \leftarrow C[\iota(\mathbf{y})] + \mathrm{dot}(\mathbf{x}, \mathbf{y}')$
        if $s \geq \theta$ then
            $P \leftarrow P \cup \{(\mathbf{x}, \mathbf{y}, s)\}$
return $P$
```

bounds depend only on the vector being index. This implies that by using only the ℓ_2-based bounds, one does not need to maintain the worst case vector $\mathbf{m}(t)$, and thus, no re-indexing is required.

Thus, the `L2` index uses only the ℓ_2-based bounds and discards the `AP`-based bounds. The static version of `L2` (used in `MB`) is shown at Algorithms 2, 3, and 4, including the green lines and excluding the red ones. The main loop for the streaming case is `IndConstr-L2-STR`, shown as Algorithm 6, including the green lines and excluding the red ones.

6. IMPLEMENTATION

This section presents additional details and optimizations for both of our algorithmic frameworks, `MB` and `STR`.

6.1 Minibatch

Recall that in the `MB` framework, we construct the index for a set of data items that arrive within horizon τ. However, the `L2AP` scheme requires that we know not only the data used to construct the index, but also the data used to query the index, e.g., see Algorithm 2, line 10 This assumption does not hold for `MB`, as shown in Algorithm 1, as the data used to query the index arrive in the stream after the index has been constructed.

To address this problem, we modify the MB framework as follows. At any point of time we consider two windows, W_{k-1} and W_k, each of size τ, where W_k is the current window and W_{k-1} is the previous one. The input data are accumulated in the current window W_k, and the vector $\mathbf{m}$ is maintained. When we arrive at the end of the current window, we compute the global vector $\mathbf{m}$, defined over both W_{k-1} and W_k, by combining the $\mathbf{m}$ vectors of the two windows. We then use the data in W_{k-1} to build the index, and the data in W_k to query the index built over W_{k-1}.

When moving to the next window, W_{k+1} becomes current, W_k becomes previous, and W_{k-1} is dropped.

6.2 Streaming

Next we discuss implementation issues related to the streaming framework.

Variable-size lists. The streaming framework introduces posting lists of variable size. The size of the posting lists not only increases, but due to time filtering it may also decrease. In order to avoid many and small memory (de)allocations, we implement posting lists using a circular byte buffer. When the buffer becomes full we double its capacity, while when its size drops below 1/4 we halve it.

Time filtering. In the INV and L2 schemes it is easy to maintain the posting lists in time-increasing order. This ordering introduces a simple optimization: by scanning the posting lists backwards — from the newest to the oldest item — during candidate generation, it is possible to stop scanning and truncate the posting list as soon as we find the first item over the horizon τ. Truncating the circular buffer requires constant time (if no shrinking occurs).

In L2AP it is not possible to keep the posting lists in time order, as re-indexing may introduce out-of-order items. Thus, L2AP scans the lists forward and needs to go through all the expired items to prune the posting list.

Data structures. The residual direct index $\mathcal{R}$ and the Q array in L2AP and L2 need to be continuously pruned. To support the required operations for these data struncture, we implement them using a *linked hash-map*, which combines a hash-map for fast retrieval, and a linked list for sequential access. The sequential access is the order in which the data items are inserted in the data structure, which is also the time order. Maintaining these data structures requires amortized constant time, and memory linear in the number of vectors that arrive within a time interval τ.

Applying decay-factor pruning (λ). Central to adapting the indexing methods in the streaming setting, is the use of the decay factor λ for pruning. In order to make decay-factor pruning effective we apply the following principles.

Index construction: As the decay factor is used to prune candidates from the data seen in the past, decay-factor pruning is *never* applied during the index-construction phase.

Candidate generation: Decay-factor pruning is applied during candidate generation when computing the `remscore` bound (line 7 of Algorithm 7), as well as when applying the early ℓ_2 pruning (line 10). For L2AP, it is also applied when computing rs_1 (line 13), with a different decay factor for each coordinate (as specified in the definition of $\widehat{\mathbf{m}}^{\lambda}$).

Candidate verification: In the candidate-verification phase, decay-factor pruning is easily applied when computing all of the bounds (lines 3–5 of Algorithm 8).

7. EXPERIMENTAL EVALUATION

The methods presented in the previous sections combine four indexing schemes into two algorithmic frameworks (mini-batch vs. streaming). Altogether, they lead to a large number of trade-offs for increased pruning power vs. index maintainance. In order to better understand the behavior and evaluate the efficiency of each method we perform an extensive experimental study. The objective of our evaluation is to experimentally answer the following questions:

Q1: Which framework performs better, STR or MB?

Q2: How effective is L2, compared to L2AP and INV?

Q3: What are the effects of the parameters λ and θ?

Datasets. We test the proposed algorithms on several real-world datasets. RCV1 is the Reuters Corpus volume 1 dataset of newswires [18], WebSpam is a corpus of spam web pages [25], Blogs is a collection of one month of WordPress blog posts crawled in June 2015, and Tweets is a sample of tweets collected in June 2009 [29]. All datasets are available online in text format, while for the experiments we use a more compact and faster-to-read binary format; the text-to-binary converter is also included in the source code we provide. These datasets exhibit a wide variety in their characteristics, as summarized in Table 1, and allow us to evaluate our methods under very different scenarios. Specifically, we see that the density of the four datasets varies greatly. With respect to timestamps, items in WebSpam and RCV1 are assigned an artificial timestamp, sampled from a Poisson and a sequential arrival process, respectively. For Blogs and Tweets the publication time of each item is available, and we use it as a timestamp.

Algorithms. We test the two algorithmic frameworks, STR and MB, with the three index variants, INV, L2AP, and L2, on all four datasets. For the similarity threshold θ we explore a range of values in $[0.5, 0.99]$, which is typical for the APSS problem [3, 7], while for the time-decay factor λ we use exponentially increasing values in the range $[10^{-4}, 10^{-1}]$.

Our code also includes an implementation of AP for MB, but in preliminary experiments we found it much slower than L2AP, therefore we omit it from the set of indexing strategies under study. Some configurations are very expensive in terms of time or memory, and we are unable to run some of the algorithms. We set a timeout of 3 hours for each experiment, and we abort the run if this timout limit is exceeded, or if the JVM crashes because of lack of memory. In all cases of failure during our experiments, MB fails due to timeout, while STR because of memory requirements.

Setting. We run all the experiments on a computer with an Intel Xeon CPU E31230 @ 3.20 GHz with 8 MB of cache, 32 GB of RAM (of which 16 GB allocated for the JVM heap), and a 500 GB SATA disk. All experiments run sequentially on a single core of the CPU. We warm up the disk cache for each dataset by performing one run of the algorithm before taking running times. Times are averaged over three runs.

Table 1: Datasets used in the experimental evaluation. n: number of vectors; m: number of coordinates; $\sum|\mathbf{x}|$: number of non-zero coordinates; $\rho = {}^{\sum|\mathbf{x}|}/_{nm}$: density; $\overline{|\mathbf{x}|} = {}^{\sum|\mathbf{x}|}/_{n}$: average number of non-zero coordinates; and type of timestamps.

Dataset	n	m	$\sum\|\mathbf{x}\|$	ρ (%)	$\overline{\|\mathbf{x}\|}$	Timestamps
WebSpam	350 000	680 715	1305 M	0.55	3728.00	poisson
RCV1	804 414	43 001	61 M	0.18	75.72	sequential
Blogs	2 532 437	356 043	356 M	0.04	140.40	publishing date
Tweets	18 266 589	1 048 576	173 M	0.001	9.46	publishing date

Table 2: Fraction of the 24 configurations of (θ,λ) that successfully terminate within the allowed time budget (closer to 1.00 is better).

Dataset	MB			STR		
	INV	L2AP	L2	INV	L2AP	L2
WebSpam	1.00	1.00	1.00	1.00	0.83	0.96
RCV1	1.00	1.00	1.00	1.00	0.96	1.00
Blogs	0.25	0.25	0.25	1.00	0.96	1.00
Tweets	0.25	0.25	0.25	1.00	0.96	0.96

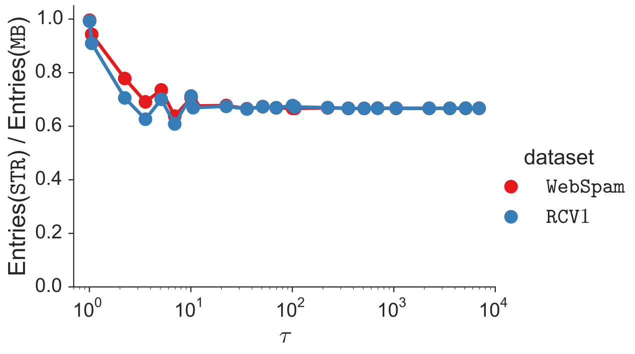

Figure 2: Ratio of index entries traversed during CG for STR compared to MB. For small τ, the ratio tends to one, while for larger horizons τ, STR needs to traverse only 65% of the entries compared to MB.

7.1 Results

Q1 (MB vs. STR): Of the four datasets we use, MB successfully runs with all configurations on only two of them (RCV1 and WebSpam). As can be seen from Table 1, Blogs and Tweets are the largest datasets, and MB is not able to scale to these sizes. On the other hand, STR is able to run with all configurations on all four datasets. The overhead of MB becomes too large when the horizon τ becomes small enough, as a new index needs to be initialized too frequently. Table 2 shows a summary of the outcomes of the experimental evaluations for the various configuration. Therefore, we restrict this comparison to RCV1 and WebSpam.

Most of the time of STR and MB, according to our profiling results, is spent in the CG phase while scanning the posting lists. Hence, we first compare the algorithms on the number of posting entries traversed. Figure 2 shows that STR usually does less total work, and traverses around 65% of the entries traversed by MB. The figure shows the ratio for L2, but other indexing strategies show the same trend (plots are omitted for brevity).

We now turn our attention to running time. Figures 3 and 4 compare the running time of STR and MB on RCV1 and WebSpam for all configuration on which both are able to run. The decay factor λ varies with the columns of the grid, while the indexing scheme with the rows. The two datasets present a different picture.

Clearly, for RCV1 (Figure 3) algorithm STR is faster than MB in most cases. More aggressive pruning reduces the difference, while in some cases with low θ (useful for recommender systems) algorithm STR can be up to 4 times faster than MB. L2AP is the exception, and for smaller τ its performance is subpar, as detailed next.

Conversely, for WebSpam the MB algorithm has the upper hand in most cases, especially for larger decay factors (i.e., shorter horizons). The different behavior is caused by the higher density of WebSpam, which has an average number of non-zero coordinates per vector, which is almost two orders of magnitude larger than RCV1 (see Table 1). For STR, this high density renders the lazy update approach inefficient, as a large number of posting lists need to be updated and pruned for each vector, especially for short horizons. MB can simply throw away old indexes, rather than mending them.

Summarizing our findings related to the MB-vs.-STR study, we conclude that STR is more efficient in most cases, as it is able to run on all datasets, while MB becomes too slow for larger datasets. For very specific conditions of high density, MB has a small advantage on STR.

Q2 (Indexing schemes): Now that we have established that STR scales better than MB, we focus on the former algorithm: the rest of the experiments are performed on STR. We turn our attention to the effectiveness of pruning, and compare L2 with INV and L2AP. For brevity, we show results for RCV1 only. The other datasets follow the same trends.

Figure 5 shows the comparison in time. Several interesting patterns emerge. First, L2 is almost always the fastest indexing scheme. Second, INV works well for short horizons, where the overheads incurred by pruning are not compensated by a large reduction in candidates. Instead, for larger horizons (low θ and λ) the gains of pruning are more pronounced, thus making INV a poor choice. Finally, L2AP is slightly slower than L2 in most cases, and much slower when the horizon is short. Even though L2 uses a subset of the bounds of L2AP, the overhead of computing the AP bounds offsets any possible gain in pruning.

In addition, L2AP may need to re-index residuals of vectors, and a shorter horizon causes more frequent re-indexings. Therefore, the two rightmost plots show an upward trend in L2AP, which is due to the re-indexing overhead. The overhead is so large that we were not able to run L2AP for $\theta = 0.99$ and $\lambda = 0.1$. While there are possible practical workarounds (e.g., use a more lax bound to decrease the frequency of re-indexing), L2 always provides a better choice and does not require tinkering with the algorithm.

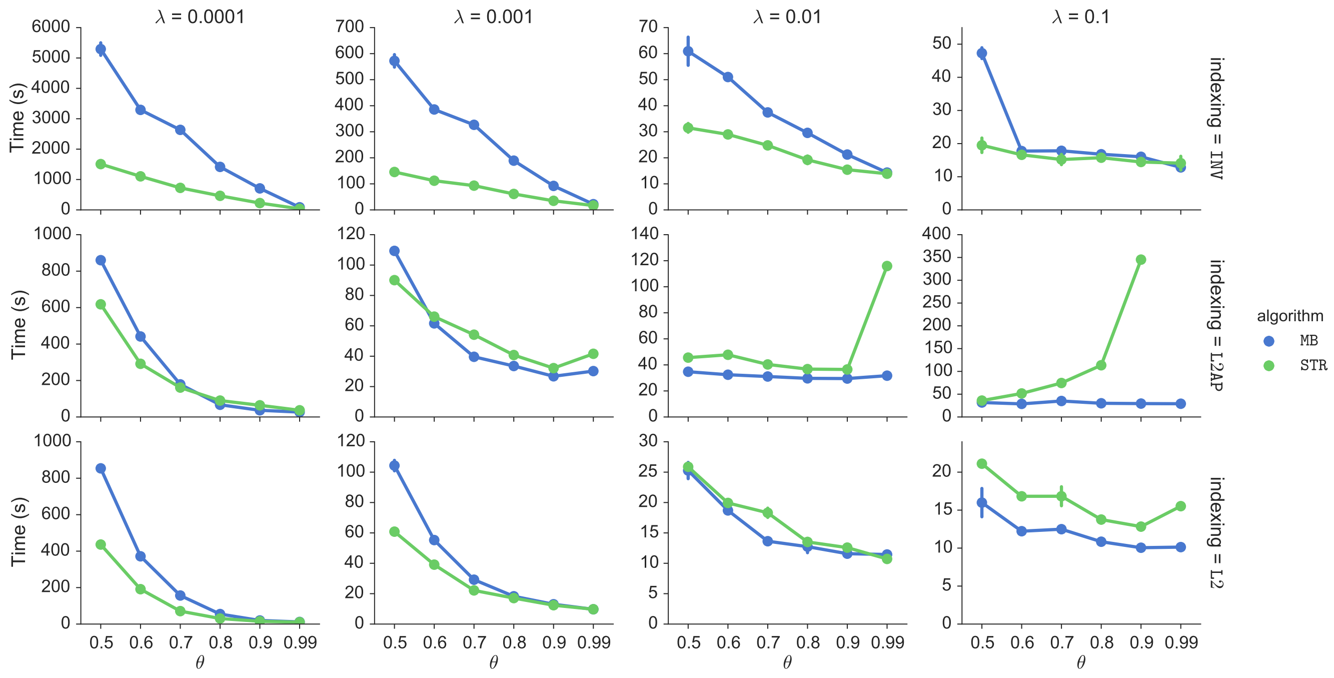

Figure 3: Time taken by the MB and STR algorithms as a function of the similarity threshold θ, on the RCV1 dataset.

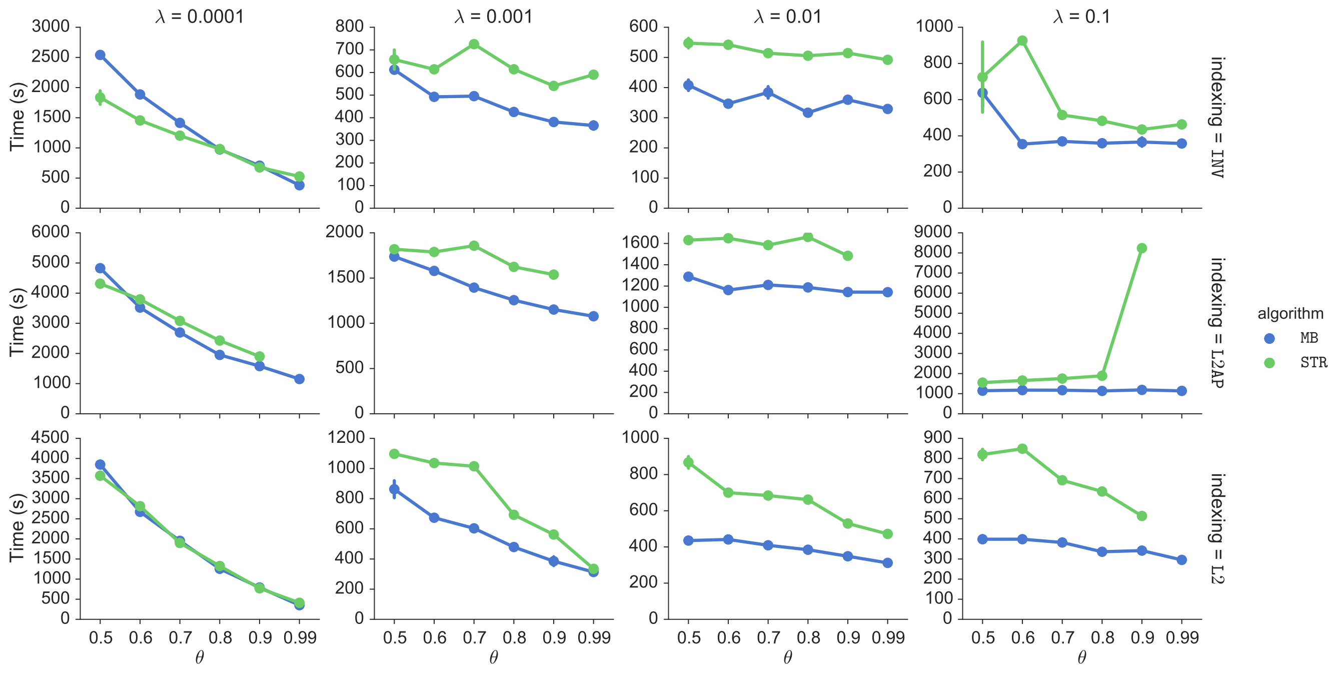

Figure 4: Time taken by the MB and STR algorithms as a function of the similarity threshold θ, on the WebSpam dataset.

In general, a shorter horizon makes other pruning strategies less relevant compared to time filtering, as can be seen from the progressive flattening of the curves from left to right as λ increases.

Figure 6 shows the comparison in the number of index entries traversed. Clearly, INV has usually the largest amount due to the absence of pruning. The relative effectiveness of pruning for L2 is almost constant throughout the range of λ, however the number of entries traversed decreases, and so does the importance of filtering the index (compared to the time filtering). This is in line with the behavior exhibited in Figure 5 where the difference in time decreases for larger λ.

Interestingly, L2AP starts very close to L2, but as the horizon shortens the number of entries traversed increases significantly, evens surpassing INV in the rightmost plot. This result shows that L2 does not lose much in terms of pruning power, despite not using the bounds from AP. Furthermore, the optimization to the implementation of time filtering de-

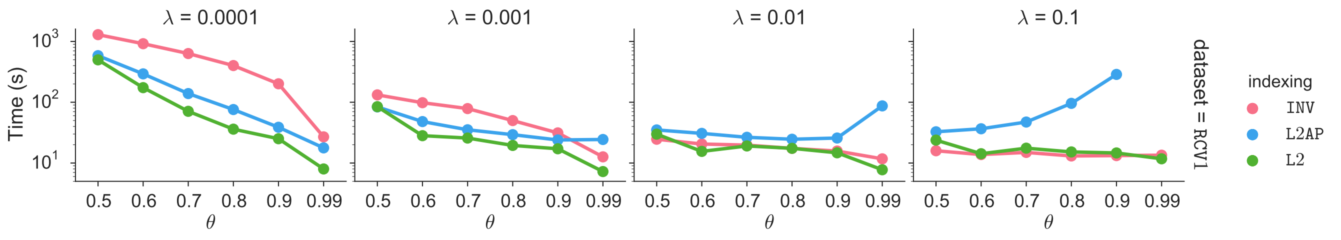

Figure 5: Time taken by STR using different indexes as a function of the similarity threshold θ, for the RCV1 dataset.

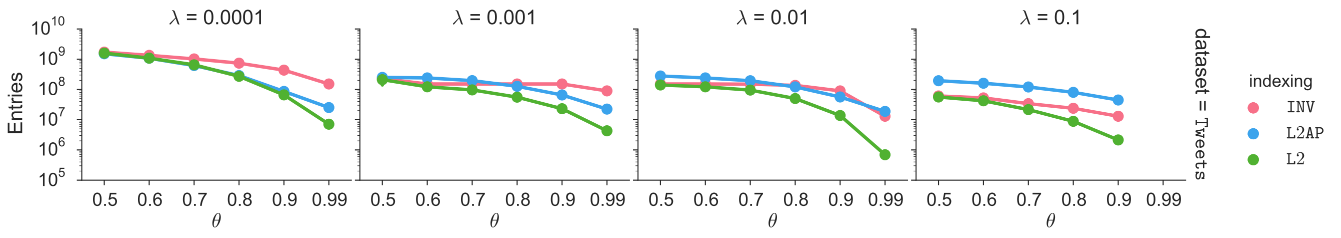

Figure 6: Entries traversed by STR using different indexes as a function of the similarity threshold θ, for the Tweets dataset.

scribed in Section 6 (backwards posting-list scanning) is not applicable. The reason is that the AP bounds required during indexing are data-dependent, which leads to re-indexing and therefore there is no guarantee that the posting lists are sorted by time. Therefore, L2AP ends up traversing more entries than L2.

Note that the last points on the rightmost plot of Figure 6 ($\theta = 0.99$ and $\lambda = 0.1$) are not shown because their value is zero, which cannot be represented on a logarithmic axis. Indeed, in this configuration $\tau = 0.1$ which is smaller than any timestamp delta, so the index gets continuously pruned before bing traversed, and therefore the output is empty.

Similar trends are observed for the number of candidates generated and the number of full similarities computed. Those results are omitted due to space constraints.

Q3 (Parameters θ and λ): Next we study the effect of the parameters of our methods: the decay factor λ and the similarity threshold θ. Hereinafter, we present results only for L2, for which we have established that it is an effective and efficient pruning scheme.

Figure 7 shows the effect of the decay factor λ. Increasing the decay factor decreases the computation time for all datasets. However, the decrease is more marked for lower threshold θ, and flattens out quickly for higher values of λ.

Figure 8 shows the effect of the similarity threshold θ. The pattern is similar to the previous figure, with the roles of θ and λ reversed. Increasing the threshold decreases the computation time for all datasets, but is more marked for lower λ, and flattens out quickly.

By combining the insights from the previous two figures, we can infer that both parameters jointly affect the computation time. The reason is that time is mainly affected by time filtering, which is the most effective pruning strategy in this setting, and its effect depends on the horizon τ, which jointly depends on the other two main parameters.

To explore this hypothesis, we perform a linear regression of the computation time on the value of the horizon τ. Figure 9 shows that the computation time is roughly a linear function of τ. In addition, from the inclination of the regression line it is clear that WebSpam is an outlier when compared to the other datasets, as previously mentioned.

8. CONCLUSION

We introduced the problem of computing the similarity self-join in data streams. Our approach relies on incorporating a forgetting factor in the similarity measure so as to be able to prune data items when they become old enough. Given the new definition of time-dependent similarity, we developed two algorithmic frameworks that incorporate existing indexing techniques for computing similarity self-join on static data, and we extended those techniques to the streaming case. We explored several different combinations of bounds used for index pruning, which, in the context of streaming data, lead to interesting performance trade-offs. Our extensive analysis allows to understand better these trade-offs, and consequently, to design an index that is optimized for streaming data. Our analysis indicates that the STR algorithm equipped with the L2 index is the most scalable and robust across all datasets and configurations.

One promising direction for future work is to experiment with dimension-ordering strategies and evaluate the cost-benefit trade-off of maintaining a dimension ordering. Other directions include applying the developed techniques in real-world applications for filtering near-duplicate items in data streams, as well as extending our model for different definitions of time-dependent similarity.

9. REFERENCES

[1] F. N. Afrati, A. Das Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy Joins Using MapReduce. Technical report, Stanford InfoLab, jul 2011.

[2] M. A. Alabduljalil, X. Tang, and T. Yang. Optimizing Parallel Algorithms for All Pairs Similarity Search. In *WSDM*, pp. 203–212. ACM, 2013.

[3] D. C. Anastasiu and G. Karypis. L2AP: Fast cosine similarity search with prefix L_2 norm bounds. In *ICDE*, pp. 784–795, 2014.

[4] A. Arasu, V. Ganti, and R. Kaushik. Efficient Exact Set-Similarity Joins. In *VLDB*, pp. 918–929. VLDB Endowment, 2006.

[5] A. Awekar and N. F. Samatova. Fast Matching for All Pairs Similarity Search. In *WI-IAT*, pp. 295–300. IEEE, 2009.

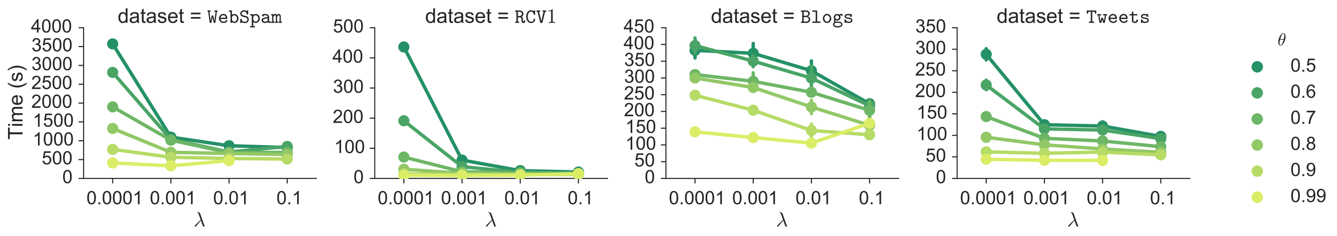

Figure 7: Time taken by STR using the L2 index as a function of the decay factor λ, for different similarity thresholds θ.

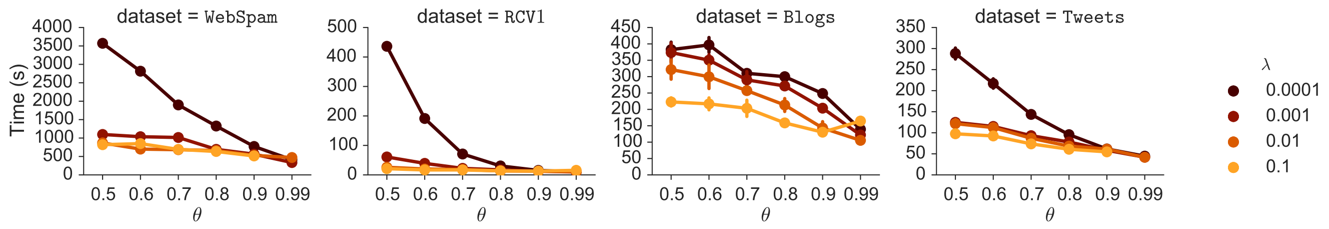

Figure 8: Time taken by STR using the L2 index as a function of the similarity thresholds θ, for different decay factors λ.

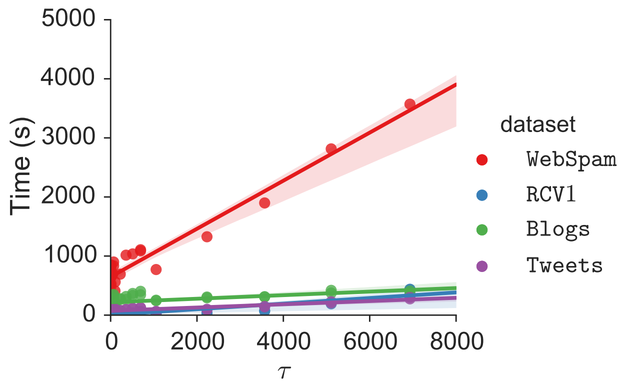

Figure 9: Time taken by the STR algorithm when using the L2 index as a function of the time horizon τ.

[6] R. Baraglia, G. De Francisci Morales, and C. Lucchese. Document Similarity Self-Join with MapReduce. In *ICDM '10*, pp. 731–736. IEEE, 2010.

[7] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. In *WWW*, pp. 131–140, 2007.

[8] C. Böhm, B. Braunmüller, M. Breunig, and H.-P. Kriegel. High Performance Clustering Based on the Similarity Join. In *CIKM*, pp. 298–305. ACM, 2000.

[9] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[10] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, p. 5, 2006.

[11] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding Replicated Web Collections. *SIGMOD Record*, 29(2): 355–366, 2000.

[12] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google News Personalization: Scalable Online Collaborative Filtering. In *WWW*, pp. 271–280, 2007.

[13] G. De Francisci Morales, C. Lucchese, and R. Baraglia. Scaling Out All Pairs Similarity Search with MapReduce. In *LSDS-IR*, pp. 25–30, 2010.

[14] L. Gao and X. S. Wang. Continually Evaluating Similarity-Based Pattern Queries on a Streaming Time Series. In *SIGMOD*, pp. 370–381. ACM, 2002.

[15] T. Haveliwala, A. Gionis, and P. Indyk. Scalable Techniques for Clustering the Web. In *WebDBs*, pp. 129–134, 2000.

[16] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *JASIST*, 54(3):203–215, 2003.

[17] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *KAIS*, 7(3):358–386, 2005.

[18] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *JMLR*, 5:361–397, 2004.

[19] X. Lian and L. Chen. Efficient Similarity Join over Multiple Stream Time Series. *TKDE*, 21(11):1544–1558, 2009.

[20] X. Lian and L. Chen. Similarity Join Processing on Uncertain Data Streams. *TKDE*, 23(11):1718–1734, 2011.

[21] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *WWW*, pp. 377–386, 2006.

[22] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating Similarity Measures: A Large-Scale Study in the Orkut Social Network. In *KDD*, pp. 678–684, 2005.

[23] E. Valari and A. N. Papadopoulos. Continuous Similarity Computation over Streaming Graphs. In *ECML-PKDD*, pp. 638–653. Springer, 2013.

[24] R. Vernica, M. J. Carey, and C. Li. Efficient Parallel Set-Similarity Joins Using MapReduce. In *SIGMOD*, pp. 495–506, New York, New York, USA, 2010. ACM.

[25] D. Wang, D. Irani, and C. Pu. Evolutionary study of web spam: Webb spam corpus 2011 versus webb spam corpus 2006. In *CollaborateCom*, pp. 40–49, 2012.

[26] S. Wang and E. Rundensteiner. Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-Slicing. In *EDBT*, pp. 299–310. ACM, 2009.

[27] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient Similarity Joins for Near Duplicate Detection. In *WWW*, pp. 131–140. ACM, 2008.

[28] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15, 2011.

[29] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM*, pp. 177–186, 2011.