SSTR: Set Similarity Join over Stream Data

Lucas Pacífico and Leonardo Andrade Ribeiro
Instituto de Informática, Universidade Federal de Goiás, Goiânia – Goiás Brazil

Keywords: Advanced Query Processing, Data Streams, Databases, Similarity Join

Abstract: In modern application scenarios, large volumes of data are continuously generated over time at high speeds. Delivering timely analysis results from such massive stream of data imposes challenging requirements for current systems. Even worse, similarity matching can be needed owing to data inconsistencies, which is computationally much more expensive than simple equality comparisons. In this context, this paper presents SSTR, a novel similarity join algorithm for streams of sets. We adopt the concept of temporal similarity and exploit its properties to improve efficiency and reduce memory usage. We provide an extensive experimental study on several synthetic as well as real-world datasets. Our results show that the techniques we proposed significantly improve scalability and lead to substantial performance gains in most settings.

1 INTRODUCTION

In the current Big Data era, large volumes of data are continuously generated over time at high speeds. Very often, there is a need for immediate processing of such stream of data to deliver analysis results in a timely fashion. Examples of such application scenarios abound, including social networks, Internet of Things, sensor networks, and a wide variety of log processing systems. Over the years, several stream processing systems have emerged seeking to meet this demand (Abadi et al., 2005; Carbone et al., 2015).

However, the requirements for stream processing systems are often conflicting. Many applications demand comparisons between historical and live data, together with the requirements for instantaneously processing and fast response times (see Rules 5 and 8 in (Stonebraker et al., 2005)). To deliver results in real-time, it is imperative to avoid extreme latencies caused by disk accesses. However, maintaining all data in the main memory is impractical for unbounded data streams.

The problem becomes even more challenging in the presence of stream imperfections, which has to be handled without causing delays in operations (Rule 3 in (Stonebraker et al., 2005)). In the case of streams coming from different sources, such imperfections may include the so-called *fuzzy duplicates*, i.e., multiple and non-identical representations of the same information. The identification of this type of redundancy requires similarity comparisons, which are computationally much more expensive than simple equality comparisons.

Table 1: Messages from distinct sources about a football match.

Source	Time	Message
X	270	Great chance missed within the penalty area.
Y	275	Shooting chance missed within the penalty area.
Z	420	Great chance missed within the penalty area.

Further, data stream has an intrinsic temporal nature. A timestamp is typically associated with each data object recording, for example, the time of its arrival. This temporal attribute represents important semantic information and, thus, can affect a given notion of similarity. Therefore, it is intuitive to consider that the similarity between two data objects decreases with their temporal distance.

As a concrete example, consider a web site providing live scores and commentary about sporting events, which aggregates streams from different sources. Because an event can be covered by more than one source, multiple arriving messages can be actually describing a same moment. Posting such redundant messages are likely to annoy users and degrade their experience. This issue can be addressed by performing a similarity (self-)join over the incoming streams — a similarity join returns all data objects whose similarity is not less than a specified threshold.

Pacífico, L. and Ribeiro, L.
SSTR: Set Similarity Join over Stream Data.
DOI: 10.5220/0009420400520060
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 1*, pages 52-60
ISBN: 978-989-758-423-7

Thus, a new message is only posted if there are no previous ones that are similar to it. In this context, temporal information is crucial for similarity assessment because two textually similar messages might be considered as distinct if the difference in their arrival time is large. For example, Table 1 shows three messages about a soccer match from different sources. All messages are very similar to one another. However, considering the time of arrival of each message, one can conclude that, while the first two messages refer to the same moment of the match, the third message, despite being identical to the first one, actually is related to a different moment.

Morales and Gionis introduced the concept of temporal similarity for streams (Morales and Gionis, 2016). Besides expressing the notion of time-dependent similarity, this concept is directly used to design efficient similarity join algorithms for streams of vectors. The best-performing algorithm exploits temporal similarity to reduce the number of comparisons. Moreover, such time-dependent similarity allows to establish an "aging factor": after some time, a given data object cannot be similar to any new data arriving in the stream and, thus, can be safely discarded to reduce memory consumption.

This paper presents an algorithm for similarity joins over set streams. There is a vast literature on set similarity joins for static data (Sarawagi and Kirpal, 2004; Chaudhuri et al., 2006; Xiao et al., 2011; Vernica et al., 2010; Ribeiro and Härder, 2011; Quirino et al., 2017; Ribeiro-Júnior et al., 2017; Mann et al., 2016; Wang et al., 2017); however, to the best of our knowledge, there is no prior work on this type of similarity join for stream data. Here, we adapt the notion of temporal similarity to sets and exploit its properties to reduce both comparison and memory spaces. We provide an extensive experimental study on several synthetic as well as real-world datasets. Our results show that the techniques we proposed significantly improve scalability and lead to substantial performance gains in most settings.

The remainder of the paper is organized as follows. Section 2 provides background material. Section 3 presents our proposed algorithm and techniques. Section 4 describes the experimental study and analyzes its results. Section 5 reviews relevant related work. Section 6 summarizes the paper and discusses future research.

2 BACKGROUND

In this section, we define the notions of set and temporal similarity, together with essential optimizations derived from these definitions. Then, we formally state the problem considered in this paper.

2.1 Set Similarity

This work focus on streams of data objects represented as sets. Intuitively, the similarity between two sets is determined by their intersection. Representing data objects as sets for similarity assessment is a widely used approach for string data (Chaudhuri et al., 2006).

Strings can be mapped to sets in several ways. For example, the string *" Great chance missed within the penalty area"* can be mapped to the set of words {*'Great', 'chance', 'missed', 'within', 'the', 'penalty', 'area'*}. Another well-known method is based on the concept of *q*-grams, i.e., substrings of length *q* obtained by "sliding" a window over the characters of the input string. For example, the string *"similar"* can be mapped to the set of *3-grams* {*'sim', 'imi', 'mil', 'ila', 'lar'*}. Henceforth, we generically refer to a set element as a *token*.

A *similarity function* returns a value in the interval $[0,1]$ quantifying the underlying notion of similarity between two sets; greater values indicate higher similarity. In this paper, we focus on the well-known Jaccard similarity; nevertheless, all techniques described here apply to other similarity functions such as *Dice* e *Cosine* (Xiao et al., 2011).

Definition 1 (Jaccard Similarity)**.** *Given two sets x and y, the Jaccard similarity between them is defined as* $J(x,y) = \frac{|x \cap y|}{|x \cup y|}$.

Example 1. *Consider the sets x and y below, derived from the two first messages in Table 1 (sources X and Y):*

$x =${*'Great', 'chance', 'missed' 'within', 'the', 'penalty', 'area'*},
$y =${*'Shooting', 'chance', 'missed', 'within', 'the', 'penalty', 'area'*}.

Then, we have $J(x,y) = \frac{6}{7+7-6} = 0.75$.

A fundamental property of the Jaccard similarity is that any predicate of the form $J(x,y) \geq \gamma$, where γ is a threshold, can be equivalently rewritten in terms of an *overlap bound*.

Lemma 1 (Overlap Bound (Chaudhuri et al., 2006))**.** *Given two sets, r and s, and a similarity threshold* γ*, let* $O(x,y,\gamma)$ *denote the corresponding overlap bound, for which the following holds:*

$J(x,y) \geq \gamma \Longleftrightarrow |x \cap y| \geq O(x,y,\gamma) = \frac{\gamma}{1+\gamma} \times (|x| + |y|)$.

Overlap bound provides the basis for several filtering techniques. Arguably, the most popular and effective techniques are *size-based filter* (Sarawagi and

Kirpal, 2004), *prefix filter* (Chaudhuri et al., 2006), and *positional filter* (Xiao et al., 2011), which we review in the following.

2.2 Optimization Techniques

Intuitively, the difference in size between two similar sets cannot be too large. Thus, one can quickly discard set pairs whose sizes differ enough.

Lemma 2 (Size-based Filter (Sarawagi and Kirpal, 2004))**.** *For any two sets x and y, and a similarity threshold γ, the following holds:*

$$J(x,y) \geq \gamma \implies \gamma \leq \frac{|x|}{|y|} \leq \frac{1}{\gamma}.$$

Prefix filter allows discarding candidate set pairs by only inspecting a fraction of them. To this end, we first fix a total order on the universe $\mathcal{U}$ from which all tokens are drawn.

Lemma 3 (Prefix Filter (Chaudhuri et al., 2006))**.** *Given a set r and a similarity threshold γ, let $pref(x,\gamma) \subseteq x$ denote the subset of x containing its first $|x| - \lceil |x| \times \gamma \rceil + 1$ tokens. For any two sets x e y, and a similarity threshold γ, the following holds:*

$$J(x,y) \geq \gamma \implies pref(x,\gamma) \cap pref(y,\gamma) \neq \varnothing.$$

The positional filter also exploits token ordering for pruning. This technique filters dissimilar set pairs using the position of matching tokens.

Lemma 4 (Positional filter (Xiao et al., 2011))**.** *Given a set x, let $w = x[i]$ be a token of x at position i, which divides x into two partitions, $x_l(w) = x[1,..,(i-1)]$ and $x_r(w) = x[i,..,|x|]$. Thus, for any two sets x e y, and a similarity threshold γ, the following holds:*

$$J(x,y) \geq \gamma \implies |x_l \cap y_l| + min(|x_r|,|y_r|) \geq O(x,y,\gamma).$$

2.3 Temporal Similarity

Each set x is associated with a timestamp, denoted by $t(x)$, which indicates, for example, its arrival time. Formally, the input stream is denoted by $\mathcal{S} = \langle \ldots, (x_i, t(x_i)), (x_i, t(x_{i+1}), \ldots \rangle$.

The concept of temporal similarity captures the intuition that the similarity between two sets diminishes with their temporal distance. To this end, the difference in the arrival time is incorporated into the similarity function.

Definition 2 (Temporal Similarity (Morales and Gionis, 2016))**.** *Given two sets x e y, let $\Delta t_{xy} = |t(x) - t(y)|$ be the difference in their arrival time. The temporal similarity between x and y is defined as*

$$J_{\Delta t}(x,y) = J(x,y) \times e^{-\lambda \times \Delta t_{xy}},$$

where λ is a time-decay parameter.

Example 2. *Consider again the sets x and y from Example 1, obtained from sources X and Y, respectively, in Table 1, and $\lambda = 0.01$. We have $\Delta t_{xy} = 5$ and, thus, $J_{\Delta t}(x,y) = 0.75 \times e^{-\lambda \times 5} \approx 0.71$. Consider now set z obtained from source Z. Despite of sharing all tokens, x and z have a relatively large temporal distance, i.e., $\Delta t_{xz} = 150$. As a result, we have $J_{\Delta t}(x,z) = 1 \times e^{-\lambda \times 150} \approx 0.22$.*

Note that $J_{\Delta t}(x,y) = J(x,y)$ when $\Delta t_{xy} = 0$ or $\lambda = 0$, and its limit is 0 as Δt_{xy} approaches infinity, at an exponential rate modulated by λ. The time-decay factor, together with the similarity threshold, allows defining a *time filter*: given a set x, after a certain period, called *time horizon*, no newly arriving set can be similar to x.

Lemma 5 (Time Filter (Morales and Gionis, 2016))**.** *Given a time-decay factor λ, let $\tau = \frac{1}{\lambda} \times ln\frac{1}{\gamma}$ be the time horizon. Thus, for any two sets x e y, the following holds:*

$$J_{\Delta t}(x,y) \geq \gamma \implies \Delta t_{xy} < \tau.$$

Note that the time horizon establishes a temporal window of fixed size, which slides as a new set arrives. While in the traditional sliding window model (Babcock et al., 2002) the amount of stream data is fixed, the number of sets can vary widely across different temporal windows.

2.4 Problem Statement

We are now ready to formally define the problem considered in this paper.

Definition 3 (Similarity Join over Set Streams)**.** *Given a stream of timestamped sets $\mathcal{S}$, a similarity threshold γ, and a time-decay factor λ, a similarity join over $\mathcal{S}$ returns all set pairs (x,y) in $\mathcal{S}$ such that $J_{\Delta t}(x,y) \geq \gamma$.*

3 SIMILARITY JOIN OVER SET STREAMS

In this section, we present our proposal to solve the problem of efficiently answering similarity joins over set streams. We first describe a baseline approach based on a straightforward adaptation of an existing set similarity join algorithm for static data. Then, we present the main contribution of this paper, a new algorithm deeply integrating characteristics of temporal similarity to improve runtime and reduce memory consumption.

Algorithm 1: The PPJoin algorithm over set streams.

```
    Input: Set stream $\mathcal{S}$, threshold $\gamma$, decay $\lambda$
    Output: All pairs $(x,y) \in \mathcal{S}$ s.t. $J_{\Delta t}(x,y) \geq \gamma$
1   $I_i \leftarrow \varnothing \; (1 \leq i \leq |\mathcal{U}|)$
2   while true do
3       $x \leftarrow$ read $(\mathcal{S})$
4       $M \leftarrow$ empty map from set id to int
5       for $i \leftarrow 1$ to $|pref(x,\gamma)|$ do
6           $k \leftarrow x[i]$
7           foreach $(y,j) \in I_k$ do
8               if $|y| < |x| \times \gamma$ then
9                   continue
10              $ubound \leftarrow 1 + min(|x|-i, |y|-j)$
11              if $M[y] + ubound \geq O(x,y,\gamma)$ then
12                  $M[y] \leftarrow M[y] + 1$
13              else
14                  $M[y] \leftarrow -\infty$
15          $I_k \leftarrow I_k \cup (x,i)$
16      $R' \leftarrow$ Verify $(x, M, \gamma)$
17      $R \leftarrow$ ApplyDecay $(R', \gamma, \lambda)$
18      Emit $(R)$
```

3.1 Baseline Approach

Most state-of-the-art set similarity join algorithms follow a filtering-verification framework (Mann et al., 2016). In this framework, the input set collection is scanned sequentially, and each set goes through the filtering and verification phases. In the filtering phase, tokens of the current set (called henceforth probe set) are used to find potentially similar sets that have already been processed (called henceforth candidate sets). The filters discussed in the previous section are then applied to reduce the number of candidates. This phase is supported by an inverted index, which is incrementally built as the sets are processed. In the verification phase, the similarity between the probe set and each of the surviving candidates is fully calculated, and those pairs satisfying the similarity predicate are sent to the output.

A naive way to perform set similarity join in a stream setting is to simply carry out the filtering and verification phases of an existing algorithm on each incoming set. The temporal decay is then applied to the similarity of the pairs returned by the verification in a post-processing phase, before sending results to the output.

Algorithm 1 describes this naive approach for PPJoin (Xiao et al., 2011), one of the best performing algorithms in a recent empirical evaluation (Mann et al., 2016). The algorithm continuously processes sets from the input stream as they arrive. The filtering phase uses prefix tokens (Line 5) to probe the inverted index (Line 7). Each set found in the associated inverted list is considered a candidate and checked against conditions using the size-based filter (Line 8) and the positional filter (Lines 10–11). A reference to the probe set is appended to the inverted list associated with each prefix token (Line 15). Not shown in the algorithm, the verification phase (Line 16) can be highly optimized by exploiting the token ordering in a merge-like fashion and the overlap bound to define early stopping conditions (Ribeiro and Härder, 2011). Finally, the temporal decay is applied, and a last check against the threshold is performed to produce an output (Line 17).

Clearly, the above approach has two serious drawbacks. First, space consumption of the inverted index can be exorbitant and quickly exceed the available memory. Even worse, a large part of the index can be stale entries, i.e., entries referencing sets that will not be similar to any set arriving in the future. Second, temporal decay is applied only after the verification phase. Therefore, much computation in the verification is wasted on set pairs that cannot be similar owing to the difference in their arrival times.

3.2 The SSTR Algorithm

We now present our proposed algorithm called SSTR for similarity joins over set streams. SSTR exploits properties of the temporal similarity definition to avoid the pitfalls of the naive approach. First, SSTR dynamically removes old entries from the inverted lists that are outside the window induced by the probe set and the time horizon. Second, it uses the temporal decay to derive a new similarity threshold between the probe set and each candidate set. This new threshold is greater than the original, which increases the effectiveness of the size-based and positional filters.

The steps of STTR are formalized in Algorithm 2. References to sets whose difference in arrival time with the probe set is greater than the time horizon is removed as the inverted lists are scanned (Line 8). Note that the entries in the inverted lists are sorted in increasing timestamp order. Thus, all stale entries are grouped at the beginning of the lists. For each candidate set, a new threshold value is calculated (Line 10), which is used in the size-based filter and to calculate the overlap bound (Lines 11 and 15, respectively). In the same way, such increased, candidate-specific threshold is also used in the verification phase to obtain greater overlap bounds and, thus, improve the effectiveness of the early-stop conditions. For this

Algorithm 2: The SSTR algorithm.

```
   Input: Set stream $\mathcal{S}$, threshold $\gamma$, decay $\lambda$
   Output: All pairs $(x,y) \in \mathcal{F}$ s.t. $J_{\Delta t}(x,y) \geq \gamma$
1  $\tau = \frac{1}{\lambda} \times ln\frac{1}{\gamma}$
2  $I_i \leftarrow \varnothing \ (1 \leq i \leq |\mathcal{U}|)$
3  while true do
4      $x \leftarrow$ read $(\mathcal{S})$
5      $M \leftarrow$ empty map from set id to int
6      for $i \leftarrow 1$ to $|pref(x,\gamma)|$ do
7          $k \leftarrow x[i]$
8          Remove all $(y,j)$ from $I_k$ s.t. $\Delta t_{xy} > \tau$
9          foreach $(y,j) \in I_k$ do
10             $\gamma' \leftarrow \frac{\gamma}{e^{-\lambda \times \Delta t_{xy}}}$
11             if $|y| < |x| \times \gamma'$ then
12                 $M[y] \leftarrow -\infty$
13                 continue
14             $ubound \leftarrow 1 + min(|x| - i, |y| - j)$
15             if $M[y].s + ubound \geq O(x,y,\gamma')$
                then
16                 $M[y].s \leftarrow M[y].s + 1$
17             else
18                 $M[y] \leftarrow -\infty$
19         $I_k \leftarrow I_k \cup (x,i)$
20     $R \leftarrow$ Verify $(x,M,\gamma,\lambda)$
21     Emit $(R)$
```

reason, the time-decay parameter is passed to the *Verify* procedure (Line 20), which now directly produces output pairs[1].

Even with the removal of stale entries from the inverted lists, SSTR still can incurs into high memory consumption issues for temporal windows containing too many sets. This situation can happen due to very small time-decay parameters leading to large windows or at peak data stream rate leading to "dense" windows. In such cases, sacrificing timeliness by resorting to some approximation method, such as batch processing (Babcock et al., 2002), is inevitable. Nevertheless, considering a practical scenario where a memory budget has been defined, the SSTR algorithm can dramatically reduce the frequency of such batch processing modes in comparison to the baseline approach, as we empirically demonstrate next.

[1]In our implementation, we avoid repeated calculations of candidate-specific thresholds and overlap bounds by storing them in the map M.

Table 2: Datasets statistics.

Name	Population	Avg. set size	Timestamp
DBLP	350 000	76	Poisson
WIKI	1 000 000	53	Uniform
TWITTER	2 824 998	90	Publishing Date
REDDIT	19 456 493	53	Publishing Date

4 EXPERIMENTS

We now present an experimental study of the techniques proposed in this paper. The goal of our experiments is to evaluate the effectiveness of our proposed techniques for reducing the comparison space and memory consumption. To this end, we compare our SSTR algorithm with the baseline approach, which is abbreviated to SPPJ (streaming PPJoin). In this context, we also evaluated the effect of the parameters γ and λ in the resulting execution times.

4.1 Datasets and Setup

We used four datasets: DBLP[2], containing information about computer science publications; WIKI[3], an encyclopedia containing generalized information about different topics; TWITTER[4], geocoded tweets collected during Brazil elections from 2018; and REDDIT, a social news aggregation, web content rating, and discussion website (Baumgartner, 2019). For DBLP and WIKI, we started by randomly selecting 70k and 200k article titles, respectively. Then, we generated four fuzzy duplicates from each string by performing transformations on string attributes, such as characters insertions, deletions or substitutions. We end up with 350k and 1M strings for DBLP and WIKI, respectively. Finally, we assigned artificial timestamps to each string in these datasets, sampled from a Poisson (DBLP) and Uniform (WIKI) distribution function. For this reason, we call DBLP and WIKI (semi)synthetic datasets. For TWITTER and REDDIT, we used the complete dataset available without applying any modification, where the publication time available for each item was used as timestamp. For this reason, we call TWITTER and REDDIT real-world datasets. The datasets are heterogeneous, exhibiting different characteristics, as summarized in Table 2.

For the similarity threshold γ, we explore a range of values in $[0.5, 0.95]$, while the time-decay fac-

[2]dblp.uni-trier.de/xml

[3]https://en.wikipedia.org/wiki/Wikipedia:Database_download

[4]https://developer.twitter.com/en/products/tweets

tor λ we use exponentially increasing values in the range $[10^{-4}, 10^{-1}]$. For all datasets, we tokenized the strings into sets of *3*-grams, hashed the tokens into four byte values, and ordered them within each set lexicographically.

We conduct our experiments on an Intel E5-2620 @ 2.10GHz with 15MB of cache, 16GB of RAM, running Ubuntu 16.04 LTS. We report the average runtime over five runs. All algorithms were implemented in Java SDK 11.

Some parameter configurations were very troublesome to execute in our hardware environment, both in terms of runtime and memory; this is particularly the case for SPPJ on the largest datasets. As a result, we were unable to finish the execution of the algorithms in some settings. In this study, the experiments have a timeout of 3 hours for each execution.

4.2 Results

We first analyze the results on the synthetic datasets. Figure 1 plots the runtimes for SSTR and SPPJ on DBLP and WIKI datasets. As expected, SPPJ was only able to finish its execution for very high threshold values. In contrast, SSTR successfully terminated in all settings on WIKI. Higher threshold values increase the effectiveness of the prefix filter, which benefits both SPPJ and SSTR. Yet, in most cases where SPPJ was able to terminate, SSTR was up to three orders of magnitude faster. These results highlight the effectiveness of our techniques in drastically reducing the number of similarity comparisons as well as memory usage.

On the DBLP dataset, SSTR terminates within the time limit for all threshold values only for $\lambda = 0.1$. The reason is that the Poisson distribution generates some very dense temporal windows, with set objects temporally very close to each other. For small time-decay values, temporal windows are large and more sets have to be kept in the inverted lists and compared in the verification phase. Conversely, greater time-decay values translate into a smaller time horizon and, thus, narrower temporal windows. As a result, the time filter is more effective for pruning stale entries from the inverted lists. Moreover, time-decay values lead to greater candidate-specific thresholds, which, in turn, improve the pruning power of the size-based and positional filters.

We now analyze the results on the real-world datasets. Figure 2 plots the runtimes for SSTR on TWITTER and REDDIT. We do not show the results for SPPJ because it failed due to lack of memory on these datasets in all settings. Obviously, as SPPJ does not prune stale entries from the index, it cannot directly handle the largest datasets in our experimental setting. Note that we can always reconstruct the inverted index, for example after having reached some space limit. However, this strategy sacrifices timeliness, accuracy, or both. While resorting to such batch processing mode is inevitable in stressful scenarios, the results show that the SSTR algorithm can nevertheless sustain continuous stream processing much longer than SPPJ.

Another important observation is that, overall, SSTR successfully terminates in all settings on real-world datasets; the only exception is on REDDIT for the smallest λ value. Moreover, even though those datasets are larger than DBLP and WIKI, SSTR is up to two orders of magnitude faster on them. The explanation lies in the timestamp distribution of the real-world datasets, which exhibit more ”gaps” as compared to the synthetic ones. Hence, the induced temporal windows are more ”sparse” on those datasets. which is effectively exploited by the time filter to dynamically maintain the length of the inverted lists reduced to a minimum. The other trends remain the same: execution times increase and decrease as similarity thresholds and time-decay parameters decrease and increase, respectively.

5 RELATED WORK

There is a long line of research on efficiently answering set similarity joins (Sarawagi and Kirpal, 2004; Chaudhuri et al., 2006; Xiao et al., 2011; Vernica et al., 2010; Ribeiro and Härder, 2011; Quirino et al., 2017; Ribeiro-Júnior et al., 2017; Mann et al., 2016; Wang et al., 2017). Popular optimizations, such as size-based filtering, prefix filtering, and positional filtering, were incorporated into our algorithm. Recently, reference (Wang et al., 2017) exploited set relations to improve performance — the key insight is that similar sets produce similar results. However, one of the underlying techniques, the so-called index-level skipping, relies on building the whole inverted index before start processing and, thus, cannot be used in our context where new sets are continuously arriving.

Further, set similarity join has been addressed in a wide variety of settings, including: distributed platforms (Vernica et al., 2010; do Carmo Oliveira et al., 2018); many-core architectures (Quirino et al., 2017; Ribeiro-Júnior et al., 2017); relational DBMS, either declaratively in SQL (Ribeiro et al., 2016b) or within the query engine as a physical operator (Chaudhuri et al., 2006); cloud environments (Sidney et al., 2015); integrated into clustering algorithms

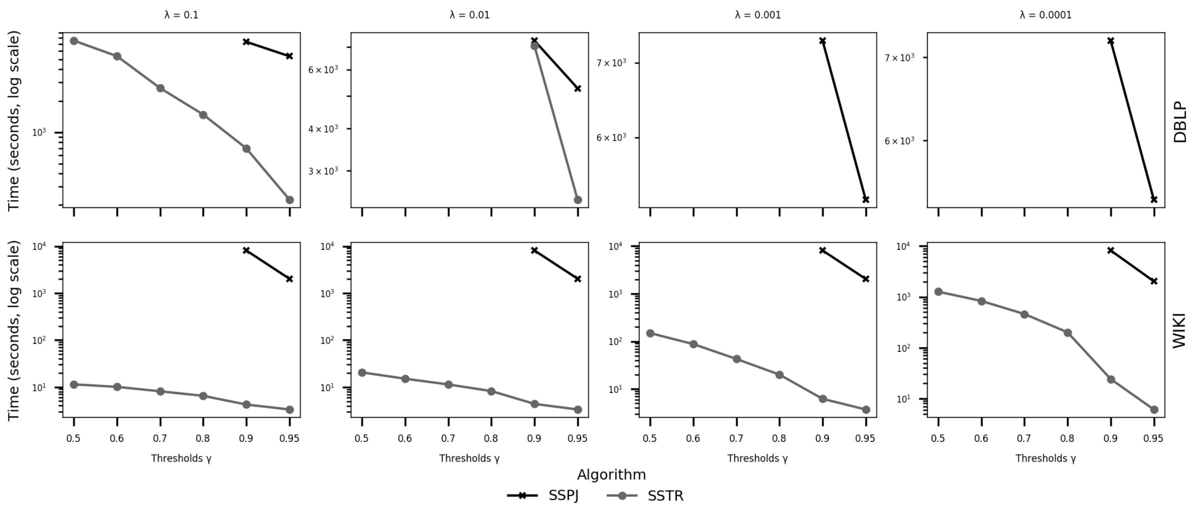

Figure 1: Runtime results of the algorithms SPPJ and SSTR on synthetic datasets.

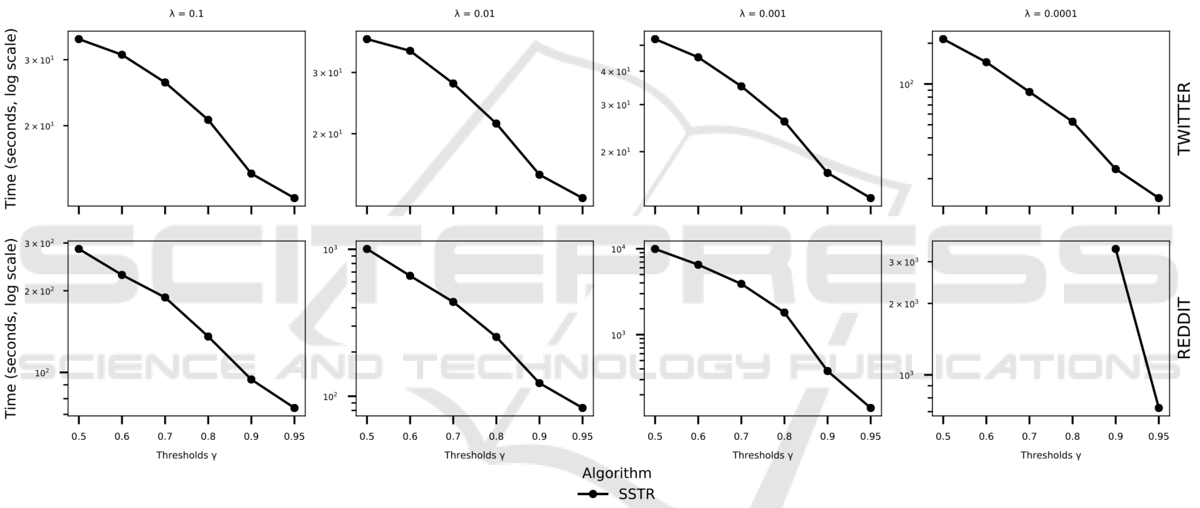

Figure 2: Runtime results of the algorithms SSTR on real-world datasets.

(Ribeiro et al., 2016a; Ribeiro et al., 2018); and probabilistic, either for increasing performance (at the expense of missing some valid results) (Broder et al., 1998; Christiani et al., 2018) or modeling uncertain data (Lian and Chen, 2010). However, none of these previous studies considered similarity join over set streams.

Previous work on similarity join over streams focused on data objects represented as vectors, where the similarity between two using vectors is measured using Euclidean distance (Lian and Chen, 2009; Lian and Chen, 2011) or cosine (Morales and Gionis, 2016). Lian and Chen (Lian and Chen, 2009) proposed an adaptive approach based on a formal cost model for multi-way similarity join over streams. The same authors later addressed similarity joins over uncertain streams (Lian and Chen, 2011).

Morales and Gionis (Morales and Gionis, 2016) introduced the notion of time-dependent similarity. The authors then adapted existing similarity join algorithms for vectors, namely AllPairs (Bayardo et al., 2007) and L2AP (Anastasiu and Karypis, 2014), to incorporate this notion and exploit its properties to reduce the number of candidate pairs and dynamically remove stale entries from the inverted index. We follow a similar approach here, but the details of these optimizations are not directly applicable to our context, as we focus on a stream of data objects represented as sets.

Processing the entire data of possibly unbounded streams is clearly infeasible. Therefore, some method has to be used to limit the portion of stream history processed at each query evaluation. The sliding window model is popularly used in streaming similarity processing (Lian and Chen, 2009; Lian and Chen, 2011; Shen et al., 2014). As already mentioned, only a fixed amount of recent stream data is computed at each query evaluation in this model (Babcock et al., 2002). In contrast, the temporal similarity adopted here induces a fixed temporal window (i.e., the time horizon) with a variable amount of stream data.

Streaming similarity search finds all data objects that are similar to a given query (Kraus et al., 2017). To some extent, similarity join can be viewed as a sequence of searches using each arriving object as a query object. A fundamental difference in this context is that the threshold is fixed for joins, while it can vary along distinct queries for searches.

Top-k queries have also been studied in the streaming setting (Shen et al., 2014; Amagata et al., 2019). Focusing on streams of vectors, Shen et al. (Shen et al., 2014) proposed a framework supporting queries with different similarity functions and window sizes. Amagata et al. (Amagata et al., 2019) presented an algorithm for kNN self-join, a type top-k query that finds the k most similar objects for each object. This work assumes objects represented as sets, however the dynamic scenario considered is very different: instead of a stream of sets, the focus is on a stream of updates continuously inserting and deleting elements of existing sets.

Finally, duplicate detection in streams is a well-studied problem (Metwally et al., 2005; Deng and Rafiei, 2006; Dutta et al., 2013). A common approach to dealing with unbounded streams is to employ space-preserving, probabilistic data structures, such as Bloom Filters and Quotient Filters together with window models. However, these proposals aim at detecting exact duplicates and, therefore, similarity matching is not addressed.

6 CONCLUSIONS AND FUTURE WORK

This paper presented a new algorithm called SSTR for set similarity join over set streams. To the best of our knowledge, set similarity join has not been previously investigated in a streaming setting. We adopted the concept of temporal similarity and exploited its properties to reduce processing cost and memory usage. We reported an extensive experimental study on synthetic and real-world datasets, whose results confirmed the efficiency of our solution. Future work is mainly oriented towards designing a parallel version of SSTR and an algorithmic framework for seamless integration with batch processing models.

ACKNOWLEDGEMENTS

This work was partially supported by Brazilian agency CAPES.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. B. (2005). The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research*, pages 277–289.

Amagata, D., Hara, T., and Xiao, C. (2019). Dynamic Set kNN Self-Join. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 818–829.

Anastasiu, D. C. and Karypis, G. (2014). L2AP: fast cosine similarity search with prefix L-2 norm bounds. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 784–795.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and Issues in Data Stream Systems. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–16.

Baumgartner, J. (2019). Reddit May 2019 Submissions. Harvard Dataverse.

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up All Pairs Similarity Search. In *Proceedings of the International World Wide Web Conferences*, pages 131–140. ACM.

Broder, A. Z., Charikar, M., Frieze, A. M., and Mitzenmacher, M. (1998). Min-Wise Independent Permutations (Extended Abstract). In *Proceedings of the ACM SIGACT Symposium on Theory of Computing*, pages 327–336. ACM.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink^{TM}: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38(4):28–38.

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the IEEE International Conference on Data Engineering*, page 5. IEEE Computer Society.

Christiani, T., Pagh, R., and Sivertsen, J. (2018). Scalable and Robust Set Similarity Join. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1240–1243. IEEE Computer Society.

Deng, F. and Rafiei, D. (2006). Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 25–36.

do Carmo Oliveira, D. J., Borges, F. F., Ribeiro, L. A., and Cuzzocrea, A. (2018). Set similarity joins with complex expressions on distributed platforms. In *Proceedings of the Symposium on Advances in Databases and Information Systems*, pages 216–230.

Dutta, S., Narang, A., and Bera, S. K. (2013). Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams. *Proceedings of the VLDB Endowment*, 6(8):589–600.

Kraus, N., Carmel, D., and Keidar, I. (2017). Fishing in the Stream: Similarity Search over Endless Data. In *bigdata*, pages 964–969.

Lian, X. and Chen, L. (2009). Efficient Similarity Join over Multiple Stream Time Series. *IEEE Transactions on Knowledge and Data Engineering*, 21(11):1544–1558.

Lian, X. and Chen, L. (2010). Set Similarity Join on Probabilistic Data. *Proceedings of the VLDB Endowment*, 3(1):650–659.

Lian, X. and Chen, L. (2011). Similarity Join Processing on Uncertain Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 23(11):1718–1734.

Mann, W., Augsten, N., and Bouros, P. (2016). An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB*, 9(9):636–647.

Metwally, A., Agrawal, D., and El Abbadi, A. (2005). Duplicate Detection in Click Streams. In *Proceedings of the International World Wide Web Conferences*, pages 12–21.

Morales, G. D. F. and Gionis, A. (2016). Streaming Similarity Self-Join. *Proceedings of the VLDB Endowment*, 9(10):792–803.

Quirino, R. D., Ribeiro-Júnior, S., Ribeiro, L. A., and Martins, W. S. (2017). fgssjoin: A GPU-based Algorithm for Set Similarity Joins. In *International Conference on Enterprise Information Systems*, pages 152–161. SCITEPRESS.

Ribeiro, L. A., Cuzzocrea, A., Bezerra, K. A. A., and do Nascimento, B. H. B. (2016a). Sjclust: Towards a framework for integrating similarity join algorithms and clustering. In *International Conference on Enterprise Information Systems*, pages 75–80. SCITEPRESS.

Ribeiro, L. A., Cuzzocrea, A., Bezerra, K. A. A., and do Nascimento, B. H. B. (2018). SjClust: A Framework for Incorporating Clustering into Set Similarity Join Algorithms. *LNCS Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 38:89–118.

Ribeiro, L. A. and Härder, T. (2011). Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems*, 36(1):62–78.

Ribeiro, L. A., Schneider, N. C., de Souza Inácio, A., Wagner, H. M., and von Wangenheim, A. (2016b). Bridging Database Applications and Declarative Similarity Matching. *Journal of Information and Data Management*, 7(3):217–232.

Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. (2017). Fast Parallel Set Similarity Joins on Many-core Architectures. *Journal of Information and Data Management*, 8(3):255–270.

Sarawagi, S. and Kirpal, A. (2004). Efficient Set Joins on Similarity Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 743–754.

Shen, Z., Cheema, M. A., Lin, X., Zhang, W., and Wang, H. (2014). A Generic Framework for Top-k Pairs and Top-k Objects Queries over Sliding Windows. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1349–1366.

Sidney, C. F., Mendes, D. S., Ribeiro, L. A., and Härder, T. (2015). Performance Prediction for Set Similarity Joins. In *Proceedings of the ACM Symposium on Applied Computing*, pages 967–972.

Stonebraker, M., Çetintemel, U., and Zdonik, S. B. (2005). The 8 Requirements of Real-time Stream Processing. *SIGMOD Record*, 34(4):42–47.

Vernica, R., Carey, M. J., and Li, C. (2010). Efficient Parallel Set-similarity Joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 495–506. ACM.

Wang, X., Qin, L., Lin, X., Zhang, Y., and Chang, L. (2017). Leveraging Set Relations in Exact Set Similarity Join. *Proceedings of the VLDB Endowment*, 10(9):925–936.

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient Similarity Joins for Near-duplicate Detection. *ACM Transactions on Database Systems*, 36(3):15:1–15:41.