CONTENTS

ABSTRACT 1

1. INTRODUCTION 1

2. BACKGROUND & PROBLEM DEFINITION 2

2.1. SET SIMILARITY AND Î• -NEIGHBORHOOD 2

2.2. INDEXING TECHNIQUES FOR SETS 2

2.3. DENSITY-BASED CLUSTERING 3

2.4. THE DBSCAN ALGORITHM 3

2.5. PROBLEM STATEMENT 3

3. BASELINE APPROACHES 4

3.1. SYM-CLUST: DBSCAN WITH INVERTED INDEX 4

3.2. JOIN-CLUST: MATERIALIZED NEIGHBORHOODS 5

4. THE SPREAD ALGORITHM 5

4.1. KEY CHALLENGES 5

4.2. DATA STRUCTURES 6

4.3. THE ALGORITHM 6

4.4. CORRECTNESS 7

4.5. COMPLEXITY ANALYSIS 8

4.6. MULTI-CORE EXTENSION 8

5. EXPERIMENTAL EVALUATION 8

5.1. INDEX & CLUSTER STATISTICS 9

5.2. RUNTIME EFFICIENCY 9

5.3. MEMORY EFFICIENCY 9

5.4. SCALABILITY 9

6. RELATED WORK 11

7. CONCLUSION 11

REFERENCES 12

Scaling Density-Based Clustering to Large Collections of Sets

Daniel Kocher
University of Salzburg
Salzburg, Austria
dkocher@cs.sbg.ac.at

Nikolaus Augsten
University of Salzburg
Salzburg, Austria
nikolaus.augsten@sbg.ac.at

Willi Mann
Celonis SE
Munich, Germany
w.mann@celonis.com

ABSTRACT

We study techniques for clustering large collections of sets into DBSCAN clusters. Sets are often used as a representation of complex objects to assess their similarity. The similarity of two objects is then computed based on the overlap of their set representations, for example, using Hamming distance. Clustering large collections of sets is challenging. A baseline that executes the standard DBSCAN algorithm suffers from poor performance due to the unfavorable neighborhood-by-neighborhood order in which the sets are retrieved. The DBSCAN order requires the use of a symmetric index, which is less effective than its asymmetric counterpart. Precomputing and materializing the neighborhoods to gain control over the retrieval order often turns out to be infeasible due to excessive memory requirements.

We propose a new, density-based clustering algorithm that processes data points in any user-defined order and does not need to materialize neighborhoods. Instead, so-called backlinks are introduced that are sufficient to achieve a correct clustering. Backlinks require only linear space while there can be a quadratic number of neighbors. To the best of our knowledge, this is the first DBSCAN-compliant algorithm that can leverage asymmetric indexes in linear space. Our empirical evaluation suggests that our algorithm combines the best of two worlds: it achieves the runtime performance of materialization-based approaches while retaining the memory efficiency of non-materializing techniques.

1 INTRODUCTION

We consider the problem of partitioning large collections of sets into DBSCAN [15] clusters. Our work is motivated by a process mining use case at Celonis SE that models processes as sets. A *process* is a sequence of timestamped activities. Large companies store hundreds of millions of activities in millions of processes. In order to analyze the processes, they should be clustered into groups of similar activity sequences that can be further explored [5, 22, 39]. To this end, a process is represented by the set of all its neighboring activity pairs, e.g., the process with the activity sequence (S, O, P, H, R, F, E) (Start, Order, Pay, sHip, Return good, reFund, End) is represented by the set $\{(S, O), (O, P), (P, H), (H, R), (R, F), (F, E)\}$. The similarity of two processes is then assessed by the Hamming distance[1] of their set representations.

Sets are used in many other applications [29] to represent objects for the purpose of clustering, e.g., sales may be represented by sets of product categories, photos by sets of tags and title words, user interactions on a website by sets of visited links, users of a social network by their group memberships, or users of a music streaming platform by sets of tracks they listen to.

[1]Hamming distance $H(r, s) = |r \cup s| - |r \cap s|$ for two sets r and s.

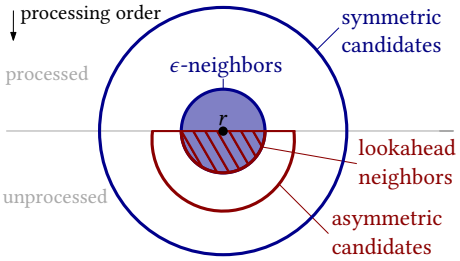

Figure 1: Symmetric candidates with ϵ-neighbors (blue); asymmetric candidates with lookahead neighbors (red).

The popular DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm [15] identifies clusters of arbitrary shape without requiring the number of clusters as input. Intuitively, DBSCAN finds dense regions that are separated by regions of lower density. The density of a region (given a distance function between pairs of data points) is defined by two parameters: the number of neighbors, *minPts*, and the radius, ϵ, of the neighborhood. A data point is called *core point* (i.e., it is at the core of a dense region) if it has at least minPts neighbors (including itself) within radius ϵ; a non-core point in the ϵ-neighborhood of a core point is a *border point* (i.e., it is at the border of a dense region); all other points are *noise* [37].

The runtime of the DBSCAN algorithm heavily depends on the efficiency of the neighborhood computation. In our experiments, the neighborhood computation accounts for up to 99% of the overall runtime for some datasets. Therefore, in order to efficiently cluster large collections of sets, effective indexing techniques for sets are required.

Similarity indexes for sets have been proposed in the context of ϵ-neighborhood joins, which are executed in an index nested loop fashion. A prominent representative is the *prefix index* [1, 8], which is linear in size and has been shown to be highly effective [29]. The *symmetric* prefix index returns the complete ϵ-neighborhood for a given query point r. The *asymmetric* prefix index assumes a processing order on the sets in R and retrieves only the *lookahead neighbors*: the ϵ-neighbors that follow r in processing order. A typical processing order for sets is based on the set sizes (ties broken arbitrarily). The *asymmetric* prefix index further leverages the length information to avoid many of the candidates that the symmetric index must inspect (among the unprocessed sets). Figure 1 illustrates the ϵ-neighborhood, the lookahead neighbors, and the candidate regions of symmetric and asymmetric index, respectively. The region above the gray line represents the sets that have been processed before r, the region below the gray line are unprocessed sets. The circles and semicircles show subset relationships.

Clearly, the asymmetric prefix index is preferable in terms of effectiveness over its symmetric counterpart. Unfortunately, there is an inherent mismatch between the asymmetric index and the DBSCAN algorithm. DBSCAN suffers from the following issues when executed with the asymmetric index: (1) *Core status*

problem: The lookahead neighbors of r are not sufficient to update the core status of r. (2) *Border vs. Noise problem:* To distinguish border points from noise, a border point must be visible from a core point, which is not guaranteed by the asymmetric lookahead neighborhood. (3) *Disconnected clusters:* To guarantee connected clusters, DBSCAN imposes a (partial) processing order on the neighborhood computations: all core points of the current cluster must be expanded (i.e., their neighborhood must be computed) before any point belonging to a different cluster is processed.

A well-known clustering approach [3] is based on a self-join that precomputes and materializes all neighborhoods. The precomputed neighborhoods are then used while executing DBSCAN. This approach can leverage the asymmetric index and is efficient in runtime. Unfortunately, this join-based technique requires quadratic memory in the worst case and suffers from a large memory footprint in practice. For example, for our social media dataset (LIVEJ) that stores the interests of 3.1M users, this approach requires almost 100GB of memory.

Summarizing, applications that must cluster large collections of sets have two options, which we call Sym-Clust and Join-Clust. (1) Sym-Clust: Retrieve the full ϵ-neighborhoods in the processing order imposed by DBSCAN using the symmetric index. (2) Join-Clust: Compute and materialize neighborhoods in a join using the effective asymmetric index. None of the options is satisfying: Sym-Clust runs almost up to an order of magnitude slower than Join-Clust, while Join-Clust is infeasible for many datasets and parameter settings due to its excessive memory usage.

We propose a new clustering algorithm, *Spread*, that computes correct DBSCAN clusters using the asymmetric prefix index. Spread runs in linear space and does not need to materialize the (quadratic-size) neighborhoods. Spread avoids symmetric neighbor computations, therefore reducing the number of neighbors retrieved by Sym-Clust. So-called *backlinks* are introduced to achieve a correct clustering. Backlinks are dynamically added and removed as required and occupy only a small fraction of the memory that is used by materialized neighborhoods. Spread maintains a graph of subclusters in a disjoint-set data structure and guarantees that connected components in the resulting graph represent correct DBSCAN clusters.

In general, Spread can process data points in any *user-defined order* given an index that retrieves the lookahead neighbors, i.e., all data points that follow the query point in the user-defined processing order. In our usage scenario – set clustering – the processing order is defined by the set sizes (ties broken arbitrarily) and the asymmetric prefix index retrieves lookahead neighbors.

Summarizing, our contributions are the following:

- We propose *Spread*, a novel algorithm for partitioning large collections of sets into DBSCAN clusters. To the best of our knowledge, this is the first linear space DBSCAN-compliant algorithm that leverages the *asymmetric* prefix index for sets.
- We introduce the new concept of *backlinks* that keep sufficient information to build correct clusters independently of the processing order that the user imposes on Spread. We prove invariants for backlinks and the correctness of our approach.
- Our extensive empirical evaluation on 13 real-world datasets suggests that Spread is as fast as Join-Clust (that materializes all neighborhoods) while being competitive in memory usage with Sym-Clust (that computes all neighborhoods on the fly).

The remainder of this paper is organized as follows. In Section 2, we cover the background on ϵ-neighborhood and set similarity, indexing techniques for sets, and density-based clustering, and we define the *density-based set clustering problem*. Section 3 presents the two baseline approaches for density-based set clustering, Join-Clust and Sym-Clust. In Section 4, we present Spread, our time- and space-efficient solution for density-based set clustering. We evaluate our solution against the baseline algorithms and discuss the results in Section 5. Related work is summarized in Section 6. Finally, Section 7 concludes this paper.

2 BACKGROUND & PROBLEM DEFINITION

We revisit set similarity indexes and density-based clustering, and define our problem. To simplify the presentation, we focus on prefix indexes for the Hamming distance. Our results, however, extend to other distance and similarity measures (e.g., Jaccard or Cosine) and the respective indexes [12, 29, 40]. The required adaptations of the index that have been studied in the context of set similarity joins [29, 49] (e.g., prefix length, size lower bound, equivalent overlap) also apply to our scenario.

2.1 Set Similarity and ϵ-Neighborhood

R is a collection of n unique sets, each set $r \in R$ consists of unique tokens $t_1, \ldots, t_m$, $|r| = m$. The *processing order*, $\succ$, is a total order defined over R. The similarity between two sets r and s is assessed by the Hamming distance, $H(r, s) = |r \cup s| - |r \cap s|$, which counts the number of tokens that exist only in one of the sets, e.g., $H(r_1, r_2) = 4$ and $H(r_2, r_3) = 3$ for the sets in Figure 2.

The *ϵ-neighborhood* of set r includes r and all sets within distance ϵ from r, $N_\epsilon(r) = \{s \in R \mid H(r, s) \leq \epsilon\}$. A *region query* on r computes $N_\epsilon(r)$. A set r splits its ϵ-neighborhood into two disjoint parts based on the processing order: the *lookahead neighbors* that follow r in processing order, $N_\epsilon^{\succ}(r) = \{s \in N_\epsilon(r) \mid s \succ r\}$ and the *preceding neighbors*, $N_\epsilon^{\prec}(r) = \{s \in N_\epsilon(r) \mid s \prec r\}$.

2.2 Indexing Techniques for Sets

Prefix Filter and Inverted Index. A naive approach computes a region query $N_\epsilon(r)$ by verifying the predicate $H(r, s) \leq \epsilon$ for all sets $s \in R$. An effective indexing technique, which was originally developed for set similarity joins [2, 29], is based on the so-called prefix filter. The prefix, π_r, of a set r consists of the first π tokens of r according to some total token order (which must be the same for all sets). The prefix length depends on the distance function and is $\pi = \epsilon + 1$ for the Hamming distance. Figure 2 shows the prefix of three sets for distance threshold $\epsilon = 3$ and a numerical token order. The prefix filter works best if the tokens in the prefix are infrequent, thus the tokens are typically ordered by ascending global token frequency.

A set $s \in R$ can be in the ϵ-neighborhood $N_\epsilon(r)$ only if the prefixes of r and s share at least one token, i.e., $H(r, s) \leq \epsilon \Rightarrow \pi_r \cap \pi_s \neq \emptyset$ (assuming $|r| + |s| > \epsilon$; otherwise r and s are always similar). Therefore, if two sets do not share a token in the prefix, the pair can be safely pruned. If two sets r and s share a prefix token, (r, s) is a *candidate pair* and must undergo *verification*, i.e., the predicate $H(r, s) \leq \epsilon$ must be evaluated. Candidates that fail verification are *false positives*. Mann et al. [29] discuss efficient prefix-based verification.

Symmetric Prefix Index. An inverted index on the prefix tokens is used to retrieve candidate pairs efficiently. The inverted index maps prefix tokens to sets that contain that token in the prefix. A lookup of set r retrieves all lists of the prefix tokens of r. The

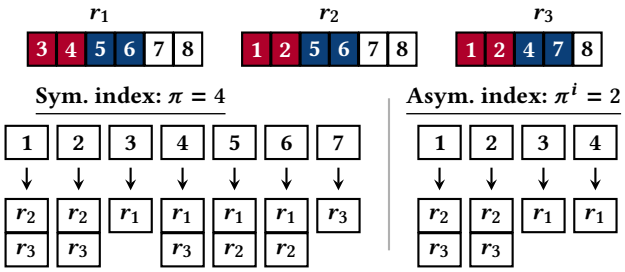

Figure 2: Symmetric and asymmetric prefix index, $\epsilon = 3$.

union of these lists (except r itself) are the candidates of r. The index is *symmetric* and returns the ϵ-neighborhood of r.

For example, the candidates for r_2 returned by the symmetric index, $\pi = \epsilon + 1 = 4$, in Figure 2 are $\{r_1, r_3\}$ (resulting from the union of $[r_2, r_3]$ for token 1, $[r_2, r_3]$ for token 2, $[r_1, r_2]$ for token 5, and $[r_1, r_2]$ for token 6). Candidate r_1 is a false positive since $H(r_1, r_2) > \epsilon$; r_3 is a true positive due to $H(r_2, r_3) \leq \epsilon$.

Asymmetric Prefix Index. We construct an *asymmetric* prefix index that returns only the lookahead neighbors, $N_\epsilon^{\succ}(r)$. To this end, we define a length-based processing order on R (longest to shortest): r precedes s if $|r| > |s|$, i.e., $|r| > |s| \Rightarrow s \succ r$; ties ($|r| = |s|$) are broken by the lexicographic order of the sorted sets.

Since we are interested only in sets $s \in N_\epsilon(r)$ that are no larger than r, $|s| \leq |r|$, we need to index only a subset of the prefix tokens: the tokens in the so-called *indexing prefix* [49]. The so-called *probing prefix* of the lookup set, r, remains of length $\pi = \epsilon + 1$. For the Hamming distance, the indexing prefix is of length $\pi^i = \left\lfloor \frac{\epsilon}{2} + 1 \right\rfloor$. For $\epsilon > 0$, the probing prefix is always longer than the indexing prefix, e.g., $\pi = 4$ and $\pi^i = 2$ for $\epsilon = 3$. A shorter prefix results in fewer candidates and renders the asymmetric index more effective than its symmetric counterpart.

In the case of r_2, the asymmetric index, $\pi^i = 2$, in Figure 2 returns only a true positive candidate, r_3. The false positive candidate, r_1, which is returned by the symmetric index, is avoided.

2.3 Density-Based Clustering

We formally define DBSCAN clusters and the related concepts. A set r represents a point to be clustered. The *density* of r is the number of ϵ-neighbors $|N_\epsilon(r)|$ (cf. Section 2.1).

Core, Border, Noise. A set r is a *core point* iff the ϵ-neighborhood of r contains at least minPts sets: r is core $\Leftrightarrow |N_\epsilon(r)| \geq \text{minPts}$. A set s is a *border point* iff it is in the ϵ-neighborhood of a core point r and s is not core: s is border $\Leftrightarrow s \in N_\epsilon(r) \wedge |N_\epsilon(r)| \geq \text{minPts} \wedge |N_\epsilon(s)| < \text{minPts}$. All remaining sets in R are *noise*. We denote the set of core and border points with $\mathcal{C}$ and $\mathcal{B}$, respectively. The set of noise points is $\mathcal{N} = R \setminus (\mathcal{C} \cup \mathcal{B})$.

Density-Reachability. Let $r, s \in R$ and r is core: s is *directly density-reachable* from r iff s is in the ϵ-neighborhood of r: $r \blacktriangleright s \Leftrightarrow s \in N_\epsilon(r)$. If there is a sequence of sets $r_1, r_2, \ldots, r_k$ with $r_1 = r$ and $r_k = s$, $r_i \blacktriangleright r_{i+1}$ for $1 \leq i < k$, s is *density-reachable* from r, denoted $r \blacktriangleright \ldots \blacktriangleright s$. Two sets r, s are *density-connected* if there is a set x s.t. both r and s are density-reachable from x.

A *density-based cluster* is a subset $C_i \subseteq R$ that satisfies two criteria [38]:

(1) *Maximality* $\mathbb{M}$*:* For any two sets $r, s \in R$, $r \in C_i$. If s is density-reachable from r, then $s \in C_i$. Formally,

$$\forall r, s \in R : r \in C_i \wedge r \blacktriangleright \ldots \blacktriangleright s \implies s \in C_i$$

Table 1: Notation overview.

Notation	Description
R	a collection of sets
r, s, x	sets of R
$\|r\|$	cardinality of set r
$r \prec s, r \succ s$	r precedes/succeeds s (in R)
$H(r, s)$	the Hamming distance of two sets r, s
π, π^i	probing/indexing prefix
ϵ	distance threshold
minPts	minimum density s.t. a set r is core
$N_\epsilon(r)$	full ϵ-neighborhood of r
$N_\epsilon^{\prec}(r), N_\epsilon^{\succ}(r)$	preceding/lookahead neighbors of r
$r \blacktriangleright s$	s is directly density-reachable from r
$r \blacktriangleright \ldots \blacktriangleright s$	s is density-reachable from r
$\mathcal{C}, \mathcal{B}, \mathcal{N}$	the set of core, border, and noise sets
C_i	a density-based cluster with id i

(2) *Connectivity* $\mathbb{C}$*:* For any two sets r, s in C_i, there is a set x that density-connects r and s. Formally,

$$\forall r, s \in R : r, s \in C_i \implies \exists x \in C_i : r \blacktriangleleft \ldots \blacktriangleleft x \blacktriangleright \ldots \blacktriangleright s$$

DBSCAN Clustering. A border point may be part of multiple density-based clusters such that the clusters overlap. We define the *DBSCAN clustering* that partitions the data into non-overlapping clusters. The standard DBSCAN algorithm [15] produces a DBSCAN clustering.

Definition 2.1. Let $R^* = R \setminus \mathcal{N}$ and $C_1, C_2, \ldots, C_k$ be density-based clusters such that $\bigcup_{i=1}^{k} C_i = R^*$. A DBSCAN clustering is a partitioning $\Gamma = \{C'_1, C'_2, \ldots, C'_k\}$, $C'_i \subseteq C_i$, such that $\bigcup_{i=1}^{k} C'_i = R^*$, $C'_i \cap C'_j = \emptyset$ for $i \neq j$.

A *subclustering* of a cluster C_i, $\psi_i = \{c_1, c_2, \ldots, c_l\}$, is a partitioning of C_i into $1 \leq l \leq |C_i|$ non-empty, disjoint subclusters, $c_j \subseteq C_i$, such that $\bigcup_{j=1}^{l} c_j = C_i$, $c_j \cap c_k = \emptyset$ for $j \neq k$.

A *subcluster graph* of R^* is an undirected graph in which nodes are subclusters and an edge between two nodes can only exist if the respective nodes are in the same DBSCAN cluster.

2.4 The DBSCAN Algorithm

The standard DBSCAN algorithm [15] forms clusters by repeatedly picking a seed point from the set of unvisited data points (initially all points are unvisited). If the seed is a core point, it forms a new cluster with all points that are density-reachable from the seed and are not yet assigned to a cluster. The set of density-reachable points is computed by recursively adding the ϵ-neighbors of all core points to the current cluster. The algorithm terminates when all points have been visited. Points that cannot be assigned to a cluster are noise.

2.5 Problem Statement

Definition 2.2 (Density-Based Set Clustering). Given a collection of sets R, a distance threshold ϵ, and the neighborhood density minPts, the goal is to find a DBSCAN clustering $\Gamma = \{C_1, C_2, \ldots, C_k\}$ of R.

For sets, asymmetric indexes with a lookahead neighbor function $N_\epsilon^{\succ}(r)$ promise the best performance (cf. Section 2.2). Given an ordering $\succ$ on R, we strive for a time- and space-efficient algorithm that solves the density-based set clustering problem with an asymmetric index.

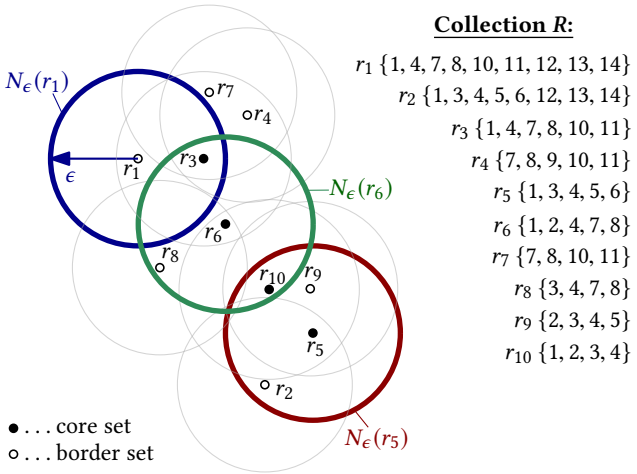

Figure 3: Running example, $\epsilon = 3$, minPts = 4.

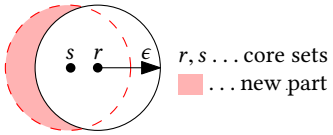

Figure 4: Redundant neighborhood queries.

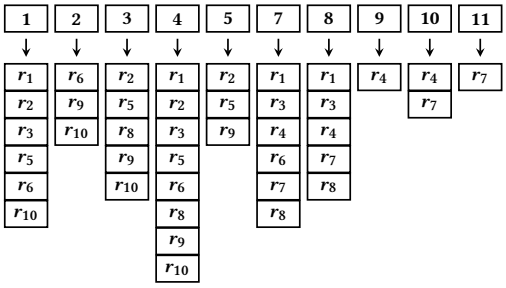

Figure 5: Symmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi = 4$.

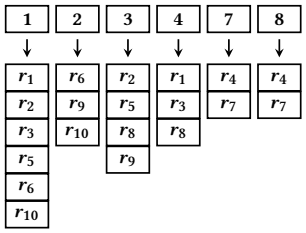

Figure 6: Asymmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi^i = 2$.

Running Example. Figure 3 shows an example collection R of ten sets, r_1–r_{10}, and their neighborhoods for Hamming distance $\epsilon = 3$. Sets r_3, r_5, r_6, and r_{10}, are core sets; all sets r_1–r_{10} form a single cluster.

3 BASELINE APPROACHES

This section presents two baseline solutions for the density-based set clustering problem. (1) *Sym-Clust* is memory-efficient and follows the standard DBSCAN approach with the symmetric prefix index to answer neighborhood queries on the fly. (2) *Join-Clust* is speed-optimized and materializes all ϵ-neighborhoods using a state-of-the-art set similarity join algorithm [2] (which leverages the asymmetric prefix index) before the standard DBSCAN algorithm is executed.

Both baselines leverage state-of-the-art set indexes. We are not aware of other previous solutions that can outperform Sym-Clust or Join-Clust for the density-based set clustering problem. Note that using the standard DBSCAN [15] (rather than some advanced techniques presented in later works, cf. Section 6) is not a limiting factor: Most of the overall execution time is spent computing the neighborhoods, and prefix-based indexes are highly efficient in combination with efficient verification [29].

3.1 Sym-Clust: DBSCAN with Inverted Index

When the standard DBSCAN algorithm (cf. Section 2.4) picks a seed point that is core, it forms a cluster with all points that are density-reachable from the seed. The density-reachable points are computed by pushing all core neighbors of the seed onto a stack. Then, each point on the stack is processed in the same manner (i.e., all its core neighbors are pushed onto the stack) until the stack is empty. All neighbors of core points retrieved in this process belong to the cluster.

The neighborhood queries will overlap to some extent. Assume r is processed before s, $s \in N_\epsilon(r)$, then $|N_\epsilon(s) \cap N_\epsilon(r)| \geq 2$ (at least r and s are in both neighborhoods). Since r assigns all its neighbors to the current cluster, only the non-overlapping neighbors of s, $N_\epsilon(s) \setminus N_\epsilon(r)$, will further increase the cluster.

Figure 4 illustrates this observation for the neighborhoods of two example points r (black circle) and s (red circle): only the new, non-overlapping area of $N_\epsilon(s)$ (shaded in red) is relevant for expanding the cluster.

The standard DBSCAN algorithm requires the use of a symmetric index since it assumes to see all neighbors of a point s when s is processed. The asymmetric index is not compatible with the standard DBSCAN algorithm: We cannot impose an order on the points such that all non-overlapping neighbors of s follow s in the processing order. Further, the size of the neighborhood of s, $|N_\epsilon(s)|$, is required to decide its core status.

Figure 5 shows the symmetric prefix index for our running example, $\epsilon = 3$, $\pi = \epsilon + 1 = 4$. We probe $r_8 = \{3, 4, 7, 8\}$. The prefix of r_8 consists of all tokens in r_8 (due to $|r_8| = \pi$). The union of the respective index lists yields the candidates $\{r_2, r_5, r_9, r_{10}, r_1, r_3, r_6, r_4, r_7\}$. Note that the candidates include both sets that are smaller and sets that are larger than r_8.

The so-called *length filter* [2], an optimization of the symmetric prefix index that also applies to its asymmetric counterpart, prunes candidates r_2 and r_1. Due to their length difference to r_8, they cannot be in the ϵ-neighborhood of r_8. By ordering the lists in processing order (i.e., longer sets precede shorter sets, as illustrated in Figure 5), the length filter can prune the head (sets that are too long) and the tail (sets that are too short) of a list without inspecting all elements in head and tail, respectively.

All candidates that are not pruned by the length filter must undergo verification. Only r_6 passes verification, therefore $N_\epsilon(r_8) = \{r_8, r_6\}$, and r_8 is classified as non-core (minPts = 4).

Complexity Analysis. We probe each set $r \in R$ against the index once. With cost C for an index lookup and $n = |R|$, the runtime is $O(n \cdot C)$; $C = O(n)$ since r may have $O(n)$ neighbors, thus the overall runtime of Sym-Clust is $O(n^2)$. The symmetric index is of linear size leading to space complexity $O(n)$.

Algorithm 1: Materialize-Neighborhoods(R, ϵ)

Input: Collection of sets R, distance threshold ϵ
Result: Materialized neighborhoods of R w.r.t. ϵ

$I \leftarrow$ Create-Index (R, ϵ);
$pairs \leftarrow \emptyset$ // Result set of similar pairs
foreach $r \in R$ *in processing order* **do**
  $M \leftarrow$ Probe (r, I, ϵ) // candidates with prefix overlaps
  foreach $(s, po) \in M$ **do** // po . . . prefix overlap
    if Verify-Pair (r, s, ϵ, po) **then**
      $pairs \leftarrow pairs \cup \{(r, s)\}$
$neighborhoods \leftarrow$ new associative array of size $|R|$;
foreach $(r, s) \in pairs$ **do**
  $neighborhoods[r] \leftarrow neighborhoods[r] \cup \{s\}$;
  $neighborhoods[s] \leftarrow neighborhoods[s] \cup \{r\}$;
return *neighborhoods*

Algorithm 2: Create-Index (R, ϵ)

Input: Collection of sets R, distance threshold ϵ
Result: Inverted index of R w.r.t. ϵ

$I \leftarrow \emptyset$ // inv. index of set prefixes, $I_{r[p]}$ . . . list of token $r[p]$
foreach $r \in R$ **do**
  $\pi \leftarrow \left\lfloor \frac{\epsilon+2}{2} \right\rfloor$ // indexing prefix length of r
  for $p \leftarrow 1$ **to** π **do** $I_{r[p]} \leftarrow I_{r[p]} \cup \{r\}$;
return I

Algorithm 3: Probe (r, I, ϵ)

Input: Probing set r, inv. index I, distance threshold ϵ
Result: Set of candidates for r w.r.t. ϵ

// M maps a candidate s to its prefix overlap with r
$M \leftarrow$ new associative array // candidates
$\pi \leftarrow \epsilon + 1$ // probing prefix length of r
$lb_r \leftarrow |r| - \epsilon$ // size lower bound wrt. r
for $p \leftarrow 1$ **to** π **do**
  foreach $s \in I_{r[p]}$ *in proc. order* **do** // list of token $r[p]$
    if $|s| < lb_r$ **then break** ;
    else // add candidate
      if $s \notin M$ **then** $M[s] \leftarrow 0$; // init.
      $M[s] \leftarrow M[s] + 1$ // incr. overlap of (r, s)
return M

Algorithm 4: Verify-Pair (r, s, ϵ, po)

Input: Probing set r, candidate set s, distance threshold ϵ, prefix overlap po
Result: True iff r and s are similar w.r.t. ϵ, false otherwise

// cf. Mann et al. [29] for prefixes, equiv. overlaps, and Verify proc.
$\pi_r, \pi_s \leftarrow$ probing resp. indexing prefix length of r resp. s;
$w_r, w_s \leftarrow \pi_r$- resp. π_s-th token in r resp. s;
$t \leftarrow$ equivalent overlap for r, s, and ϵ;
if $w_r < w_s$ **then**
  return Verify $(r, s, t, po, \pi_r + 1, po + 1)$
return Verify $(r, s, t, po, po + 1, \pi_s + 1)$

3.2 Join-Clust: Materialized Neighborhoods

Join-Clust executes a set similarity self-join on R and materializes the ϵ-neighborhoods in main memory. The self-join traverses all sets of $r \in R$ in processing order and computes their lookahead neighbors, $N_\epsilon^\succ(r)$. The lookahead neighbors of r are appended to the list of r's neighbors, and r is appended to the neighborhood lists of all $s \in N_\epsilon^\succ(r)$. After processing all sets, the neighborhood list of each set $r \in R$ is complete and stores $N_\epsilon(r)$.

Next, standard DBSCAN (cf. Section 2.4) is executed to form clusters using the materialized neighborhoods. Algorithms 1–4 implement the similarity join with neighborhood materialization, index creation, probing, and efficient verification [29].

Mann et al. [29] found that the prefix-based index in combination with the length filter can be considered state of the art given an efficient verification procedure (which we use).

Figure 6 shows the asymmetric prefix index for our running example, $\epsilon = 3$, $\pi^i = 2$. We probe $r_8 = \{3, 4, 7, 8\}$ and look up the lists of the tokens 3, 4, 7, and 8 (the length of the probing prefix is $\pi = 4$). Since we are only interested in the lookahead neighbors, i.e., all neighbors that follow r_8 in processing order, we need to inspect the lists only starting from the point where r_8 or a set $r_i \succ r_8$ appears. The length filter does not prune any candidate in this example, and the candidate set is $\{r_9\}$. Since $H(r_8, r_9) > \epsilon$, the lookahead neighborhood of r_8 is empty, $N_\epsilon^\succ(r_8) = \emptyset$.

In the context of the self-join, r_8 will be retrieved as a lookahead neighbor of r_6, which is processed before r_8. Therefore, the neighborhood list of r_6 will store r_8 and vice versa.

Join-Clust produces fewer candidates than Sym-Clust and is therefore faster. However, the efficiency of Join-Clust comes at the cost of a larger memory footprint since all neighborhoods must be materialized.

Complexity Analysis. A neighborhood query is a constant-time lookup and a traversal of $O(|N_\epsilon(r)|)$ neighbors. In the worst case, the join reports $O(n^2)$ pairs. Consequently, materializing the neighborhoods takes $O(n^2)$ time and space for $n = |R|$. The asymmetric prefix index requires only $O(n)$ space and does not dominate memory usage.

4 THE SPREAD ALGORITHM

We present *Spread*, a novel time- and space-efficient solution for the density-based clustering problem. Spread leverages the effective asymmetric index and clusters all sets by traversing the sets in processing order. We identify key challenges that must be solved, discuss the algorithm, prove its correctness, analyze time and space complexity, and sketch a multi-core extension.

4.1 Key Challenges

Since Spread uses an asymmetric neighborhood index, a processing order, $\succ$, must be imposed on the data points, and an index lookup of query point r retrieves only the lookahead neighbors, $N_\epsilon^\succ(r)$. To achieve a correct clustering without materializing the neighborhoods, three key challenges must be solved.

In the following discussion, we assume that all sets of R are processed in processing order. When the *current set* r_i is to be processed, we know the core status of all preceding sets, $r_j \prec r_i$, but we do not know the core status of any unprocessed sets, $r_k \succ r_i$. We further assume that all sets $r \in R$ that are directly density-reachable from any r_j that precedes r_i (i.e., are neighbors of a core point $r_j \prec r_i$) are assigned to the same cluster as the core point r_j; this may also include sets $r_k \succ r_i$ that have not been processed yet.

Core Status. A set r_i is core if $|N_\epsilon(r_i)| \geq$ minPts. Sym-Clust and Join-Clust have access to the full neighborhood, $N_\epsilon(r_i)$, thus deciding the core status of r_i is trivial. In contrast, Spread sees only the lookahead neighbors, $N_\epsilon^\succ(r_i)$. To identify the core status of r_i, however, additional knowledge about the size of the preceding neighborhood, $N_\epsilon^\prec(r_i)$, is required.

Consider probing $r_i = r_5$ in our running example. According to our assumptions, the core status of sets r_1–r_4 is known (only r_3 is core), and all neighbors of r_3 are in cluster $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$. An index lookup of r_5 returns $N_\epsilon^\succ(r_5) = \{r_9, r_{10}\}$. Since $\left|N_\epsilon^\succ(r_5)\right| + 1 = 3 < 4 =$ minPts we cannot decide if r_5 is core. In fact, r_5 should be classified core since the full neighborhood is $N_\epsilon(r_5) = \{r_2, r_5, r_9, r_{10}\}$ (cf. red circle in Figure 3).

Border vs. Noise. Assume that the current set r_i is a non-core point that is not assigned to any cluster. We need to decide if r_i is border or noise. A border point has at least one core point in its neighborhood. None of the preceding neighbors, $r_j \in N_\epsilon^\prec(r_i)$, is core, otherwise r_i would be assigned to the cluster of r_j. Thus, r_i is core iff one of the lookahead neighbors is core. Unfortunately, we do not know the core status of the lookahead neighbors and can therefore not label r_i as border or noise.

Assume a core point, $r_k \in N_\epsilon^\succ(r_i)$, among the lookahead neighbors of r_i. When r_k is processed, r_k will not see r_i in its lookahead neighborhood since $r_i \prec r_k$. Therefore, r_i will not be included into the cluster of r_k and will wrongly be classified noise. The challenge is to correctly decide the border status of r_i despite seeing only the lookahead neighbors of r_i and r_k.

In our running example, r_1 is processed first. Thus, no core points are known and no clusters exist. $N_\epsilon^\succ(r_1) = \{r_3\}$ and r_1 remains noise (cf. blue circle in Figure 3). When the neighbor r_3 of r_1 is processed, r_3 will be detected as a core point and start a new cluster. However, since r_3 only sees its lookahead neighbors, $N_\epsilon^\succ(r_3) = \{r_4, r_6, r_7\}$, r_1 is not included into the cluster and is not detected as a border point.

Disconnected Clusters. Assume that the current set r_i is core and there is a core point $r_j \prec r_i$ in a cluster C_j, $r_i \notin C_j$. The current set r_i will assign all its lookahead neighbors to its cluster, $C_i = C_i \cup N_\epsilon^\succ(r_i)$ (C_i can be a new cluster started by r_i or an existing cluster to which r_i belongs). Unfortunately, we cannot assume that C_i and C_j are indeed distinct clusters: there can be a core point $r_k \succ r_i$ that density-reaches both r_i and r_j, i.e., C_i and C_j should form a single cluster. In general, multiple subclusters of the same DBSCAN cluster may grow independently. The challenge is to identify subclusters that should be merged and to merge them efficiently.

We process the current set $r_i = r_6$ in our running example. According to our assumptions, we know that r_3 and r_5 are core and we are aware of two clusters, $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$, $C_5 = \{r_2, r_5, r_9, r_{10}\}$. In addition, assume that we know that r_6 is core. Then, r_6 extends its current cluster, C_3, with its lookahead neighbors $N_\epsilon^\succ(r_6) = \{r_8, r_{10}\}$. Note that r_{10} is already part of cluster C_5. Since we do not know the core status of r_{10}, we cannot decide if C_5 and C_3 should be merged into a single cluster. If r_{10} is core, r_5 and r_3 are density-reachable from r_{10} and should be in the same cluster. If r_{10} is a border point, however, the clusters must not be merged, and r_{10} can be assigned to either C_5 or C_3.

4.2 Data Structures

Disjoint-Set. The disjoint-set (or union-find) data structure maintains a dynamic collection of non-overlapping sets for n objects in $O(n)$ space [10, 41]. A typical use case is the efficient computation of (minimum) spanning trees. It supports three operations: (1) For a given element u, make_set(u) creates a new (singleton) set that contains u. (2) The union(u, v) operation merges the two sets that contain u resp. v into a new set. (3) find_set(u) returns the representative for the set that contains u or ∞ if u is not found. The amortized worst-case time complexity is $\Theta(\alpha(n))$ for all operations, $\alpha(.)$ being the inverse Ackermann function. In practice, $\alpha(n)$ is considered a constant. In our setting, set elements are subclusters, and the disjoint-set data structure links subclusters that belong to the same DBSCAN cluster.

Backlinks. The backlinks data structure of a set $r \in R$ is a collection of unique references to other sets s that precede r, $s \prec r$. The backlinks bl support the add operation, $bl \cup \{s\}$, which adds a reference to a new set s in time $O(1)$ (on average). Depending on the type of sets that are referenced in the backlinks, we distinguish core and non-core backlinks, denoted *c_bl* and *nc_bl*, respectively. We implement backlinks as unordered sets of integer identifiers.

4.3 The Algorithm

Algorithm 5 shows the pseudocode of Spread. We use the following notation: r is the current probing set, $s \succ r$ is a lookahead neighbor, and $x \prec r$ is a preceding neighbor. Initially, all sets are noise, i.e., their cluster identifier is $-\infty$, $\forall r \in R : r.cid = -\infty$. Although we initialize all sets in Algorithm 5 explicitly (lines 3–4), this can also be done during indexing (cf. Algorithm 2).

Algorithm Outline. Spread proceeds in three main steps: (1) A counter and the processing order guarantee that the cardinality of the ϵ-neighborhood is known when a set is processed despite using the asymmetric prefix index. (2) Each set is assigned to a subcluster solely based on its lookahead neighboorhood. Subclusters of the same DBSCAN cluster are linked in a subcluster graph. Backlinks ensure that we do not miss border sets or links between subclusters. (3) Each connected component in the subcluster graph represents a DBSCAN cluster.

Core Status. A set r is core if $|N_\epsilon(r)| \geq$ minPts. In Spread, however, only $N_\epsilon^\succ(r)$ is computed. To capture the cardinality of $N_\epsilon^\prec(r)$, we store a density counter with each set r, denoted $r.dens$. Initially, $\forall r \in R : r.dens = 1$. For every lookahead neighbor $s \in N_\epsilon^\succ(r)$, $r.dens$ and $s.dens$ are incremented (due to the symmetry of the distance). Core set identification is highlighted in green.

Border vs. Noise. A probing set r that is not core is a border set iff $\exists y \in N_\epsilon(r) : y$ is core. Due to our processing order and the fact that only $N_\epsilon^\succ(r)$ is computed, the existence of a core neighbor y may be unknown when r is probed. However, for each $s \in N_\epsilon^\succ(r)$, we know that r is part of $N_\epsilon^\prec(s)$. We store this information by adding r to the non-core backlinks $nc\_bl[s]$ of each $s \in N_\epsilon^\succ(r)$ (lines 31–33). Then, the first $s \in N_\epsilon^\succ(r)$ that becomes core claims r (and all other unassigned sets in $nc\_bl[s]$) as border point for its subcluster. If none of the neighbors $s \in N_\epsilon^\succ(r)$ becomes core, then r remains noise. Lines 26–30 deal with a special case: If any $s \in N_\epsilon^\succ(r)$ is already core when r is probed, then s claims r immediately without adding r to its non-core backlinks. The relevant code lines are marked in red.

Subcluster Linkage. If the probing set r is core and a core neighbor y is part of another subcluster, the subclusters of r and y must be linked in our subcluster graph. The subcluster graph represents all connected components of subclusters, each of which is

a DBSCAN cluster. We use the disjoint-set data structure ds to track the connected components. Two subclusters u, v are linked by $ds.\text{union}(u, v)$. We may not be able to determine if there is a set $s \in N_\epsilon^\succ(r)$ that is core before s is probed. We use the core backlinks, c_bl, to book-keep potential subclusters for linkage: r adds its subcluster identifier to $c\_bl[s]$ of each $s \in N_\epsilon^\succ(r)$ (lines 22-23). After $N_\epsilon^\succ(r)$ has been processed, a link between the subcluster of r and every entry in $c\_bl[r]$ is created (line 24). The special case when s is already core allows us to create the link immediately without using core backlinks (lines 20–21). Linkage is only required if two subclusters coalesce (condition in line 19). Otherwise, r simply claims $s \in N_\epsilon^\succ(r)$ for its subcluster (lines 17–18). Linkage of subclusters is highlighted in blue ▢.

All backlinks of r are released after r has been processed to save memory (line 34). The subcluster graph in ds is used to assign consistent cluster IDs in a final scan over R (lines 35–36).

4.4 Correctness

We show that Algorithm 5 partitions R into DBSCAN clusters (cf. Definition 2.1). Set $r_i \in R$ is the i-th set of R in processing order. We prove the correctness by induction over i and increasing subsets $R^i \subseteq R$. $R^0 = \emptyset$, $R^i = R^{i-1} \cup \{r_i\} \cup N_\epsilon^\succ(r_i)$ for $1 \leq i \leq n = |R|$, thus $R^n = R$. Due to space constraints, we omit the full proofs and only provide the invariants that must be shown.

Core Status. The core status of set r_i is determined in the i-th iteration of the main loop. r_i is core if $\text{minPts} \leq |N_\epsilon(r_i)| = 1 + \left|N_\epsilon^\prec(r_i)\right| + \left|N_\epsilon^\succ(r_i)\right|$. In line 5, $r_i.dens = 1 + \left|N_\epsilon^\prec(r_i)\right|$. Lines 6–11 compute $N_\epsilon^\succ(r_i)$. The index lookup in line 6 returns candidate set M, $N_\epsilon^\succ(r_i) \subseteq M \subseteq \{s \mid s \succ r_i\}$. Every set $s \in M$ is verified in line 9 such that $N_\epsilon^\succ(r_i)$ is available starting from line 12.

Lemma 4.1. *Algorithm 5 correctly identifies* all *core sets in R.*

Proof Sketch. We show that at the start of the i-th iteration in line 5, for all r_k and r_j, $1 \leq k < i \leq j$ the following invariants hold: (I1) $r_k.dens = |N_\epsilon(r_k)|$; (I2) $r_j.dens = 1 + \left|\left\{r_k \mid r_j \in N_\epsilon^\succ(r_k)\right\}\right|$, i.e., $r_i.dens = 1 + \left|N_\epsilon^\prec(r_i)\right|$. Further, (I3) in line 12 of the i-th iteration, $r_i.dens = |N_\epsilon(r_i)|$, i.e., Algorithm 5 correctly identifies the core status of r_i. □

Border vs. Noise. Lines 25–33 cover the case that r_i is not core. If any $s \in N_\epsilon^\succ(r_i)$ qualifies as core, s claims r_i. Otherwise, r_i is stored in the non-core backlinks $nc\_bl[s]$ of every $s \in N_\epsilon^\succ(r_i)$ (lines 31–33). The next core neighbor in processing order claims r_i (lines 14–15) such that all border sets are assigned to a cluster.

Lemma 4.2. *Algorithm 5 correctly clusters* all *border sets in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all border sets $r_k \in \mathcal{B}$, $1 \leq k < i$: if r_k is not stored in $nc\_bl[s]$ for any $s \in N_\epsilon^\succ(r_k)$, $s = r_i$ or $s \succ r_i$, then r_k is assigned to the cluster of a core point in its neighborhood. □

Subcluster Linkage. Lines 12–24 cover the case that r_i is core. Each core point may form a subcluster on its own or together with other core points. We must ensure that all subclusters of the same DBSCAN cluster are linked in the disjoint-set, ds.

Lemma 4.3. *Algorithm 5 correctly links* all *subclusters in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all core neighors $c \in CN(r_k) = N_\epsilon(r_k) \cap C$ of a core set $r_k \in C$, $1 \leq k < i$: (a) c and r_k have the same cluster representative (in ds), or (b) c is stored in some $c\_bl[s]$, $s \in N_\epsilon^\succ(r_k)$, $s = r_i$ or $s \succ r_i$. □

Algorithm 5: Spread(R, ϵ, minPts)

Input: Collection of sets R, distance threshold ϵ, min. density minPts
Result: A correct DBSCAN clustering of R w.r.t. ϵ, minPts

```
1  ds ← new disjoint-set; nc_bl, c_bl ← new backlinks;
2  I ← Create-Index(R, ε);
3  foreach r ∈ R do
4      r.dens ← 1; r.cid ← −∞; ds.make_set(r.id);
5  foreach r ∈ R in processing order do
6      M ← Probe(r, I, ε);
7      N_ε^≻(r) ← ∅;
8      foreach (s, po) ∈ M do // po ... prefix overlap
9          if Verify-Pair(r, s, ε, po) then
10             r.dens ← r.dens + 1; s.dens ← s.dens + 1;
11             N_ε^≻(r) ← N_ε^≻(r) ∪ {s};
12     if r.dens ≥ minPts then // r is core
13         if r.cid = −∞ then r.cid ← r.id;
14         foreach x ∈ nc_bl[r] do // claim border sets x ≺ r
15             if x.cid = −∞ then x.cid ← r.cid;
16         foreach s ∈ N_ε^≻(r) do // s ≻ r
17             if s.cid = −∞ then // claim unclaimed s ≻ r
18                 s.cid ← r.cid
19             else if r.cid ≠ s.cid then // s already claimed
20                 if s.dens ≥ minPts then // s is core
21                     ds.union(r.cid, s.cid) // link subclusters
22                 else // remember core neighbor r
23                     c_bl[s] ← c_bl[s] ∪ {r.cid}
24         foreach x ∈ c_bl[r] do ds.union(r.cid, x);
25     else // r is not core, i.e., r.dens < minPts
26         if r.cid = −∞ then // claim potential border set r
27             foreach s ∈ N_ε^≻(r) do
28                 if s.dens ≥ minPts then // s is core
29                     if s.cid = −∞ then s.cid ← s.id;
30                     r.cid ← s.cid; break;
31         if r.cid = −∞ then // remember potential border set r
32             foreach s ∈ N_ε^≻(r) do
33                 nc_bl[s] ← nc_bl[s] ∪ {r}
34     release c_bl[r] and nc_bl[r] // not needed anymore
35 foreach r ∈ R do // final assignment of cluster IDs
36     if r.cid ≠ −∞ then r.cid ← ds.find_set(r.cid);
```

Theorem 4.4. *Algorithm 5 returns a correct set clustering* $\Gamma = \{C_1, C_2, \ldots, C_k\}$ *of R according to Definition 2.1.*

Proof Sketch. By Lemmata 4.1–4.3 and due to our final scan over R (lines 35–36), $x.cid = ds.\text{find\_set}(x.cid)$ holds for all $x \in R$. Initially, $x.cid = -\infty$ for all $x \in R$. The cluster IDs are updated only for border and core sets. Consequently, $x.cid = -\infty$ holds for all $x \in R \setminus (C \cup \mathcal{B}) \equiv \mathcal{N}$, i.e., also noise is correctly identified. □

4.5 Complexity Analysis

Memory. The asymmetric prefix index requires $O(n)$ space. In addition, Spread maintains the following data structures. (i) A density counter for each set $r \in R$ requires $O(n)$ space. (ii) A disjoint-set data structure with at most $O(n)$ entries, i.e., the disjoint-set structure requires $O(n)$ space [41]. (iii) In the worst case, we allocate two backlink structures for each $r \in R$, i.e., $O(n)$ backlinks. We release $c\_bl[r]$ and $nc\_bl[r]$ after probing r. Backlinks are only extended in lines 23 and 33. However, both lines are executed iff $\nexists s \in N_\epsilon^>(r) : s$ is core. Set s is core iff $s.dens \geq$ minPts, and the density is updated for every neighbor, therefore any backlink holds at most minPts entries. As a result, no more than $O(n \cdot \text{minPts})$ entries are allocated, thus requiring $O(n)$ space since minPts and ϵ are constants. *Runtime.* For each $r \in R$, we process $O\left(\left|N_\epsilon^>(r)\right|\right)$ neighbors and the backlinks of r if it is core. Recall that the disjoint-set operations take constant time. Therefore, the final for-loop (lines 35–36) runs in $O(n)$ time. Overall, Spread runs in $O\left(n^2\right)$ time and $O(n)$ space.

4.6 Multi-core Extension

Spread is designed as a single-core algorithm. We sketch an extension to multi-core processors that requires little synchronization between threads. Our extension is based on the observation that Spread spends most of the runtime in neighborhood computations (lines 6-11). While for some datasets the neighborhood computation accounts for only about half of the overall runtime (e.g., 55% for ORKUT, $\epsilon = 3$), for the configuration with the highest runtime in our experiments (CELONIS1, $\epsilon = 5$), Spread spends over 99% of the runtime in computing the neighborhoods.

We distribute the workload to $k+1$ threads, $T_1, T_2, \ldots, T_{k+1}$. Threads $T_1 - T_k$ are responsible for the neighborhood computations (lines 6-11), T_{k+1} performs the actual clustering (lines 12-34). The runtime of the other steps in the algorithm is negligible.

Neighborhood Computation. Let $r_i \in R$, $1 \leq i \leq |R|$ be the i-th set of R in processing order. Thread T_j, $1 \leq j \leq k$, computes the neighborhoods $N_\epsilon^>(r_i)$ of all r_i with $j = i \mod k$ (i.e., round robin). Each thread processes the assigned sets r_i in processing order (i.e., increasing values of i). The neighborhood computation in Algorithm 5 is interleaved with updating the density counters of r_i and its neighbors. Only this step requires synchronization (e.g., using atomic writes) since multiple threads may access the same counter concurrently. We do not expect congestions since the density updates are distributed over all neighbors.

Cluster Scan. Thread T_{k+1} scans the sets in processing order and performs the steps in lines 12-34 (maintain backlinks and disjoint-set, assign preliminary cluster IDs). After processing a set r_i, the memory for the neighbors of r_i is released.

Synchronization. We need to make sure that T_{k+1} processes set r_i only after r_i's neighbors have been computed. This can be achieved with a lock (implemented as condition variable[2]) on r_i that is held by T_j, $j \leq k$, until the neighborhood of r_i is computed. T_{k+1} needs to get the lock on r_i before processing it.

Memory. T_{k+1} releases the neighbors after processing them. If the parallel neighborhood computation is faster than T_{k+1}, the precomputed neighborhoods will fill up the memory. This is avoided with a shared counter that is incremented by $T_1 - T_k$ (when they process a new set r_i) and is decremented by T_{k+1} (after processing r_i). The neighborhood computation of r_i is postponed until the counter is below some threshold that bounds the number of concurrently materialized lookahead neighborhoods.

[2] A queue of threads waiting for a condition to become true.

Table 2: Characteristics of datasets.

Dataset	Coll. Size	Set Size		Univ. Size
		avg.	max.	
BMS-POS[4]	$3.2 \cdot 10^5$	9.3	164	$1.7 \cdot 10^3$
FLICKR[5]	$1.2 \cdot 10^6$	10.1	102	$8.1 \cdot 10^5$
KOSARAK[6]	$6.1 \cdot 10^5$	11.9	$2.5 \cdot 10^3$	$4.1 \cdot 10^4$
LIVEJ[7]	$3.1 \cdot 10^6$	36.4	300	$7.5 \cdot 10^6$
ORKUT[7]	$2.7 \cdot 10^6$	119.7	$4.0 \cdot 10^4$	$8.7 \cdot 10^6$
SPOT[8]	$4.4 \cdot 10^5$	12.8	$1.2 \cdot 10^4$	$7.6 \cdot 10^5$
CELONIS1	$8.2 \cdot 10^6$	20.3	91	$1.2 \cdot 10^4$
CELONIS2	$2.6 \cdot 10^6$	22.1	130	$3.5 \cdot 10^3$

5 EXPERIMENTAL EVALUATION

Algorithms. We compare our solution, Spread, against the two baseline approaches Sym-Clust and Join-Clust (cf. Section 3). All algorithms are single-threaded C++ implementations (2017 standard). Our implementations of Spread, Join-Clust, and the index of Sym-Clust follow the guidelines by Mann et al. [29], e.g., regarding symmetric and asymmetric prefix index, candidate generation, and optimized prefix-based verification.

Datasets. We execute all experiments on 13 real-world datasets: (a) Nine of the datasets where previously used for benchmarking set similarity joins [16, 29]: BMS-POS, DBLP, ENRON, FLICKR, KOSARAK, LIVEJ, NETFLIX, ORKUT, and SPOT. For a description of the datasets and preprocessing instructions[3] we refer to Mann et al. [29]. (b) Four large real-world datasets from the process mining domain, CELONIS1–4, that store one set per process. Compared to most datasets of the join benchmark, the universe size of these datasets is rather small. Table 2 summarizes important characteristics of our benchmark data.

Due to space constraints we omit detailed results for the following datasets: (a) DBLP, ENRON, and NETFLIX show very low runtimes (< 4s) and a small and stable memory footprint (< 1GiB) for all algorithms and configurations. (b) CELONIS3–4 show results similar to the other process mining datasets.

Parameters. The algorithms take two parameters: the neighborhood radius, ϵ, and the density, minPts. Typically, density-based clustering is sensitive to ϵ and quite robust to minPts. In our experiments, we vary both parameters: $\epsilon \in \{2, \mathbf{3}, 4, 5\}$ and minPts $\in \{2, 4, 8, \mathbf{16}, 32, 64, 128\}$ (defaults in bold font).

Environment. All experiments have been conducted on a 64-bit machine with 2 physical Intel Xeon E5-2630 v3 CPUs, 2.40GHz, 8 cores each (i.e., 16 logical processors, hyper-threading disabled). The cores share a 20MiB L3 cache and have another 256KiB of independent L2 cache. The system has 96GiB of RAM and runs Debian 10 Buster (Linux 4.19.0-12-amd64 #1 SMP Debian 4.19.152-1 (2020-10-18)). Our code is compiled with `clang`[9] version 7, highest optimization level (`-O3`). The runtime is measured with `clock_gettime`[10] at process level, memory usage is the `heap`

[3] http://ssjoin.dbresearch.uni-salzburg.at/datasets.html
[4] BMS-POS: http://www.kdd.org/kdd-cup/view/kdd-cup-2000 [52]
[5] FLICKR: Bouros et al. [6]
[6] KOSARAK: http://fimi.uantwerpen.be/data/
[7] LIVEJ, ORKUT: http://socialnetworks.mpi-sws.org/data-imc2007.html [30]
[8] SPOT: Pichl et al. [33]
[9] https://releases.llvm.org/7.0.0/tools/clang/docs/ReleaseNotes.html
[10] https://man7.org/linux/man-pages/man2/clock_gettime.2.html

Table 3: Index & cluster statistics for $\epsilon = 3$, minPts = 16.

(a) BMS-POS.

	Candidates	True Positives	Clusters
Sym-Clust	$3.9 \cdot 10^9$	$38.0 \cdot 10^6$	1
Join-Clust	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1
Spread	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1

(b) KOSARAK.

	Candidates	True Positives	Clusters
Sym-Clust	$40.7 \cdot 10^9$	$2.8 \cdot 10^9$	5
Join-Clust	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5
Spread	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5

(c) CELONIS1.

	Candidates	True Positives	Clusters
Sym-Clust	$644.6 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Join-Clust	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Spread	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075

peak of Linux `memusage`[11] (using `LD_PRELOAD`). A single instance is executed at a time with no other load on the machine.

5.1 Index & Cluster Statistics

We compare the number of candidates, true positives, and the number of clusters. The numbers are sums over all region queries. Table 3 shows the results obtained for BMS-POS, KOSARAK, and CELONIS1. We observe that Spread produces exactly the same number of candidates as Join-Clust since both solutions use the asymmetric index. Sym-Clust generates significantly more candidates due to the symmetric prefix index and the symmetric distance computations. For CELONIS1, Spread and Join-Clust verify about 5 times fewer candidates compared to Sym-Clust.

5.2 Runtime Efficiency

We measure the overall runtime, i.e., the CPU time that is required to cluster all sets into DBSCAN clusters (excluding the time to load the data from disk). Figure 7 shows the results for varying ϵ (minPts = 16). We observe that Sym-Clust is not competitive in terms of overall runtime in most cases. For all datasets, except KOSARAK and SPOT, the runtime of Sym-Clust increases much faster with ϵ than observed for Join-Clust and Spread. This is mainly due to the use of the symmetric prefix index (more candidates) and redundant computations (symmetric pairs).

Our experiments reveal that Join-Clust suffers from the following issues: (i) High runtimes for LIVEJ, ORKUT, and SPOT due to the expensive neighborhood materialization. (ii) Join-Clust runs out of memory for many instances (missing points in plots), in particular for FLICKR (any ϵ), KOSARAK ($\epsilon \geq 4$), LIVEJ, ORKUT, and SPOT ($\epsilon \geq 3$).

Spread outperforms its competitors in most settings and is competitive with Join-Clust otherwise (cf. Figures 7a, 7g, and 7h). For CELONIS1 and CELONIS2, Spread outperforms Sym-Clust by almost an order of magnitude and is competitive with Join-Clust.

Figure 8 shows the runtime results for varying minPts values ($\epsilon = 3$). We observe that the runtime of all three solutions is quite robust to minPts. The insights are similar for all datasets and values of ϵ. We include the plots for BMS-POS and KOSARAK.

[11] https://man7.org/linux/man-pages/man1/memusage.1.html

5.3 Memory Efficiency

We study the memory usage of Join-Clust, Sym-Clust, and Spread. All three solutions store (i) the collection, (ii) the inverted index, (iii) the candidates, and (iv) the result of a region query on the heap. The symmetric prefix index of Sym-Clust is larger than the asymmetric index, but still linear in the collection size. Sym-Clust generates more candidates than Join-Clust and Spread (cf. Section 5.1), which both use the asymmetric prefix index . Join-Clust materializes all neighborhoods in main memory. Sym-Clust and Spread materialize only a single neighborhood at a time. Spread stores also backlinks and the disjoint-set in main memory.

Figure 11 shows our results for varying ϵ (minPts = 16, y-axis log scale). Join-Clust runs out of memory for many instances (cf. Section 5.2). The neighborhood materialization in Join-Clust can be memory intensive even for small values of ϵ. We observe different growth rates with increasing radius ϵ, which we attribute to the different neighborhood sizes. The memory consumption of Sym-Clust is significantly lower and robust to varying ϵ. Spread shows a similar behavior. In some cases (e.g., LIVEJ, ORKUT), Spread occupies even less memory than Sym-Clust. When few backlinks are materialized, the smaller asymmetric prefix index of Spread outweighs the storage overhead for the backlinks.

Figure 12 shows the memory usage over minPts ($\epsilon = 3$, log-log scale). The memory consumption of Sym-Clust and Join-Clust is stable w.r.t. increasing values of minPts, while the memory usage of Spread slightly increases. This is due to the number of concurrently stored backlinks: the larger minPts, the higher the chance that a succeeding core neighbor is not yet classified, which triggers the creation of a backlink entry. The memory grows slowly with increasing minPts and does not limit the scalability of Spread. We include the results for BMS-POS and KOSARAK, $\epsilon = 3$; other datasets and ϵ values show similar results.

Backlinks Peak. We evaluate the effect of releasing the backlinks of a set in Spread after the set has been processed (cf. line 34, Algorithm 5). Figures 10 and 14 show the peak number of allocated backlinks relative to the maximum number of backlinks for varying ϵ (minPts = 16) and minPts ($\epsilon = 3$), respectively. Since two backlink structures, core (green) and non-core (orange), are maintained for each set in R, at most $2|R|$ backlinks can be allocated (light blue). Deallocating the backlinks of probed sets is highly effective: Only a small fraction of the maximum number of backlinks is allocated at any point in time. For increasing values of ϵ and minPts also the number of allocated backlinks grows.

5.4 Scalability

We evaluate the scalability of Spread and its competitors to increasing data sizes. To this end, we increase the size of BMS-POS and KOSARAK using the procedure of Vernica et al. [42]. This approach does not affect the token universe, and the number of similar pairs in the dataset increases linearly with the data size.

Figure 9 (runtime) and Figure 13 (main memory) report the results for our default parameter setting. Spread shows runtimes similar to Join-Clust and outperforms Sym-Clust by a factor of about 12 (BMS-POS) resp. 5.7 (KOSARAK) on the largest dataset (×16). As we increase BMS-POS by a factor of 16, the runtime increases by a factor of 195 for Spread, 204 for Join-Clust, and 460 for SymClust. The memory grows linearly for all measured data points and increases by a factor of about 2 when we double the data size. Join-Clust requires 18-25 (BMS-POS) resp. 499-569 (KOSARAK) times more memory than its competitors and runs out of memory on KOSARAK except for the ×1 dataset.

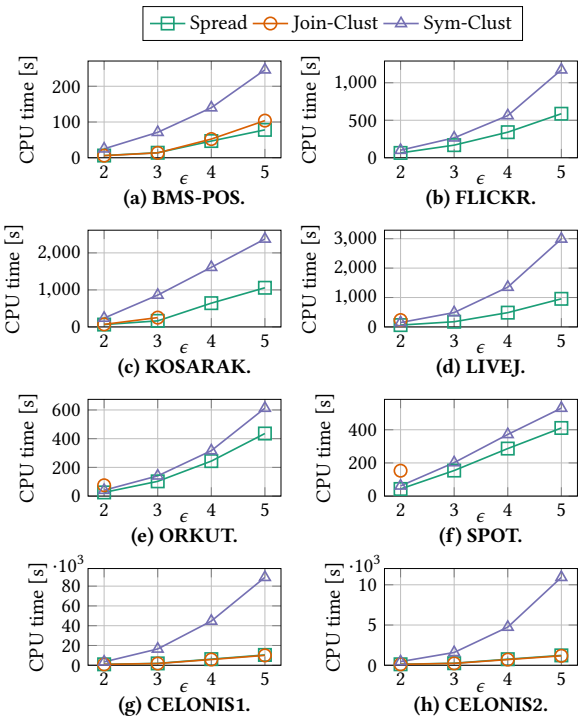

Figure 7: Runtime over ϵ, minPts = 16.

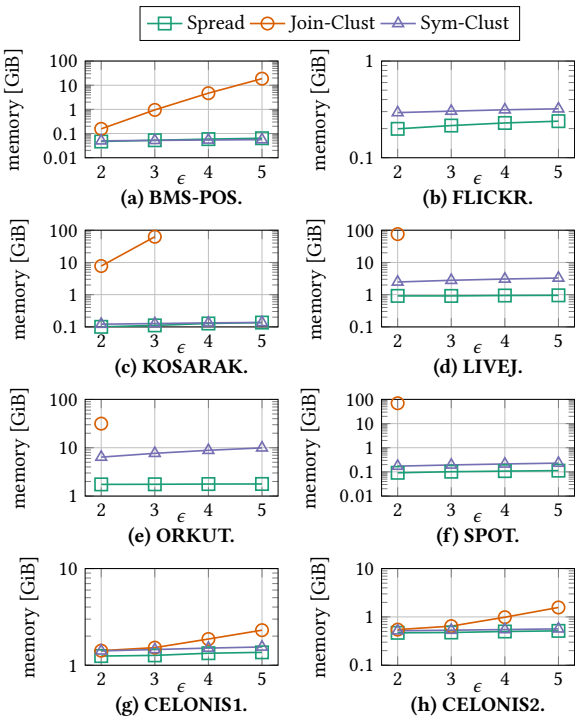

Figure 11: Main memory over ϵ, minPts = 16.

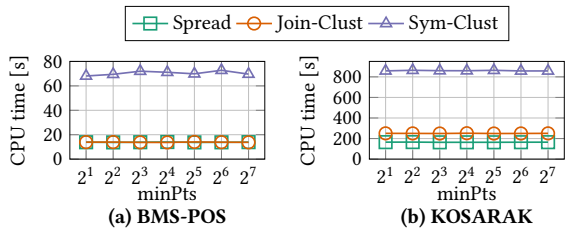

Figure 8: Runtime over minPts, ϵ = 3.

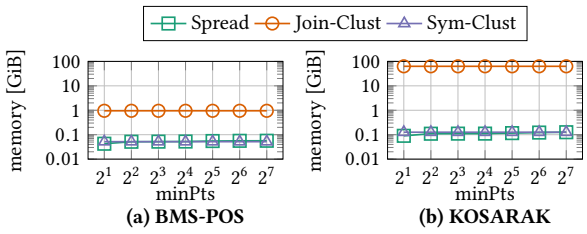

Figure 12: Main memory over minPts, ϵ = 3.

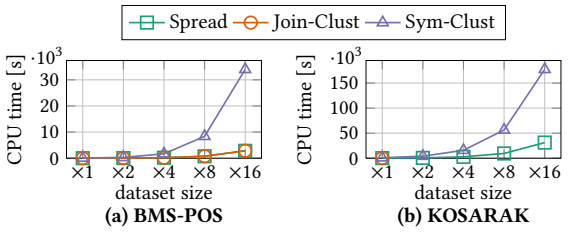

Figure 9: Runtime over data size, ϵ = 3, minPts = 16.

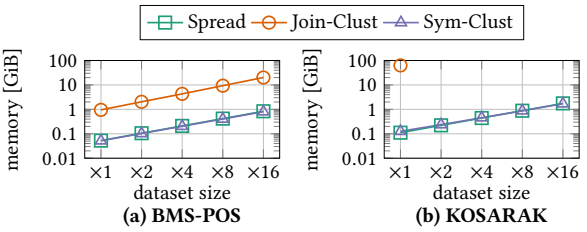

Figure 13: Main memory over data size, ϵ = 3, minPts = 16.

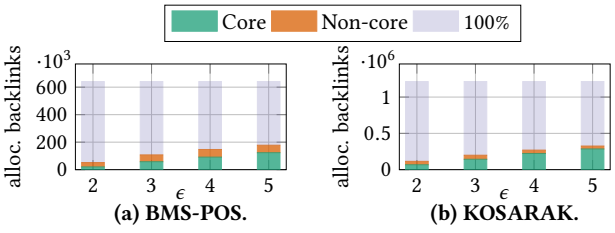

Figure 10: Backlinks peak over ϵ, minPts = 16.

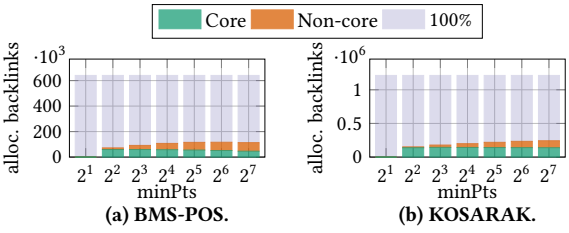

Figure 14: Backlinks peak over minPts, ϵ = 3.

Summarizing, Spread clearly outperforms Sym-Clust in runtime (by a factor of 5-12) and Join-Clust in memory usage (by more than an order of magnitude) as we increase the data size.

6 RELATED WORK

Indexes for Sets. Most set similarity joins operate on an inverted list index that maps signatures to candidate sets. Various signatures have been proposed [2, 8, 40, 45]. Prefixes [8] in conjunction with the length filter [1] have been shown to prune sets effectively. More sophisticated filters include positional and suffix filter [49], the removal filter [35], the position-enhanced length filter [28], and the adaptive prefix filter [44]. Wang et al. [46] leverage the similarity of the sets in an ϵ-neighborhood to reduce the overall number of false positives. Dong et al. [13] propose a size-aware algorithm that runs in $o(n^2) + O(k)$ time for k result pairs. Qin and Xiao [34] propose the pigeonring, a generalization of the pigeonhole principle that yields stronger constraints. Indexing and join techniques for sets have been studied extensively in the single-machine [29] and the distributed context [16].

Most of these approaches focus on self-joins, which order the sets and compute the lookahead neighborhood to avoid symmetric distance computations. In our work, we use the prefix filter, but any of the other asymmetric indexes is applicable.

Efficient Region Queries. Ester et al. [15] propose the first exact DBSCAN algorithm with $O(n \log n)$ runtime for vectors of arbitrary dimension. $O(n \log n)$ runtime holds for a small number of neighbors (compared to n) and an index with $O(\log n)$ lookup time. Henceforth, efficient region query computation has been of great interest and many improvements have been proposed. Brecheisen et al. [7] use minPts-nearest neighbor queries to identify core points and postpone the other distance computations until the distances are required to get a correct DBSCAN clustering. The proposed *Xseedlist* data structure is designed for expensive distance functions and assumes a cheap but selective filter. These assumptions do not hold for sets: The verification (i.e., distance computation) of candidate pairs has shown to be highly efficient [29] (a small number of integer comparisons). Brecheisen et al. must insert the candidates into the Xseedlist data structure, which maintains sorted lists of candidates. Due to the expensive sorting procedure, we do not expect Xseedlist to improve the DBSCAN algorithm for sets. TI-DBSCAN [25] exploits the triangle inequality to reduce the search space of region queries. The solution is not index-based, sorts the points w.r.t. a reference point, and shifts a window of size 2ϵ over the sorted points. The reference point is the point with minimal values in all dimensions. This is equivalent to the empty set, and our processing order in combination with the prefix index for sets subsumes this technique. Patwary et al. [32] introduce PARDICLE, a parallel approximate density-based clustering algorithm for Euclidean space. Its aim is to reduce the neighborhood computation time by sampling high-density regions. Kumar and Reddy [26] propose a new graph-based index structure called Groups. It discovers groups of patterns in two scans over the dataset and applies a standard DBSCAN afterwards. Groups accelerates region queries by pruning noise points effectively. This technique assumes Euclidean distance and does not consider Hamming distance or other set similarity measures. Recently, Jiang et al. [24] proposed SNG-DBSCAN, which prevents the computation of the full ϵ-neighborhood graph via subsampling its edges. This results in $O(sn^2)$-time complexity with s being the sampling rate. Under certain distribution assumptions, SNG-DBSCAN has been shown to preserve the exact ϵ-neighborhood graph for $s \approx (\log n)/n$ with $O(n \log n)$ runtime.

DBSCAN Techniques. Yang et al. [51] propose the distributed DBSCAN-MS clustering algorithm for metric spaces. DBSCAN-MS uses pivots to map the data from metric space to vector space, where it is partitioned in order to be distributed. A local DBSCAN is then executed on each partition. Our solution does not rely on the metric properties of set distances, but uses specialized set indexes. However, our techniques may be leveraged in the context of DBSCAN-MS, where the data points are ordered by one of the dimensions for efficient neighborhood queries.

Patwary et al. [31] propose PDSDBSCAN, a parallel DBSCAN algorithm that uses the disjoint-set data structure to connect data points into clusters. We only insert links between subclusters into the disjoint-sets structure, while PDSDBSCAN inserts a link for each neighbor, rendering the number of required union operations a bottleneck for this approach.

Böhm et al. [3] use a block-nested loop join and buffer the join result to reduce the number of block accesses required to compute ϵ-neighborhoods. CUDA-DClust [4] is a GPU-based solution that splits clusters into chains that are expanded from different starting points in parallel. In order to merge chains into clusters, a quadratic-size bit matrix is used. We maintain only a linear number of links and leverage disjoint-sets to merge clusters. Incremental DBSCAN algorithms [14] deal with updates on an existing clustering. Similar to our approach, these techniques may need to merge clusters when new points are inserted. None of the above solutions supports asymmetric neighborhood indexes.

Numerous parallel and distributed algorithms [9, 11, 18–21, 23, 36, 38, 47, 50] as well as approximations [17, 27, 43, 48] have been proposed. We present an exact, single-core solution for sets.

7 CONCLUSION

In this paper, we have investigated clustering techniques for large collections of sets. Our work was motivated by an application in process mining that models processes as sets to assess their similarity. We have shown that the solutions that are currently available, Sym-Clust and Join-Clust, are not satisfying: Sym-Clust is slow since it cannot use effective asymmetric set indexes, while Join-Clust is infeasible for many settings due to its excessive memory usage. We introduced a novel, density-based clustering algorithm, Spread, that can process data points in any user-defined order and is therefore fit for the use with asymmetric indexes. Spread combines the best of both worlds: It uses the effective asymmetric index of Join-Clust, but like Sym-Clust does not need to materialize the neighborhoods. We introduced so-called backlinks to guarantee a correct DBSCAN clustering and showed the correctness of our approach. To the best of our knowledge, Spread is the first DBSCAN-compliant algorithm that uses an asymmetric index and runs in linear space.

Spread uses the index as a black box and works with any data type. Interesting future work includes evaluating the performance of Spread for vector data, where candidates are generated using a sliding window that is shifted along one dimension. The data points in the window are candidates, i.e., the window simulates an asymmetric index for Spread.

ACKNOWLEDGMENTS

We thank Alexander Miller, Mateusz Pawlik, Thomas Hütter, Manuel Widmoser, Manuel Kocher, Daniel Ulrich Schmitt, Konstantin Thiel, Daniel Grittner, Christian Böhm, and Claudia Plant

for valuable discussions, and Manuel Kocher for typesetting Figures 1 and 3. This work was partially supported by the Austrian Science Fund (FWF): P 29859.

REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*. 918–929.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling Up All Pairs Similarity Search. In *Proc. of the Int. Conf. on World Wide Web (WWW)*. 131–140.

[3] Christian Böhm, Bernhard Braunmüller, Markus Breunig, and Hans-Peter Kriegel. 2000. High Performance Clustering Based on the Similarity Join. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 298–305.

[4] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. 2009. Density-Based Clustering Using Graphics Processors. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 661–670.

[5] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. 2010. Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models. In *Business Process Management Workshops*. 170–181.

[6] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-Textual Similarity Joins. *Proc. of the VLDB Endowment* 6, 1 (Nov. 2012), 1–12.

[7] S. Brecheisen, H. . Kriegel, and M. Pfeifle. 2004. Efficient Density-Based Clustering of Complex Objects. In *Proc. of the IEEE Int. Conf. on Data Mining*. 43–50.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*. 5–5.

[9] I. Cordova and T. Moh. 2015. DBSCAN on Resilient Distributed Datasets. In *Proc. of the Int. Conf. on High Performance Computing Simulation (HPCS)*. 531–540.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.).

[11] B. Dai and I. Lin. 2012. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *Proc. of the IEEE Int. Conf. on Cloud Computing*. 59–66.

[12] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An Efficient Partition Based Method for Exact Set Similarity Joins. *Proc. of the VLDB Endowment* 9, 4 (Dec. 2015), 360–371.

[13] Dong Deng, Yufei Tao, and Guoliang Li. 2018. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 905–920.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. 1998. Incremental Clustering for Mining in a Data Warehousing Environment. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*. 323–333.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. 226–231.

[16] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set Similarity Joins on Mapreduce: An Experimental Survey. *Proc. of the VLDB Endowment* 11, 10 (June 2018), 1110–1122.

[17] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 519–530.

[18] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: Highly Parallel DBSCAN. In *Proc. of the Workshop on Machine Learning in High-Performance Computing Environments*. Article 2, 10 pages.

[19] D. Han, A. Agrawal, W. Liao, and A. Choudhary. 2016. A Novel Scalable DBSCAN Algorithm with Spark. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1393–1402.

[20] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. 2014. MR-DBSCAN: A Scalable MapReduce-Based DBSCAN Algorithm for Heavily Skewed Data. *Frontiers of Computer Science* 8, 1 (2014), 83–99.

[21] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. 2011. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*. 473–480.

[22] B.F.A. Hompes, J.C.A.M. Buijs, W.M.P. van der Aalst, P.M. Dixit, and J. Buurman. 2015. Discovering Deviating Cases and Process Variants Using Trace Clustering. In *Benelux Conf. on Artificial Intelligence*.

[23] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. 2004. Scalable Density-Based Distributed Clustering. In *Knowledge Discovery in Databases (PKDD)*. 231–244.

[24] Heinrich Jiang, Jennifer Jang, and Jakub Łącki. 2020. Faster DBSCAN via subsampled similarity queries. *CoRR* (2020).

[25] Marzena Kryszkiewicz and Piotr Lasek. 2010. TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality. In *Rough Sets and Current Trends in Computing (RSCTC)*. 60–69.

[26] K. Mahesh Kumar and A. Rama Mohan Reddy. 2016. A fast DBSCAN clustering algorithm by accelerating neighbor searching using Groups method. *Pattern Recognition* 58 (2016), 39 – 48.

[27] Yinghua Lv, Tinghuai Ma, Meili Tang, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. 2016. An efficient and scalable density-based clustering algorithm for datasets with complex structures. *Neurocomputing* 171 (2016), 9 – 22.

[28] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Proc. of the Workshop Grundlagen von Datenbanken (CEUR Workshop Proceedings)*, Vol. 1313. 89–94.

[29] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *Proc. of the VLDB Endowment* 9, 9 (2016), 636–647.

[30] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proc. of the ACM Int. Conf. on Internet Measurement (SIGCOMM)*. 29–42.

[31] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[32] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey. 2014. Pardicle: Parallel Approximate Density-Based Clustering. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. 560–571.

[33] Martin Pichl, Eva Zangerle, and Günther Specht. 2014. Combining Spotify and Twitter Data for Generating a Recent and Public Dataset for Music Recommendation. In *Proc. of the Workshop Grundlagen von Datenbanken (CEUR Workshop Proceedings)*, Vol. 1313. 35–40.

[34] Jianbin Qin and Chuan Xiao. 2018. Pigeonring: A Principle for Faster Thresholded Similarity Search. *Proc. of the VLDB Endowment* 12, 1 (2018), 28–42.

[35] Leonardo Andrade Ribeiro and Theo Härder. 2011. Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems* 36, 1 (2011), 62–78.

[36] A. Sarma, P. Goyal, S. Kumari, A. Wani, J. S. Challa, S. Islam, and N. Goyal. 2019. μDBSCAN: An Exact Scalable DBSCAN Algorithm for Big Data Exploiting Spatial Locality. In *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER)*. 1–11.

[37] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems* 42, 3 (2017), 21.

[38] Hwanjun Song and Jae-Gil Lee. 2018. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 1173–1187.

[39] Minseok Song, Christian W. Günther, and Wil M. P. van der Aalst. 2009. Trace Clustering in Process Mining. In *Business Process Management Workshops*. 109–120.

[40] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2019. Balance-aware Distributed String Similarity-based Query Processing System. *Proc. of the VLDB Endowment* 12, 9 (2019), 961–974.

[41] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM* 22, 2 (1975), 215–225.

[42] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient Parallel Set-Similarity Joins Using MapReduce. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 495–506.

[43] P. Viswanath and V. Suresh Babu. 2009. Rough-DBSCAN: A fast hybrid density based clustering method for large data sets. *Pattern Recognition Letters* 30, 16 (2009), 1477 – 1488.

[44] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can We Beat the Prefix Filtering? An Adaptive Framework for Similarity Join and Search. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 85–96.

[45] Pei Wang, Chuan Xiao, Jianbin Qin, Wei Wang, Xiaoyang Zhang, and Yoshiharu Ishikawa. 2016. Local Similarity Search for Unstructured Text. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 1991–2005.

[46] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2017. Leveraging Set Relations in Exact Set Similarity Join. *Proc. of the VLDB Endowment* 10, 9 (May 2017), 925–936.

[47] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2555–2571.

[48] Y. Wu, J. Guo, and X. Zhang. 2007. A Linear DBSCAN Algorithm Based on LSH. In *Proc. of the Int. Conf. on Machine Learning and Cybernetics*, Vol. 5. 2608–2614.

[49] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient Similarity Joins for Near-Duplicate Detection. *ACM Transactions on Database Systems* 36, 3 (2011), 41.

[50] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. 1999. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery* 3, 3 (1999), 263–290.

[51] K. Yang, Y. Gao, R. Ma, L. Chen, S. Wu, and G. Chen. 2019. DBSCAN-MS: Distributed Density-Based Clustering in Metric Spaces. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*. 1346–1357.

[52] Zijian Zheng, Ron Kohavi, and Llew Mason. 2001. Real World Performance of Association Rule Algorithms. In *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. 401–406.

CONTENTS

ABSTRACT 1

1. INTRODUCTION 1

2. BACKGROUND & PROBLEM DEFINITION 2

2.1. SET SIMILARITY AND Î• -NEIGHBORHOOD 2

2.2. INDEXING TECHNIQUES FOR SETS 2

2.3. DENSITY-BASED CLUSTERING 3

2.4. THE DBSCAN ALGORITHM 3

2.5. PROBLEM STATEMENT 3

3. BASELINE APPROACHES 4

3.1. SYM-CLUST: DBSCAN WITH INVERTED INDEX 4

3.2. JOIN-CLUST: MATERIALIZED NEIGHBORHOODS 5

4. THE SPREAD ALGORITHM 5

4.1. KEY CHALLENGES 5

4.2. DATA STRUCTURES 6

4.3. THE ALGORITHM 6

4.4. CORRECTNESS 7

4.5. COMPLEXITY ANALYSIS 8

4.6. MULTI-CORE EXTENSION 8

5. EXPERIMENTAL EVALUATION 8

5.1. INDEX & CLUSTER STATISTICS 9

5.2. RUNTIME EFFICIENCY 9

5.3. MEMORY EFFICIENCY 9

5.4. SCALABILITY 9

6. RELATED WORK 11

7. CONCLUSION 11

REFERENCES 12

Scaling Density-Based Clustering to Large Collections of Sets

Daniel Kocher
University of Salzburg
Salzburg, Austria
dkocher@cs.sbg.ac.at

Nikolaus Augsten
University of Salzburg
Salzburg, Austria
nikolaus.augsten@sbg.ac.at

Willi Mann
Celonis SE
Munich, Germany
w.mann@celonis.com

ABSTRACT

We study techniques for clustering large collections of sets into DBSCAN clusters. Sets are often used as a representation of complex objects to assess their similarity. The similarity of two objects is then computed based on the overlap of their set representations, for example, using Hamming distance. Clustering large collections of sets is challenging. A baseline that executes the standard DBSCAN algorithm suffers from poor performance due to the unfavorable neighborhood-by-neighborhood order in which the sets are retrieved. The DBSCAN order requires the use of a symmetric index, which is less effective than its asymmetric counterpart. Precomputing and materializing the neighborhoods to gain control over the retrieval order often turns out to be infeasible due to excessive memory requirements.

We propose a new, density-based clustering algorithm that processes data points in any user-defined order and does not need to materialize neighborhoods. Instead, so-called backlinks are introduced that are sufficient to achieve a correct clustering. Backlinks require only linear space while there can be a quadratic number of neighbors. To the best of our knowledge, this is the first DBSCAN-compliant algorithm that can leverage asymmetric indexes in linear space. Our empirical evaluation suggests that our algorithm combines the best of two worlds: it achieves the runtime performance of materialization-based approaches while retaining the memory efficiency of non-materializing techniques.

1 INTRODUCTION

We consider the problem of partitioning large collections of sets into DBSCAN [15] clusters. Our work is motivated by a process mining use case at Celonis SE that models processes as sets. A *process* is a sequence of timestamped activities. Large companies store hundreds of millions of activities in millions of processes. In order to analyze the processes, they should be clustered into groups of similar activity sequences that can be further explored [5, 22, 39]. To this end, a process is represented by the set of all its neighboring activity pairs, e.g., the process with the activity sequence (S, O, P, H, R, F, E) (Start, Order, Pay, sHip, Return good, reFund, End) is represented by the set $\{(S, O), (O, P), (P, H), (H, R), (R, F), (F, E)\}$. The similarity of two processes is then assessed by the Hamming distance[1] of their set representations.

Sets are used in many other applications [29] to represent objects for the purpose of clustering, e.g., sales may be represented by sets of product categories, photos by sets of tags and title words, user interactions on a website by sets of visited links, users of a social network by their group memberships, or users of a music streaming platform by sets of tracks they listen to.

[1]Hamming distance $H(r, s) = |r \cup s| - |r \cap s|$ for two sets r and s.

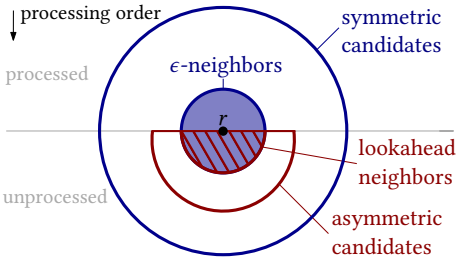

Figure 1: Symmetric candidates with ϵ-neighbors (blue); asymmetric candidates with lookahead neighbors (red).

The popular DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm [15] identifies clusters of arbitrary shape without requiring the number of clusters as input. Intuitively, DBSCAN finds dense regions that are separated by regions of lower density. The density of a region (given a distance function between pairs of data points) is defined by two parameters: the number of neighbors, *minPts*, and the radius, ϵ, of the neighborhood. A data point is called *core point* (i.e., it is at the core of a dense region) if it has at least minPts neighbors (including itself) within radius ϵ; a non-core point in the ϵ-neighborhood of a core point is a *border point* (i.e., it is at the border of a dense region); all other points are *noise* [37].

The runtime of the DBSCAN algorithm heavily depends on the efficiency of the neighborhood computation. In our experiments, the neighborhood computation accounts for up to 99% of the overall runtime for some datasets. Therefore, in order to efficiently cluster large collections of sets, effective indexing techniques for sets are required.

Similarity indexes for sets have been proposed in the context of ϵ-neighborhood joins, which are executed in an index nested loop fashion. A prominent representative is the *prefix index* [1, 8], which is linear in size and has been shown to be highly effective [29]. The *symmetric* prefix index returns the complete ϵ-neighborhood for a given query point r. The *asymmetric* prefix index assumes a processing order on the sets in R and retrieves only the *lookahead neighbors*: the ϵ-neighbors that follow r in processing order. A typical processing order for sets is based on the set sizes (ties broken arbitrarily). The *asymmetric* prefix index further leverages the length information to avoid many of the candidates that the symmetric index must inspect (among the unprocessed sets). Figure 1 illustrates the ϵ-neighborhood, the lookahead neighbors, and the candidate regions of symmetric and asymmetric index, respectively. The region above the gray line represents the sets that have been processed before r, the region below the gray line are unprocessed sets. The circles and semicircles show subset relationships.

Clearly, the asymmetric prefix index is preferable in terms of effectiveness over its symmetric counterpart. Unfortunately, there is an inherent mismatch between the asymmetric index and the DBSCAN algorithm. DBSCAN suffers from the following issues when executed with the asymmetric index: (1) *Core status*

problem: The lookahead neighbors of r are not sufficient to update the core status of r. (2) *Border vs. Noise problem:* To distinguish border points from noise, a border point must be visible from a core point, which is not guaranteed by the asymmetric lookahead neighborhood. (3) *Disconnected clusters:* To guarantee connected clusters, DBSCAN imposes a (partial) processing order on the neighborhood computations: all core points of the current cluster must be expanded (i.e., their neighborhood must be computed) before any point belonging to a different cluster is processed.

A well-known clustering approach [3] is based on a self-join that precomputes and materializes all neighborhoods. The precomputed neighborhoods are then used while executing DBSCAN. This approach can leverage the asymmetric index and is efficient in runtime. Unfortunately, this join-based technique requires quadratic memory in the worst case and suffers from a large memory footprint in practice. For example, for our social media dataset (LIVEJ) that stores the interests of 3.1M users, this approach requires almost 100GB of memory.

Summarizing, applications that must cluster large collections of sets have two options, which we call Sym-Clust and Join-Clust. (1) Sym-Clust: Retrieve the full ϵ-neighborhoods in the processing order imposed by DBSCAN using the symmetric index. (2) Join-Clust: Compute and materialize neighborhoods in a join using the effective asymmetric index. None of the options is satisfying: Sym-Clust runs almost up to an order of magnitude slower than Join-Clust, while Join-Clust is infeasible for many datasets and parameter settings due to its excessive memory usage.

We propose a new clustering algorithm, *Spread*, that computes correct DBSCAN clusters using the asymmetric prefix index. Spread runs in linear space and does not need to materialize the (quadratic-size) neighborhoods. Spread avoids symmetric neighbor computations, therefore reducing the number of neighbors retrieved by Sym-Clust. So-called *backlinks* are introduced to achieve a correct clustering. Backlinks are dynamically added and removed as required and occupy only a small fraction of the memory that is used by materialized neighborhoods. Spread maintains a graph of subclusters in a disjoint-set data structure and guarantees that connected components in the resulting graph represent correct DBSCAN clusters.

In general, Spread can process data points in any *user-defined order* given an index that retrieves the lookahead neighbors, i.e., all data points that follow the query point in the user-defined processing order. In our usage scenario – set clustering – the processing order is defined by the set sizes (ties broken arbitrarily) and the asymmetric prefix index retrieves lookahead neighbors.

Summarizing, our contributions are the following:

- We propose *Spread*, a novel algorithm for partitioning large collections of sets into DBSCAN clusters. To the best of our knowledge, this is the first linear space DBSCAN-compliant algorithm that leverages the *asymmetric* prefix index for sets.
- We introduce the new concept of *backlinks* that keep sufficient information to build correct clusters independently of the processing order that the user imposes on Spread. We prove invariants for backlinks and the correctness of our approach.
- Our extensive empirical evaluation on 13 real-world datasets suggests that Spread is as fast as Join-Clust (that materializes all neighborhoods) while being competitive in memory usage with Sym-Clust (that computes all neighborhoods on the fly).

The remainder of this paper is organized as follows. In Section 2, we cover the background on ϵ-neighborhood and set similarity, indexing techniques for sets, and density-based clustering, and we define the *density-based set clustering problem*. Section 3 presents the two baseline approaches for density-based set clustering, Join-Clust and Sym-Clust. In Section 4, we present Spread, our time- and space-efficient solution for density-based set clustering. We evaluate our solution against the baseline algorithms and discuss the results in Section 5. Related work is summarized in Section 6. Finally, Section 7 concludes this paper.

2 BACKGROUND & PROBLEM DEFINITION

We revisit set similarity indexes and density-based clustering, and define our problem. To simplify the presentation, we focus on prefix indexes for the Hamming distance. Our results, however, extend to other distance and similarity measures (e.g., Jaccard or Cosine) and the respective indexes [12, 29, 40]. The required adaptations of the index that have been studied in the context of set similarity joins [29, 49] (e.g., prefix length, size lower bound, equivalent overlap) also apply to our scenario.

2.1 Set Similarity and ϵ-Neighborhood

R is a collection of n unique sets, each set $r \in R$ consists of unique tokens $t_1, \ldots, t_m$, $|r| = m$. The *processing order*, $\succ$, is a total order defined over R. The similarity between two sets r and s is assessed by the Hamming distance, $H(r, s) = |r \cup s| - |r \cap s|$, which counts the number of tokens that exist only in one of the sets, e.g., $H(r_1, r_2) = 4$ and $H(r_2, r_3) = 3$ for the sets in Figure 2.

The *ϵ-neighborhood* of set r includes r and all sets within distance ϵ from r, $N_\epsilon(r) = \{s \in R \mid H(r, s) \leq \epsilon\}$. A *region query* on r computes $N_\epsilon(r)$. A set r splits its ϵ-neighborhood into two disjoint parts based on the processing order: the *lookahead neighbors* that follow r in processing order, $N_\epsilon^{\succ}(r) = \{s \in N_\epsilon(r) \mid s \succ r\}$ and the *preceding neighbors*, $N_\epsilon^{\prec}(r) = \{s \in N_\epsilon(r) \mid s \prec r\}$.

2.2 Indexing Techniques for Sets

Prefix Filter and Inverted Index. A naive approach computes a region query $N_\epsilon(r)$ by verifying the predicate $H(r, s) \leq \epsilon$ for all sets $s \in R$. An effective indexing technique, which was originally developed for set similarity joins [2, 29], is based on the so-called prefix filter. The prefix, π_r, of a set r consists of the first π tokens of r according to some total token order (which must be the same for all sets). The prefix length depends on the distance function and is $\pi = \epsilon + 1$ for the Hamming distance. Figure 2 shows the prefix of three sets for distance threshold $\epsilon = 3$ and a numerical token order. The prefix filter works best if the tokens in the prefix are infrequent, thus the tokens are typically ordered by ascending global token frequency.

A set $s \in R$ can be in the ϵ-neighborhood $N_\epsilon(r)$ only if the prefixes of r and s share at least one token, i.e., $H(r, s) \leq \epsilon \Rightarrow \pi_r \cap \pi_s \neq \emptyset$ (assuming $|r| + |s| > \epsilon$; otherwise r and s are always similar). Therefore, if two sets do not share a token in the prefix, the pair can be safely pruned. If two sets r and s share a prefix token, (r, s) is a *candidate pair* and must undergo *verification*, i.e., the predicate $H(r, s) \leq \epsilon$ must be evaluated. Candidates that fail verification are *false positives*. Mann et al. [29] discuss efficient prefix-based verification.

Symmetric Prefix Index. An inverted index on the prefix tokens is used to retrieve candidate pairs efficiently. The inverted index maps prefix tokens to sets that contain that token in the prefix. A lookup of set r retrieves all lists of the prefix tokens of r. The

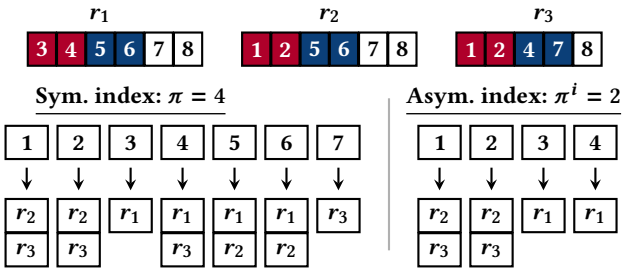

Figure 2: Symmetric and asymmetric prefix index, $\epsilon = 3$.

union of these lists (except r itself) are the candidates of r. The index is *symmetric* and returns the ϵ-neighborhood of r.

For example, the candidates for r_2 returned by the symmetric index, $\pi = \epsilon + 1 = 4$, in Figure 2 are $\{r_1, r_3\}$ (resulting from the union of $[r_2, r_3]$ for token 1, $[r_2, r_3]$ for token 2, $[r_1, r_2]$ for token 5, and $[r_1, r_2]$ for token 6). Candidate r_1 is a false positive since $H(r_1, r_2) > \epsilon$; r_3 is a true positive due to $H(r_2, r_3) \leq \epsilon$.

Asymmetric Prefix Index. We construct an *asymmetric* prefix index that returns only the lookahead neighbors, $N_\epsilon^{\succ}(r)$. To this end, we define a length-based processing order on R (longest to shortest): r precedes s if $|r| > |s|$, i.e., $|r| > |s| \Rightarrow s \succ r$; ties ($|r| = |s|$) are broken by the lexicographic order of the sorted sets.

Since we are interested only in sets $s \in N_\epsilon(r)$ that are no larger than r, $|s| \leq |r|$, we need to index only a subset of the prefix tokens: the tokens in the so-called *indexing prefix* [49]. The so-called *probing prefix* of the lookup set, r, remains of length $\pi = \epsilon + 1$. For the Hamming distance, the indexing prefix is of length $\pi^i = \left\lfloor \frac{\epsilon}{2} + 1 \right\rfloor$. For $\epsilon > 0$, the probing prefix is always longer than the indexing prefix, e.g., $\pi = 4$ and $\pi^i = 2$ for $\epsilon = 3$. A shorter prefix results in fewer candidates and renders the asymmetric index more effective than its symmetric counterpart.

In the case of r_2, the asymmetric index, $\pi^i = 2$, in Figure 2 returns only a true positive candidate, r_3. The false positive candidate, r_1, which is returned by the symmetric index, is avoided.

2.3 Density-Based Clustering

We formally define DBSCAN clusters and the related concepts. A set r represents a point to be clustered. The *density* of r is the number of ϵ-neighbors $|N_\epsilon(r)|$ (cf. Section 2.1).

Core, Border, Noise. A set r is a *core point* iff the ϵ-neighborhood of r contains at least minPts sets: r is core $\Leftrightarrow |N_\epsilon(r)| \geq \text{minPts}$. A set s is a *border point* iff it is in the ϵ-neighborhood of a core point r and s is not core: s is border $\Leftrightarrow s \in N_\epsilon(r) \wedge |N_\epsilon(r)| \geq \text{minPts} \wedge |N_\epsilon(s)| < \text{minPts}$. All remaining sets in R are *noise*. We denote the set of core and border points with $\mathcal{C}$ and $\mathcal{B}$, respectively. The set of noise points is $\mathcal{N} = R \setminus (\mathcal{C} \cup \mathcal{B})$.

Density-Reachability. Let $r, s \in R$ and r is core: s is *directly density-reachable* from r iff s is in the ϵ-neighborhood of r: $r \blacktriangleright s \Leftrightarrow s \in N_\epsilon(r)$. If there is a sequence of sets $r_1, r_2, \ldots, r_k$ with $r_1 = r$ and $r_k = s$, $r_i \blacktriangleright r_{i+1}$ for $1 \leq i < k$, s is *density-reachable* from r, denoted $r \blacktriangleright \ldots \blacktriangleright s$. Two sets r, s are *density-connected* if there is a set x s.t. both r and s are density-reachable from x.

A *density-based cluster* is a subset $C_i \subseteq R$ that satisfies two criteria [38]:

(1) *Maximality* $\mathbb{M}$*:* For any two sets $r, s \in R$, $r \in C_i$. If s is density-reachable from r, then $s \in C_i$. Formally,

$$\forall r, s \in R : r \in C_i \wedge r \blacktriangleright \ldots \blacktriangleright s \implies s \in C_i$$

Table 1: Notation overview.

Notation	Description
R	a collection of sets
r, s, x	sets of R
$\|r\|$	cardinality of set r
$r \prec s, r \succ s$	r precedes/succeeds s (in R)
$H(r, s)$	the Hamming distance of two sets r, s
π, π^i	probing/indexing prefix
ϵ	distance threshold
minPts	minimum density s.t. a set r is core
$N_\epsilon(r)$	full ϵ-neighborhood of r
$N_\epsilon^{\prec}(r), N_\epsilon^{\succ}(r)$	preceding/lookahead neighbors of r
$r \blacktriangleright s$	s is directly density-reachable from r
$r \blacktriangleright \ldots \blacktriangleright s$	s is density-reachable from r
$\mathcal{C}, \mathcal{B}, \mathcal{N}$	the set of core, border, and noise sets
C_i	a density-based cluster with id i

(2) *Connectivity* $\mathbb{C}$*:* For any two sets r, s in C_i, there is a set x that density-connects r and s. Formally,

$$\forall r, s \in R : r, s \in C_i \implies \exists x \in C_i : r \blacktriangleleft \ldots \blacktriangleleft x \blacktriangleright \ldots \blacktriangleright s$$

DBSCAN Clustering. A border point may be part of multiple density-based clusters such that the clusters overlap. We define the *DBSCAN clustering* that partitions the data into non-overlapping clusters. The standard DBSCAN algorithm [15] produces a DBSCAN clustering.

Definition 2.1. Let $R^* = R \setminus \mathcal{N}$ and $C_1, C_2, \ldots, C_k$ be density-based clusters such that $\bigcup_{i=1}^{k} C_i = R^*$. A DBSCAN clustering is a partitioning $\Gamma = \{C'_1, C'_2, \ldots, C'_k\}$, $C'_i \subseteq C_i$, such that $\bigcup_{i=1}^{k} C'_i = R^*$, $C'_i \cap C'_j = \emptyset$ for $i \neq j$.

A *subclustering* of a cluster C_i, $\psi_i = \{c_1, c_2, \ldots, c_l\}$, is a partitioning of C_i into $1 \leq l \leq |C_i|$ non-empty, disjoint subclusters, $c_j \subseteq C_i$, such that $\bigcup_{j=1}^{l} c_j = C_i$, $c_j \cap c_k = \emptyset$ for $j \neq k$.

A *subcluster graph* of R^* is an undirected graph in which nodes are subclusters and an edge between two nodes can only exist if the respective nodes are in the same DBSCAN cluster.

2.4 The DBSCAN Algorithm

The standard DBSCAN algorithm [15] forms clusters by repeatedly picking a seed point from the set of unvisited data points (initially all points are unvisited). If the seed is a core point, it forms a new cluster with all points that are density-reachable from the seed and are not yet assigned to a cluster. The set of density-reachable points is computed by recursively adding the ϵ-neighbors of all core points to the current cluster. The algorithm terminates when all points have been visited. Points that cannot be assigned to a cluster are noise.

2.5 Problem Statement

Definition 2.2 (Density-Based Set Clustering). Given a collection of sets R, a distance threshold ϵ, and the neighborhood density minPts, the goal is to find a DBSCAN clustering $\Gamma = \{C_1, C_2, \ldots, C_k\}$ of R.

For sets, asymmetric indexes with a lookahead neighbor function $N_\epsilon^{\succ}(r)$ promise the best performance (cf. Section 2.2). Given an ordering $\succ$ on R, we strive for a time- and space-efficient algorithm that solves the density-based set clustering problem with an asymmetric index.

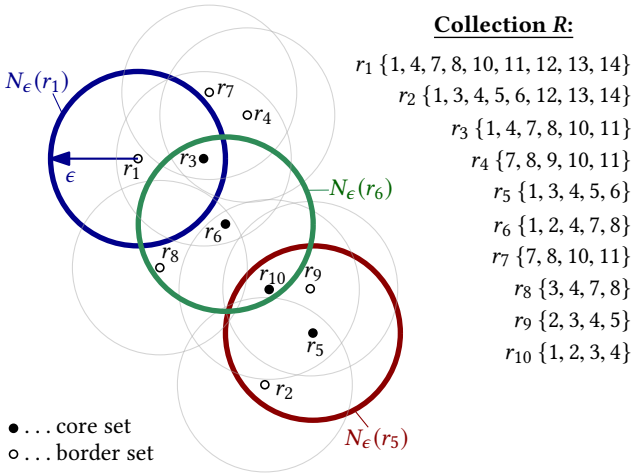

Figure 3: Running example, $\epsilon = 3$, minPts = 4.

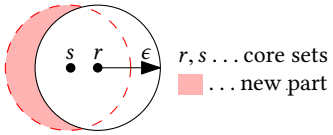

Figure 4: Redundant neighborhood queries.

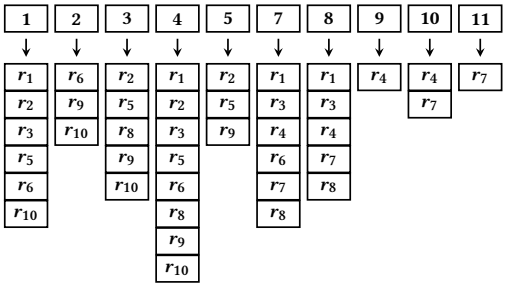

Figure 5: Symmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi = 4$.

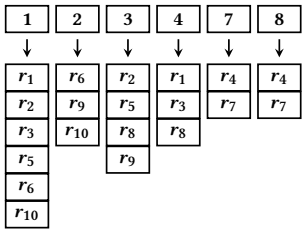

Figure 6: Asymmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi^i = 2$.

Running Example. Figure 3 shows an example collection R of ten sets, r_1–r_{10}, and their neighborhoods for Hamming distance $\epsilon = 3$. Sets r_3, r_5, r_6, and r_{10}, are core sets; all sets r_1–r_{10} form a single cluster.

3 BASELINE APPROACHES

This section presents two baseline solutions for the density-based set clustering problem. (1) *Sym-Clust* is memory-efficient and follows the standard DBSCAN approach with the symmetric prefix index to answer neighborhood queries on the fly. (2) *Join-Clust* is speed-optimized and materializes all ϵ-neighborhoods using a state-of-the-art set similarity join algorithm [2] (which leverages the asymmetric prefix index) before the standard DBSCAN algorithm is executed.

Both baselines leverage state-of-the-art set indexes. We are not aware of other previous solutions that can outperform Sym-Clust or Join-Clust for the density-based set clustering problem. Note that using the standard DBSCAN [15] (rather than some advanced techniques presented in later works, cf. Section 6) is not a limiting factor: Most of the overall execution time is spent computing the neighborhoods, and prefix-based indexes are highly efficient in combination with efficient verification [29].

3.1 Sym-Clust: DBSCAN with Inverted Index

When the standard DBSCAN algorithm (cf. Section 2.4) picks a seed point that is core, it forms a cluster with all points that are density-reachable from the seed. The density-reachable points are computed by pushing all core neighbors of the seed onto a stack. Then, each point on the stack is processed in the same manner (i.e., all its core neighbors are pushed onto the stack) until the stack is empty. All neighbors of core points retrieved in this process belong to the cluster.

The neighborhood queries will overlap to some extent. Assume r is processed before s, $s \in N_\epsilon(r)$, then $|N_\epsilon(s) \cap N_\epsilon(r)| \geq 2$ (at least r and s are in both neighborhoods). Since r assigns all its neighbors to the current cluster, only the non-overlapping neighbors of s, $N_\epsilon(s) \setminus N_\epsilon(r)$, will further increase the cluster.

Figure 4 illustrates this observation for the neighborhoods of two example points r (black circle) and s (red circle): only the new, non-overlapping area of $N_\epsilon(s)$ (shaded in red) is relevant for expanding the cluster.

The standard DBSCAN algorithm requires the use of a symmetric index since it assumes to see all neighbors of a point s when s is processed. The asymmetric index is not compatible with the standard DBSCAN algorithm: We cannot impose an order on the points such that all non-overlapping neighbors of s follow s in the processing order. Further, the size of the neighborhood of s, $|N_\epsilon(s)|$, is required to decide its core status.

Figure 5 shows the symmetric prefix index for our running example, $\epsilon = 3$, $\pi = \epsilon + 1 = 4$. We probe $r_8 = \{3, 4, 7, 8\}$. The prefix of r_8 consists of all tokens in r_8 (due to $|r_8| = \pi$). The union of the respective index lists yields the candidates $\{r_2, r_5, r_9, r_{10}, r_1, r_3, r_6, r_4, r_7\}$. Note that the candidates include both sets that are smaller and sets that are larger than r_8.

The so-called *length filter* [2], an optimization of the symmetric prefix index that also applies to its asymmetric counterpart, prunes candidates r_2 and r_1. Due to their length difference to r_8, they cannot be in the ϵ-neighborhood of r_8. By ordering the lists in processing order (i.e., longer sets precede shorter sets, as illustrated in Figure 5), the length filter can prune the head (sets that are too long) and the tail (sets that are too short) of a list without inspecting all elements in head and tail, respectively.

All candidates that are not pruned by the length filter must undergo verification. Only r_6 passes verification, therefore $N_\epsilon(r_8) = \{r_8, r_6\}$, and r_8 is classified as non-core (minPts = 4).

Complexity Analysis. We probe each set $r \in R$ against the index once. With cost C for an index lookup and $n = |R|$, the runtime is $O(n \cdot C)$; $C = O(n)$ since r may have $O(n)$ neighbors, thus the overall runtime of Sym-Clust is $O(n^2)$. The symmetric index is of linear size leading to space complexity $O(n)$.

Algorithm 1: Materialize-Neighborhoods(R, ϵ)

Input: Collection of sets R, distance threshold ϵ
Result: Materialized neighborhoods of R w.r.t. ϵ

$I \leftarrow$ Create-Index (R, ϵ);
$pairs \leftarrow \emptyset$ // Result set of similar pairs
foreach $r \in R$ *in processing order* **do**
  $M \leftarrow$ Probe (r, I, ϵ) // candidates with prefix overlaps
  foreach $(s, po) \in M$ **do** // po . . . prefix overlap
    if Verify-Pair (r, s, ϵ, po) **then**
      $pairs \leftarrow pairs \cup \{(r, s)\}$
$neighborhoods \leftarrow$ new associative array of size $|R|$;
foreach $(r, s) \in pairs$ **do**
  $neighborhoods[r] \leftarrow neighborhoods[r] \cup \{s\}$;
  $neighborhoods[s] \leftarrow neighborhoods[s] \cup \{r\}$;
return *neighborhoods*

Algorithm 2: Create-Index (R, ϵ)

Input: Collection of sets R, distance threshold ϵ
Result: Inverted index of R w.r.t. ϵ

$I \leftarrow \emptyset$ // inv. index of set prefixes, $I_{r[p]}$ . . . list of token $r[p]$
foreach $r \in R$ **do**
  $\pi \leftarrow \left\lfloor \frac{\epsilon+2}{2} \right\rfloor$ // indexing prefix length of r
  for $p \leftarrow 1$ **to** π **do** $I_{r[p]} \leftarrow I_{r[p]} \cup \{r\}$;
return I

Algorithm 3: Probe (r, I, ϵ)

Input: Probing set r, inv. index I, distance threshold ϵ
Result: Set of candidates for r w.r.t. ϵ

// M maps a candidate s to its prefix overlap with r
$M \leftarrow$ new associative array // candidates
$\pi \leftarrow \epsilon + 1$ // probing prefix length of r
$lb_r \leftarrow |r| - \epsilon$ // size lower bound wrt. r
for $p \leftarrow 1$ **to** π **do**
  foreach $s \in I_{r[p]}$ *in proc. order* **do** // list of token $r[p]$
    if $|s| < lb_r$ **then break** ;
    else // add candidate
      if $s \notin M$ **then** $M[s] \leftarrow 0$; // init.
      $M[s] \leftarrow M[s] + 1$ // incr. overlap of (r, s)
return M

Algorithm 4: Verify-Pair (r, s, ϵ, po)

Input: Probing set r, candidate set s, distance threshold ϵ, prefix overlap po
Result: True iff r and s are similar w.r.t. ϵ, false otherwise

// cf. Mann et al. [29] for prefixes, equiv. overlaps, and Verify proc.
$\pi_r, \pi_s \leftarrow$ probing resp. indexing prefix length of r resp. s;
$w_r, w_s \leftarrow \pi_r$- resp. π_s-th token in r resp. s;
$t \leftarrow$ equivalent overlap for r, s, and ϵ;
if $w_r < w_s$ **then**
  return Verify $(r, s, t, po, \pi_r + 1, po + 1)$
return Verify $(r, s, t, po, po + 1, \pi_s + 1)$

3.2 Join-Clust: Materialized Neighborhoods

Join-Clust executes a set similarity self-join on R and materializes the ϵ-neighborhoods in main memory. The self-join traverses all sets of $r \in R$ in processing order and computes their lookahead neighbors, $N_\epsilon^\succ(r)$. The lookahead neighbors of r are appended to the list of r's neighbors, and r is appended to the neighborhood lists of all $s \in N_\epsilon^\succ(r)$. After processing all sets, the neighborhood list of each set $r \in R$ is complete and stores $N_\epsilon(r)$.

Next, standard DBSCAN (cf. Section 2.4) is executed to form clusters using the materialized neighborhoods. Algorithms 1–4 implement the similarity join with neighborhood materialization, index creation, probing, and efficient verification [29].

Mann et al. [29] found that the prefix-based index in combination with the length filter can be considered state of the art given an efficient verification procedure (which we use).

Figure 6 shows the asymmetric prefix index for our running example, $\epsilon = 3$, $\pi^i = 2$. We probe $r_8 = \{3, 4, 7, 8\}$ and look up the lists of the tokens 3, 4, 7, and 8 (the length of the probing prefix is $\pi = 4$). Since we are only interested in the lookahead neighbors, i.e., all neighbors that follow r_8 in processing order, we need to inspect the lists only starting from the point where r_8 or a set $r_i \succ r_8$ appears. The length filter does not prune any candidate in this example, and the candidate set is $\{r_9\}$. Since $H(r_8, r_9) > \epsilon$, the lookahead neighborhood of r_8 is empty, $N_\epsilon^\succ(r_8) = \emptyset$.

In the context of the self-join, r_8 will be retrieved as a lookahead neighbor of r_6, which is processed before r_8. Therefore, the neighborhood list of r_6 will store r_8 and vice versa.

Join-Clust produces fewer candidates than Sym-Clust and is therefore faster. However, the efficiency of Join-Clust comes at the cost of a larger memory footprint since all neighborhoods must be materialized.

Complexity Analysis. A neighborhood query is a constant-time lookup and a traversal of $O(|N_\epsilon(r)|)$ neighbors. In the worst case, the join reports $O(n^2)$ pairs. Consequently, materializing the neighborhoods takes $O(n^2)$ time and space for $n = |R|$. The asymmetric prefix index requires only $O(n)$ space and does not dominate memory usage.

4 THE SPREAD ALGORITHM

We present *Spread*, a novel time- and space-efficient solution for the density-based clustering problem. Spread leverages the effective asymmetric index and clusters all sets by traversing the sets in processing order. We identify key challenges that must be solved, discuss the algorithm, prove its correctness, analyze time and space complexity, and sketch a multi-core extension.

4.1 Key Challenges

Since Spread uses an asymmetric neighborhood index, a processing order, $\succ$, must be imposed on the data points, and an index lookup of query point r retrieves only the lookahead neighbors, $N_\epsilon^\succ(r)$. To achieve a correct clustering without materializing the neighborhoods, three key challenges must be solved.

In the following discussion, we assume that all sets of R are processed in processing order. When the *current set* r_i is to be processed, we know the core status of all preceding sets, $r_j \prec r_i$, but we do not know the core status of any unprocessed sets, $r_k \succ r_i$. We further assume that all sets $r \in R$ that are directly density-reachable from any r_j that precedes r_i (i.e., are neighbors of a core point $r_j \prec r_i$) are assigned to the same cluster as the core point r_j; this may also include sets $r_k \succ r_i$ that have not been processed yet.

Core Status. A set r_i is core if $|N_\epsilon(r_i)| \geq$ minPts. Sym-Clust and Join-Clust have access to the full neighborhood, $N_\epsilon(r_i)$, thus deciding the core status of r_i is trivial. In contrast, Spread sees only the lookahead neighbors, $N_\epsilon^\succ(r_i)$. To identify the core status of r_i, however, additional knowledge about the size of the preceding neighborhood, $N_\epsilon^\prec(r_i)$, is required.

Consider probing $r_i = r_5$ in our running example. According to our assumptions, the core status of sets r_1–r_4 is known (only r_3 is core), and all neighbors of r_3 are in cluster $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$. An index lookup of r_5 returns $N_\epsilon^\succ(r_5) = \{r_9, r_{10}\}$. Since $\left|N_\epsilon^\succ(r_5)\right| + 1 = 3 < 4 =$ minPts we cannot decide if r_5 is core. In fact, r_5 should be classified core since the full neighborhood is $N_\epsilon(r_5) = \{r_2, r_5, r_9, r_{10}\}$ (cf. red circle in Figure 3).

Border vs. Noise. Assume that the current set r_i is a non-core point that is not assigned to any cluster. We need to decide if r_i is border or noise. A border point has at least one core point in its neighborhood. None of the preceding neighbors, $r_j \in N_\epsilon^\prec(r_i)$, is core, otherwise r_i would be assigned to the cluster of r_j. Thus, r_i is core iff one of the lookahead neighbors is core. Unfortunately, we do not know the core status of the lookahead neighbors and can therefore not label r_i as border or noise.

Assume a core point, $r_k \in N_\epsilon^\succ(r_i)$, among the lookahead neighbors of r_i. When r_k is processed, r_k will not see r_i in its lookahead neighborhood since $r_i \prec r_k$. Therefore, r_i will not be included into the cluster of r_k and will wrongly be classified noise. The challenge is to correctly decide the border status of r_i despite seeing only the lookahead neighbors of r_i and r_k.

In our running example, r_1 is processed first. Thus, no core points are known and no clusters exist. $N_\epsilon^\succ(r_1) = \{r_3\}$ and r_1 remains noise (cf. blue circle in Figure 3). When the neighbor r_3 of r_1 is processed, r_3 will be detected as a core point and start a new cluster. However, since r_3 only sees its lookahead neighbors, $N_\epsilon^\succ(r_3) = \{r_4, r_6, r_7\}$, r_1 is not included into the cluster and is not detected as a border point.

Disconnected Clusters. Assume that the current set r_i is core and there is a core point $r_j \prec r_i$ in a cluster C_j, $r_i \notin C_j$. The current set r_i will assign all its lookahead neighbors to its cluster, $C_i = C_i \cup N_\epsilon^\succ(r_i)$ (C_i can be a new cluster started by r_i or an existing cluster to which r_i belongs). Unfortunately, we cannot assume that C_i and C_j are indeed distinct clusters: there can be a core point $r_k \succ r_i$ that density-reaches both r_i and r_j, i.e., C_i and C_j should form a single cluster. In general, multiple subclusters of the same DBSCAN cluster may grow independently. The challenge is to identify subclusters that should be merged and to merge them efficiently.

We process the current set $r_i = r_6$ in our running example. According to our assumptions, we know that r_3 and r_5 are core and we are aware of two clusters, $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$, $C_5 = \{r_2, r_5, r_9, r_{10}\}$. In addition, assume that we know that r_6 is core. Then, r_6 extends its current cluster, C_3, with its lookahead neighbors $N_\epsilon^\succ(r_6) = \{r_8, r_{10}\}$. Note that r_{10} is already part of cluster C_5. Since we do not know the core status of r_{10}, we cannot decide if C_5 and C_3 should be merged into a single cluster. If r_{10} is core, r_5 and r_3 are density-reachable from r_{10} and should be in the same cluster. If r_{10} is a border point, however, the clusters must not be merged, and r_{10} can be assigned to either C_5 or C_3.

4.2 Data Structures

Disjoint-Set. The disjoint-set (or union-find) data structure maintains a dynamic collection of non-overlapping sets for n objects in $O(n)$ space [10, 41]. A typical use case is the efficient computation of (minimum) spanning trees. It supports three operations: (1) For a given element u, make_set(u) creates a new (singleton) set that contains u. (2) The union(u, v) operation merges the two sets that contain u resp. v into a new set. (3) find_set(u) returns the representative for the set that contains u or ∞ if u is not found. The amortized worst-case time complexity is $\Theta(\alpha(n))$ for all operations, $\alpha(.)$ being the inverse Ackermann function. In practice, $\alpha(n)$ is considered a constant. In our setting, set elements are subclusters, and the disjoint-set data structure links subclusters that belong to the same DBSCAN cluster.

Backlinks. The backlinks data structure of a set $r \in R$ is a collection of unique references to other sets s that precede r, $s \prec r$. The backlinks bl support the add operation, $bl \cup \{s\}$, which adds a reference to a new set s in time $O(1)$ (on average). Depending on the type of sets that are referenced in the backlinks, we distinguish core and non-core backlinks, denoted *c_bl* and *nc_bl*, respectively. We implement backlinks as unordered sets of integer identifiers.

4.3 The Algorithm

Algorithm 5 shows the pseudocode of Spread. We use the following notation: r is the current probing set, $s \succ r$ is a lookahead neighbor, and $x \prec r$ is a preceding neighbor. Initially, all sets are noise, i.e., their cluster identifier is $-\infty$, $\forall r \in R : r.cid = -\infty$. Although we initialize all sets in Algorithm 5 explicitly (lines 3–4), this can also be done during indexing (cf. Algorithm 2).

Algorithm Outline. Spread proceeds in three main steps: (1) A counter and the processing order guarantee that the cardinality of the ϵ-neighborhood is known when a set is processed despite using the asymmetric prefix index. (2) Each set is assigned to a subcluster solely based on its lookahead neighboorhood. Subclusters of the same DBSCAN cluster are linked in a subcluster graph. Backlinks ensure that we do not miss border sets or links between subclusters. (3) Each connected component in the subcluster graph represents a DBSCAN cluster.

Core Status. A set r is core if $|N_\epsilon(r)| \geq$ minPts. In Spread, however, only $N_\epsilon^\succ(r)$ is computed. To capture the cardinality of $N_\epsilon^\prec(r)$, we store a density counter with each set r, denoted $r.dens$. Initially, $\forall r \in R : r.dens = 1$. For every lookahead neighbor $s \in N_\epsilon^\succ(r)$, $r.dens$ and $s.dens$ are incremented (due to the symmetry of the distance). Core set identification is highlighted in green.

Border vs. Noise. A probing set r that is not core is a border set iff $\exists y \in N_\epsilon(r) : y$ is core. Due to our processing order and the fact that only $N_\epsilon^\succ(r)$ is computed, the existence of a core neighbor y may be unknown when r is probed. However, for each $s \in N_\epsilon^\succ(r)$, we know that r is part of $N_\epsilon^\prec(s)$. We store this information by adding r to the non-core backlinks $nc\_bl[s]$ of each $s \in N_\epsilon^\succ(r)$ (lines 31–33). Then, the first $s \in N_\epsilon^\succ(r)$ that becomes core claims r (and all other unassigned sets in $nc\_bl[s]$) as border point for its subcluster. If none of the neighbors $s \in N_\epsilon^\succ(r)$ becomes core, then r remains noise. Lines 26–30 deal with a special case: If any $s \in N_\epsilon^\succ(r)$ is already core when r is probed, then s claims r immediately without adding r to its non-core backlinks. The relevant code lines are marked in red.

Subcluster Linkage. If the probing set r is core and a core neighbor y is part of another subcluster, the subclusters of r and y must be linked in our subcluster graph. The subcluster graph represents all connected components of subclusters, each of which is

a DBSCAN cluster. We use the disjoint-set data structure ds to track the connected components. Two subclusters u, v are linked by $ds.\text{union}(u, v)$. We may not be able to determine if there is a set $s \in N_\epsilon^\succ(r)$ that is core before s is probed. We use the core backlinks, c_bl, to book-keep potential subclusters for linkage: r adds its subcluster identifier to $c\_bl[s]$ of each $s \in N_\epsilon^\succ(r)$ (lines 22-23). After $N_\epsilon^\succ(r)$ has been processed, a link between the subcluster of r and every entry in $c\_bl[r]$ is created (line 24). The special case when s is already core allows us to create the link immediately without using core backlinks (lines 20–21). Linkage is only required if two subclusters coalesce (condition in line 19). Otherwise, r simply claims $s \in N_\epsilon^\succ(r)$ for its subcluster (lines 17–18). Linkage of subclusters is highlighted in blue ▢.

All backlinks of r are released after r has been processed to save memory (line 34). The subcluster graph in ds is used to assign consistent cluster IDs in a final scan over R (lines 35–36).

4.4 Correctness

We show that Algorithm 5 partitions R into DBSCAN clusters (cf. Definition 2.1). Set $r_i \in R$ is the i-th set of R in processing order. We prove the correctness by induction over i and increasing subsets $R^i \subseteq R$. $R^0 = \emptyset$, $R^i = R^{i-1} \cup \{r_i\} \cup N_\epsilon^\succ(r_i)$ for $1 \leq i \leq n = |R|$, thus $R^n = R$. Due to space constraints, we omit the full proofs and only provide the invariants that must be shown.

Core Status. The core status of set r_i is determined in the i-th iteration of the main loop. r_i is core if $\text{minPts} \leq |N_\epsilon(r_i)| = 1 + \left|N_\epsilon^\prec(r_i)\right| + \left|N_\epsilon^\succ(r_i)\right|$. In line 5, $r_i.dens = 1 + \left|N_\epsilon^\prec(r_i)\right|$. Lines 6–11 compute $N_\epsilon^\succ(r_i)$. The index lookup in line 6 returns candidate set M, $N_\epsilon^\succ(r_i) \subseteq M \subseteq \{s \mid s \succ r_i\}$. Every set $s \in M$ is verified in line 9 such that $N_\epsilon^\succ(r_i)$ is available starting from line 12.

Lemma 4.1. *Algorithm 5 correctly identifies* all *core sets in R.*

Proof Sketch. We show that at the start of the i-th iteration in line 5, for all r_k and r_j, $1 \leq k < i \leq j$ the following invariants hold: (I1) $r_k.dens = |N_\epsilon(r_k)|$; (I2) $r_j.dens = 1 + \left|\left\{r_k \mid r_j \in N_\epsilon^\succ(r_k)\right\}\right|$, i.e., $r_i.dens = 1 + \left|N_\epsilon^\prec(r_i)\right|$. Further, (I3) in line 12 of the i-th iteration, $r_i.dens = |N_\epsilon(r_i)|$, i.e., Algorithm 5 correctly identifies the core status of r_i. □

Border vs. Noise. Lines 25–33 cover the case that r_i is not core. If any $s \in N_\epsilon^\succ(r_i)$ qualifies as core, s claims r_i. Otherwise, r_i is stored in the non-core backlinks $nc\_bl[s]$ of every $s \in N_\epsilon^\succ(r_i)$ (lines 31–33). The next core neighbor in processing order claims r_i (lines 14–15) such that all border sets are assigned to a cluster.

Lemma 4.2. *Algorithm 5 correctly clusters* all *border sets in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all border sets $r_k \in \mathcal{B}$, $1 \leq k < i$: if r_k is not stored in $nc\_bl[s]$ for any $s \in N_\epsilon^\succ(r_k)$, $s = r_i$ or $s \succ r_i$, then r_k is assigned to the cluster of a core point in its neighborhood. □

Subcluster Linkage. Lines 12–24 cover the case that r_i is core. Each core point may form a subcluster on its own or together with other core points. We must ensure that all subclusters of the same DBSCAN cluster are linked in the disjoint-set, ds.

Lemma 4.3. *Algorithm 5 correctly links* all *subclusters in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all core neighors $c \in CN(r_k) = N_\epsilon(r_k) \cap C$ of a core set $r_k \in C$, $1 \leq k < i$: (a) c and r_k have the same cluster representative (in ds), or (b) c is stored in some $c\_bl[s]$, $s \in N_\epsilon^\succ(r_k)$, $s = r_i$ or $s \succ r_i$. □

Algorithm 5: Spread(R, ϵ, minPts)

Input: Collection of sets R, distance threshold ϵ, min. density minPts
Result: A correct DBSCAN clustering of R w.r.t. ϵ, minPts

```
1  ds ← new disjoint-set; nc_bl, c_bl ← new backlinks;
2  I ← Create-Index(R, ε);
3  foreach r ∈ R do
4      r.dens ← 1; r.cid ← −∞; ds.make_set(r.id);
5  foreach r ∈ R in processing order do
6      M ← Probe(r, I, ε);
7      N_ε^≻(r) ← ∅;
8      foreach (s, po) ∈ M do // po ... prefix overlap
9          if Verify-Pair(r, s, ε, po) then
10             r.dens ← r.dens + 1; s.dens ← s.dens + 1;
11             N_ε^≻(r) ← N_ε^≻(r) ∪ {s};
12     if r.dens ≥ minPts then // r is core
13         if r.cid = −∞ then r.cid ← r.id;
14         foreach x ∈ nc_bl[r] do // claim border sets x ≺ r
15             if x.cid = −∞ then x.cid ← r.cid;
16         foreach s ∈ N_ε^≻(r) do // s ≻ r
17             if s.cid = −∞ then // claim unclaimed s ≻ r
18                 s.cid ← r.cid
19             else if r.cid ≠ s.cid then // s already claimed
20                 if s.dens ≥ minPts then // s is core
21                     ds.union(r.cid, s.cid) // link subclusters
22                 else // remember core neighbor r
23                     c_bl[s] ← c_bl[s] ∪ {r.cid}
24         foreach x ∈ c_bl[r] do ds.union(r.cid, x);
25     else // r is not core, i.e., r.dens < minPts
26         if r.cid = −∞ then // claim potential border set r
27             foreach s ∈ N_ε^≻(r) do
28                 if s.dens ≥ minPts then // s is core
29                     if s.cid = −∞ then s.cid ← s.id;
30                     r.cid ← s.cid; break;
31         if r.cid = −∞ then // remember potential border set r
32             foreach s ∈ N_ε^≻(r) do
33                 nc_bl[s] ← nc_bl[s] ∪ {r}
34     release c_bl[r] and nc_bl[r] // not needed anymore
35 foreach r ∈ R do // final assignment of cluster IDs
36     if r.cid ≠ −∞ then r.cid ← ds.find_set(r.cid);
```

Theorem 4.4. *Algorithm 5 returns a correct set clustering* $\Gamma = \{C_1, C_2, \ldots, C_k\}$ *of R according to Definition 2.1.*

Proof Sketch. By Lemmata 4.1–4.3 and due to our final scan over R (lines 35–36), $x.cid = ds.\text{find\_set}(x.cid)$ holds for all $x \in R$. Initially, $x.cid = -\infty$ for all $x \in R$. The cluster IDs are updated only for border and core sets. Consequently, $x.cid = -\infty$ holds for all $x \in R \setminus (C \cup \mathcal{B}) \equiv \mathcal{N}$, i.e., also noise is correctly identified. □

4.5 Complexity Analysis

Memory. The asymmetric prefix index requires $O(n)$ space. In addition, Spread maintains the following data structures. (i) A density counter for each set $r \in R$ requires $O(n)$ space. (ii) A disjoint-set data structure with at most $O(n)$ entries, i.e., the disjoint-set structure requires $O(n)$ space [41]. (iii) In the worst case, we allocate two backlink structures for each $r \in R$, i.e., $O(n)$ backlinks. We release $c\_bl[r]$ and $nc\_bl[r]$ after probing r. Backlinks are only extended in lines 23 and 33. However, both lines are executed iff $\nexists s \in N_\epsilon^>(r) : s$ is core. Set s is core iff $s.dens \geq$ minPts, and the density is updated for every neighbor, therefore any backlink holds at most minPts entries. As a result, no more than $O(n \cdot \text{minPts})$ entries are allocated, thus requiring $O(n)$ space since minPts and ϵ are constants. *Runtime.* For each $r \in R$, we process $O\left(\left|N_\epsilon^>(r)\right|\right)$ neighbors and the backlinks of r if it is core. Recall that the disjoint-set operations take constant time. Therefore, the final for-loop (lines 35–36) runs in $O(n)$ time. Overall, Spread runs in $O\left(n^2\right)$ time and $O(n)$ space.

4.6 Multi-core Extension

Spread is designed as a single-core algorithm. We sketch an extension to multi-core processors that requires little synchronization between threads. Our extension is based on the observation that Spread spends most of the runtime in neighborhood computations (lines 6-11). While for some datasets the neighborhood computation accounts for only about half of the overall runtime (e.g., 55% for ORKUT, $\epsilon = 3$), for the configuration with the highest runtime in our experiments (CELONIS1, $\epsilon = 5$), Spread spends over 99% of the runtime in computing the neighborhoods.

We distribute the workload to $k+1$ threads, $T_1, T_2, \ldots, T_{k+1}$. Threads $T_1 - T_k$ are responsible for the neighborhood computations (lines 6-11), T_{k+1} performs the actual clustering (lines 12-34). The runtime of the other steps in the algorithm is negligible.

Neighborhood Computation. Let $r_i \in R$, $1 \leq i \leq |R|$ be the i-th set of R in processing order. Thread T_j, $1 \leq j \leq k$, computes the neighborhoods $N_\epsilon^>(r_i)$ of all r_i with $j = i \mod k$ (i.e., round robin). Each thread processes the assigned sets r_i in processing order (i.e., increasing values of i). The neighborhood computation in Algorithm 5 is interleaved with updating the density counters of r_i and its neighbors. Only this step requires synchronization (e.g., using atomic writes) since multiple threads may access the same counter concurrently. We do not expect congestions since the density updates are distributed over all neighbors.

Cluster Scan. Thread T_{k+1} scans the sets in processing order and performs the steps in lines 12-34 (maintain backlinks and disjoint-set, assign preliminary cluster IDs). After processing a set r_i, the memory for the neighbors of r_i is released.

Synchronization. We need to make sure that T_{k+1} processes set r_i only after r_i's neighbors have been computed. This can be achieved with a lock (implemented as condition variable[2]) on r_i that is held by T_j, $j \leq k$, until the neighborhood of r_i is computed. T_{k+1} needs to get the lock on r_i before processing it.

Memory. T_{k+1} releases the neighbors after processing them. If the parallel neighborhood computation is faster than T_{k+1}, the precomputed neighborhoods will fill up the memory. This is avoided with a shared counter that is incremented by $T_1 - T_k$ (when they process a new set r_i) and is decremented by T_{k+1} (after processing r_i). The neighborhood computation of r_i is postponed until the counter is below some threshold that bounds the number of concurrently materialized lookahead neighborhoods.

[2] A queue of threads waiting for a condition to become true.

Table 2: Characteristics of datasets.

Dataset	Coll. Size	Set Size		Univ. Size
		avg.	max.	
BMS-POS[4]	$3.2 \cdot 10^5$	9.3	164	$1.7 \cdot 10^3$
FLICKR[5]	$1.2 \cdot 10^6$	10.1	102	$8.1 \cdot 10^5$
KOSARAK[6]	$6.1 \cdot 10^5$	11.9	$2.5 \cdot 10^3$	$4.1 \cdot 10^4$
LIVEJ[7]	$3.1 \cdot 10^6$	36.4	300	$7.5 \cdot 10^6$
ORKUT[7]	$2.7 \cdot 10^6$	119.7	$4.0 \cdot 10^4$	$8.7 \cdot 10^6$
SPOT[8]	$4.4 \cdot 10^5$	12.8	$1.2 \cdot 10^4$	$7.6 \cdot 10^5$
CELONIS1	$8.2 \cdot 10^6$	20.3	91	$1.2 \cdot 10^4$
CELONIS2	$2.6 \cdot 10^6$	22.1	130	$3.5 \cdot 10^3$

5 EXPERIMENTAL EVALUATION

Algorithms. We compare our solution, Spread, against the two baseline approaches Sym-Clust and Join-Clust (cf. Section 3). All algorithms are single-threaded C++ implementations (2017 standard). Our implementations of Spread, Join-Clust, and the index of Sym-Clust follow the guidelines by Mann et al. [29], e.g., regarding symmetric and asymmetric prefix index, candidate generation, and optimized prefix-based verification.

Datasets. We execute all experiments on 13 real-world datasets: (a) Nine of the datasets where previously used for benchmarking set similarity joins [16, 29]: BMS-POS, DBLP, ENRON, FLICKR, KOSARAK, LIVEJ, NETFLIX, ORKUT, and SPOT. For a description of the datasets and preprocessing instructions[3] we refer to Mann et al. [29]. (b) Four large real-world datasets from the process mining domain, CELONIS1–4, that store one set per process. Compared to most datasets of the join benchmark, the universe size of these datasets is rather small. Table 2 summarizes important characteristics of our benchmark data.

Due to space constraints we omit detailed results for the following datasets: (a) DBLP, ENRON, and NETFLIX show very low runtimes (< 4s) and a small and stable memory footprint (< 1GiB) for all algorithms and configurations. (b) CELONIS3–4 show results similar to the other process mining datasets.

Parameters. The algorithms take two parameters: the neighborhood radius, ϵ, and the density, minPts. Typically, density-based clustering is sensitive to ϵ and quite robust to minPts. In our experiments, we vary both parameters: $\epsilon \in \{2, \mathbf{3}, 4, 5\}$ and minPts $\in \{2, 4, 8, \mathbf{16}, 32, 64, 128\}$ (defaults in bold font).

Environment. All experiments have been conducted on a 64-bit machine with 2 physical Intel Xeon E5-2630 v3 CPUs, 2.40GHz, 8 cores each (i.e., 16 logical processors, hyper-threading disabled). The cores share a 20MiB L3 cache and have another 256KiB of independent L2 cache. The system has 96GiB of RAM and runs Debian 10 Buster (Linux 4.19.0-12-amd64 #1 SMP Debian 4.19.152-1 (2020-10-18)). Our code is compiled with `clang`[9] version 7, highest optimization level (`-O3`). The runtime is measured with `clock_gettime`[10] at process level, memory usage is the `heap`

[3] http://ssjoin.dbresearch.uni-salzburg.at/datasets.html
[4] BMS-POS: http://www.kdd.org/kdd-cup/view/kdd-cup-2000 [52]
[5] FLICKR: Bouros et al. [6]
[6] KOSARAK: http://fimi.uantwerpen.be/data/
[7] LIVEJ, ORKUT: http://socialnetworks.mpi-sws.org/data-imc2007.html [30]
[8] SPOT: Pichl et al. [33]
[9] https://releases.llvm.org/7.0.0/tools/clang/docs/ReleaseNotes.html
[10] https://man7.org/linux/man-pages/man2/clock_gettime.2.html

Table 3: Index & cluster statistics for $\epsilon = 3$, minPts = 16.

(a) BMS-POS.

	Candidates	True Positives	Clusters
Sym-Clust	$3.9 \cdot 10^9$	$38.0 \cdot 10^6$	1
Join-Clust	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1
Spread	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1

(b) KOSARAK.

	Candidates	True Positives	Clusters
Sym-Clust	$40.7 \cdot 10^9$	$2.8 \cdot 10^9$	5
Join-Clust	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5
Spread	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5

(c) CELONIS1.

	Candidates	True Positives	Clusters
Sym-Clust	$644.6 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Join-Clust	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Spread	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075

peak of Linux `memusage`[11] (using `LD_PRELOAD`). A single instance is executed at a time with no other load on the machine.

5.1 Index & Cluster Statistics

We compare the number of candidates, true positives, and the number of clusters. The numbers are sums over all region queries. Table 3 shows the results obtained for BMS-POS, KOSARAK, and CELONIS1. We observe that Spread produces exactly the same number of candidates as Join-Clust since both solutions use the asymmetric index. Sym-Clust generates significantly more candidates due to the symmetric prefix index and the symmetric distance computations. For CELONIS1, Spread and Join-Clust verify about 5 times fewer candidates compared to Sym-Clust.

5.2 Runtime Efficiency

We measure the overall runtime, i.e., the CPU time that is required to cluster all sets into DBSCAN clusters (excluding the time to load the data from disk). Figure 7 shows the results for varying ϵ (minPts = 16). We observe that Sym-Clust is not competitive in terms of overall runtime in most cases. For all datasets, except KOSARAK and SPOT, the runtime of Sym-Clust increases much faster with ϵ than observed for Join-Clust and Spread. This is mainly due to the use of the symmetric prefix index (more candidates) and redundant computations (symmetric pairs).

Our experiments reveal that Join-Clust suffers from the following issues: (i) High runtimes for LIVEJ, ORKUT, and SPOT due to the expensive neighborhood materialization. (ii) Join-Clust runs out of memory for many instances (missing points in plots), in particular for FLICKR (any ϵ), KOSARAK ($\epsilon \geq 4$), LIVEJ, ORKUT, and SPOT ($\epsilon \geq 3$).

Spread outperforms its competitors in most settings and is competitive with Join-Clust otherwise (cf. Figures 7a, 7g, and 7h). For CELONIS1 and CELONIS2, Spread outperforms Sym-Clust by almost an order of magnitude and is competitive with Join-Clust.

Figure 8 shows the runtime results for varying minPts values ($\epsilon = 3$). We observe that the runtime of all three solutions is quite robust to minPts. The insights are similar for all datasets and values of ϵ. We include the plots for BMS-POS and KOSARAK.

[11] https://man7.org/linux/man-pages/man1/memusage.1.html

5.3 Memory Efficiency

We study the memory usage of Join-Clust, Sym-Clust, and Spread. All three solutions store (i) the collection, (ii) the inverted index, (iii) the candidates, and (iv) the result of a region query on the heap. The symmetric prefix index of Sym-Clust is larger than the asymmetric index, but still linear in the collection size. Sym-Clust generates more candidates than Join-Clust and Spread (cf. Section 5.1), which both use the asymmetric prefix index . Join-Clust materializes all neighborhoods in main memory. Sym-Clust and Spread materialize only a single neighborhood at a time. Spread stores also backlinks and the disjoint-set in main memory.

Figure 11 shows our results for varying ϵ (minPts = 16, y-axis log scale). Join-Clust runs out of memory for many instances (cf. Section 5.2). The neighborhood materialization in Join-Clust can be memory intensive even for small values of ϵ. We observe different growth rates with increasing radius ϵ, which we attribute to the different neighborhood sizes. The memory consumption of Sym-Clust is significantly lower and robust to varying ϵ. Spread shows a similar behavior. In some cases (e.g., LIVEJ, ORKUT), Spread occupies even less memory than Sym-Clust. When few backlinks are materialized, the smaller asymmetric prefix index of Spread outweighs the storage overhead for the backlinks.

Figure 12 shows the memory usage over minPts ($\epsilon = 3$, log-log scale). The memory consumption of Sym-Clust and Join-Clust is stable w.r.t. increasing values of minPts, while the memory usage of Spread slightly increases. This is due to the number of concurrently stored backlinks: the larger minPts, the higher the chance that a succeeding core neighbor is not yet classified, which triggers the creation of a backlink entry. The memory grows slowly with increasing minPts and does not limit the scalability of Spread. We include the results for BMS-POS and KOSARAK, $\epsilon = 3$; other datasets and ϵ values show similar results.

Backlinks Peak. We evaluate the effect of releasing the backlinks of a set in Spread after the set has been processed (cf. line 34, Algorithm 5). Figures 10 and 14 show the peak number of allocated backlinks relative to the maximum number of backlinks for varying ϵ (minPts = 16) and minPts ($\epsilon = 3$), respectively. Since two backlink structures, core (green) and non-core (orange), are maintained for each set in R, at most $2|R|$ backlinks can be allocated (light blue). Deallocating the backlinks of probed sets is highly effective: Only a small fraction of the maximum number of backlinks is allocated at any point in time. For increasing values of ϵ and minPts also the number of allocated backlinks grows.

5.4 Scalability

We evaluate the scalability of Spread and its competitors to increasing data sizes. To this end, we increase the size of BMS-POS and KOSARAK using the procedure of Vernica et al. [42]. This approach does not affect the token universe, and the number of similar pairs in the dataset increases linearly with the data size.

Figure 9 (runtime) and Figure 13 (main memory) report the results for our default parameter setting. Spread shows runtimes similar to Join-Clust and outperforms Sym-Clust by a factor of about 12 (BMS-POS) resp. 5.7 (KOSARAK) on the largest dataset (×16). As we increase BMS-POS by a factor of 16, the runtime increases by a factor of 195 for Spread, 204 for Join-Clust, and 460 for SymClust. The memory grows linearly for all measured data points and increases by a factor of about 2 when we double the data size. Join-Clust requires 18-25 (BMS-POS) resp. 499-569 (KOSARAK) times more memory than its competitors and runs out of memory on KOSARAK except for the ×1 dataset.

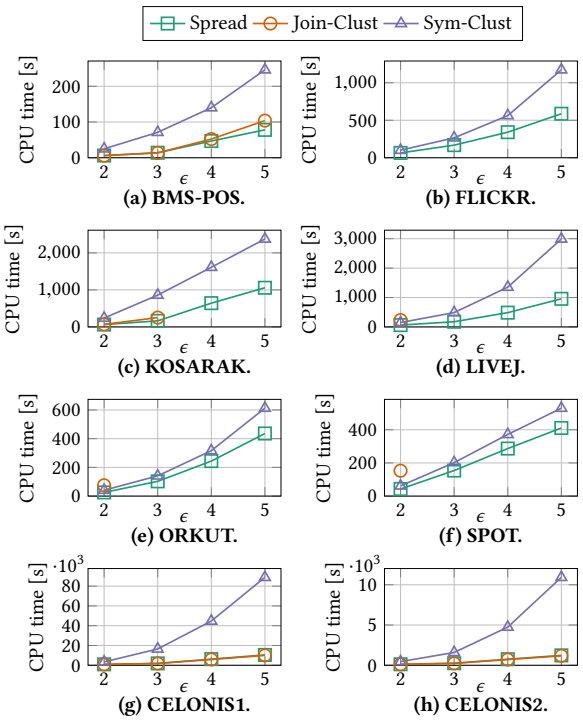

Figure 7: Runtime over ϵ, minPts = 16.

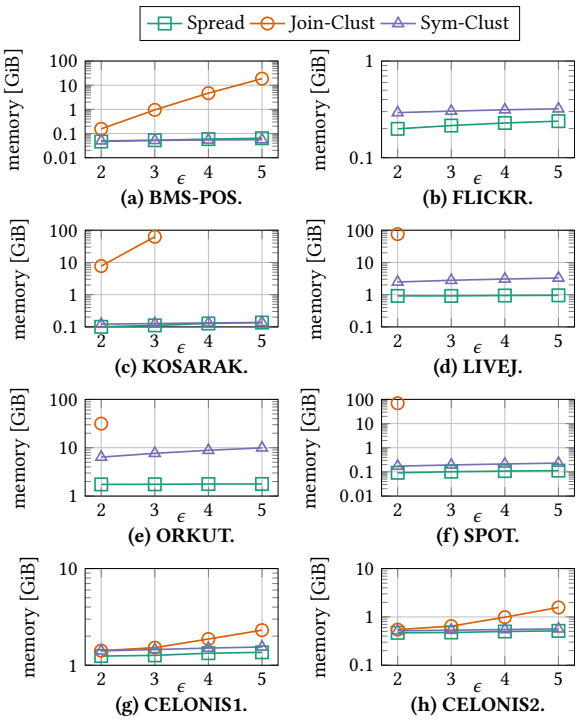

Figure 11: Main memory over ϵ, minPts = 16.

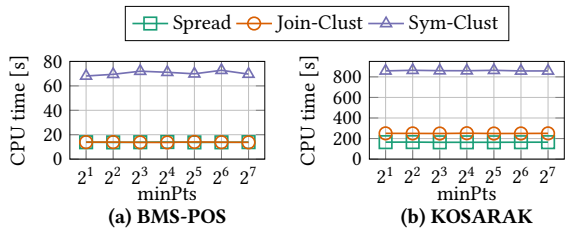

Figure 8: Runtime over minPts, ϵ = 3.

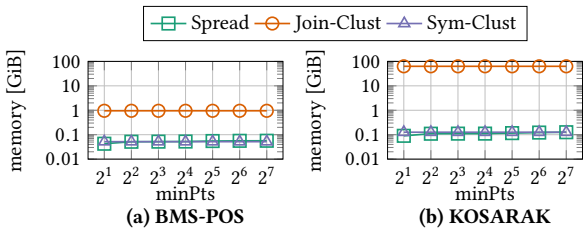

Figure 12: Main memory over minPts, ϵ = 3.

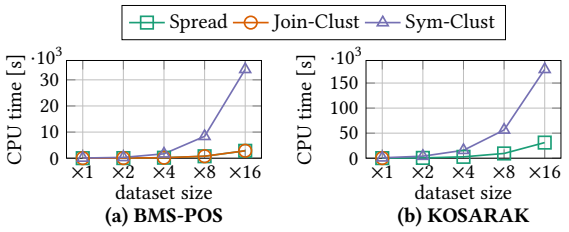

Figure 9: Runtime over data size, ϵ = 3, minPts = 16.

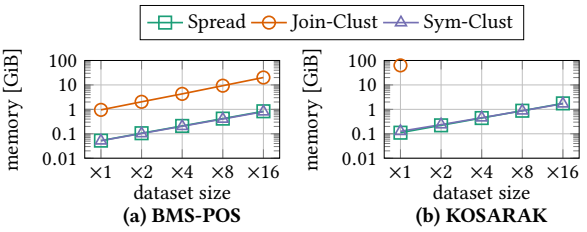

Figure 13: Main memory over data size, ϵ = 3, minPts = 16.

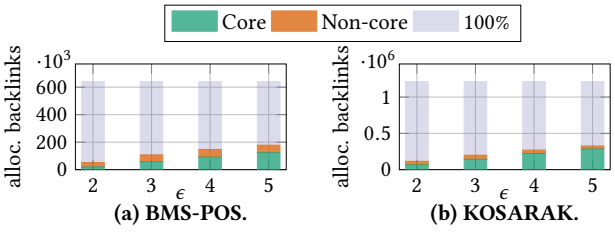

Figure 10: Backlinks peak over ϵ, minPts = 16.

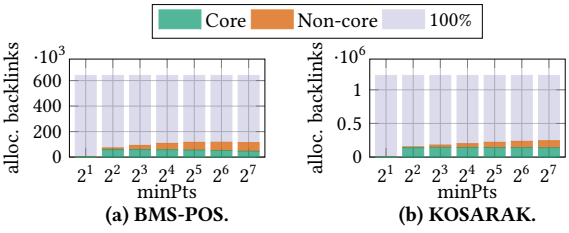

Figure 14: Backlinks peak over minPts, ϵ = 3.

Summarizing, Spread clearly outperforms Sym-Clust in runtime (by a factor of 5-12) and Join-Clust in memory usage (by more than an order of magnitude) as we increase the data size.

6 RELATED WORK

Indexes for Sets. Most set similarity joins operate on an inverted list index that maps signatures to candidate sets. Various signatures have been proposed [2, 8, 40, 45]. Prefixes [8] in conjunction with the length filter [1] have been shown to prune sets effectively. More sophisticated filters include positional and suffix filter [49], the removal filter [35], the position-enhanced length filter [28], and the adaptive prefix filter [44]. Wang et al. [46] leverage the similarity of the sets in an ϵ-neighborhood to reduce the overall number of false positives. Dong et al. [13] propose a size-aware algorithm that runs in $o(n^2) + O(k)$ time for k result pairs. Qin and Xiao [34] propose the pigeonring, a generalization of the pigeonhole principle that yields stronger constraints. Indexing and join techniques for sets have been studied extensively in the single-machine [29] and the distributed context [16].

Most of these approaches focus on self-joins, which order the sets and compute the lookahead neighborhood to avoid symmetric distance computations. In our work, we use the prefix filter, but any of the other asymmetric indexes is applicable.

Efficient Region Queries. Ester et al. [15] propose the first exact DBSCAN algorithm with $O(n \log n)$ runtime for vectors of arbitrary dimension. $O(n \log n)$ runtime holds for a small number of neighbors (compared to n) and an index with $O(\log n)$ lookup time. Henceforth, efficient region query computation has been of great interest and many improvements have been proposed. Brecheisen et al. [7] use minPts-nearest neighbor queries to identify core points and postpone the other distance computations until the distances are required to get a correct DBSCAN clustering. The proposed *Xseedlist* data structure is designed for expensive distance functions and assumes a cheap but selective filter. These assumptions do not hold for sets: The verification (i.e., distance computation) of candidate pairs has shown to be highly efficient [29] (a small number of integer comparisons). Brecheisen et al. must insert the candidates into the Xseedlist data structure, which maintains sorted lists of candidates. Due to the expensive sorting procedure, we do not expect Xseedlist to improve the DBSCAN algorithm for sets. TI-DBSCAN [25] exploits the triangle inequality to reduce the search space of region queries. The solution is not index-based, sorts the points w.r.t. a reference point, and shifts a window of size 2ϵ over the sorted points. The reference point is the point with minimal values in all dimensions. This is equivalent to the empty set, and our processing order in combination with the prefix index for sets subsumes this technique. Patwary et al. [32] introduce PARDICLE, a parallel approximate density-based clustering algorithm for Euclidean space. Its aim is to reduce the neighborhood computation time by sampling high-density regions. Kumar and Reddy [26] propose a new graph-based index structure called Groups. It discovers groups of patterns in two scans over the dataset and applies a standard DBSCAN afterwards. Groups accelerates region queries by pruning noise points effectively. This technique assumes Euclidean distance and does not consider Hamming distance or other set similarity measures. Recently, Jiang et al. [24] proposed SNG-DBSCAN, which prevents the computation of the full ϵ-neighborhood graph via subsampling its edges. This results in $O(sn^2)$-time complexity with s being the sampling rate. Under certain distribution assumptions, SNG-DBSCAN has been shown to preserve the exact ϵ-neighborhood graph for $s \approx (\log n)/n$ with $O(n \log n)$ runtime.

DBSCAN Techniques. Yang et al. [51] propose the distributed DBSCAN-MS clustering algorithm for metric spaces. DBSCAN-MS uses pivots to map the data from metric space to vector space, where it is partitioned in order to be distributed. A local DBSCAN is then executed on each partition. Our solution does not rely on the metric properties of set distances, but uses specialized set indexes. However, our techniques may be leveraged in the context of DBSCAN-MS, where the data points are ordered by one of the dimensions for efficient neighborhood queries.

Patwary et al. [31] propose PDSDBSCAN, a parallel DBSCAN algorithm that uses the disjoint-set data structure to connect data points into clusters. We only insert links between subclusters into the disjoint-sets structure, while PDSDBSCAN inserts a link for each neighbor, rendering the number of required union operations a bottleneck for this approach.

Böhm et al. [3] use a block-nested loop join and buffer the join result to reduce the number of block accesses required to compute ϵ-neighborhoods. CUDA-DClust [4] is a GPU-based solution that splits clusters into chains that are expanded from different starting points in parallel. In order to merge chains into clusters, a quadratic-size bit matrix is used. We maintain only a linear number of links and leverage disjoint-sets to merge clusters. Incremental DBSCAN algorithms [14] deal with updates on an existing clustering. Similar to our approach, these techniques may need to merge clusters when new points are inserted. None of the above solutions supports asymmetric neighborhood indexes.

Numerous parallel and distributed algorithms [9, 11, 18–21, 23, 36, 38, 47, 50] as well as approximations [17, 27, 43, 48] have been proposed. We present an exact, single-core solution for sets.

7 CONCLUSION

In this paper, we have investigated clustering techniques for large collections of sets. Our work was motivated by an application in process mining that models processes as sets to assess their similarity. We have shown that the solutions that are currently available, Sym-Clust and Join-Clust, are not satisfying: Sym-Clust is slow since it cannot use effective asymmetric set indexes, while Join-Clust is infeasible for many settings due to its excessive memory usage. We introduced a novel, density-based clustering algorithm, Spread, that can process data points in any user-defined order and is therefore fit for the use with asymmetric indexes. Spread combines the best of both worlds: It uses the effective asymmetric index of Join-Clust, but like Sym-Clust does not need to materialize the neighborhoods. We introduced so-called backlinks to guarantee a correct DBSCAN clustering and showed the correctness of our approach. To the best of our knowledge, Spread is the first DBSCAN-compliant algorithm that uses an asymmetric index and runs in linear space.

Spread uses the index as a black box and works with any data type. Interesting future work includes evaluating the performance of Spread for vector data, where candidates are generated using a sliding window that is shifted along one dimension. The data points in the window are candidates, i.e., the window simulates an asymmetric index for Spread.

ACKNOWLEDGMENTS

We thank Alexander Miller, Mateusz Pawlik, Thomas Hütter, Manuel Widmoser, Manuel Kocher, Daniel Ulrich Schmitt, Konstantin Thiel, Daniel Grittner, Christian Böhm, and Claudia Plant

for valuable discussions, and Manuel Kocher for typesetting Figures 1 and 3. This work was partially supported by the Austrian Science Fund (FWF): P 29859.

REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*. 918–929.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling Up All Pairs Similarity Search. In *Proc. of the Int. Conf. on World Wide Web (WWW)*. 131–140.

[3] Christian Böhm, Bernhard Braunmüller, Markus Breunig, and Hans-Peter Kriegel. 2000. High Performance Clustering Based on the Similarity Join. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 298–305.

[4] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. 2009. Density-Based Clustering Using Graphics Processors. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 661–670.

[5] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. 2010. Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models. In *Business Process Management Workshops*. 170–181.

[6] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-Textual Similarity Joins. *Proc. of the VLDB Endowment* 6, 1 (Nov. 2012), 1–12.

[7] S. Brecheisen, H. . Kriegel, and M. Pfeifle. 2004. Efficient Density-Based Clustering of Complex Objects. In *Proc. of the IEEE Int. Conf. on Data Mining*. 43–50.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*. 5–5.

[9] I. Cordova and T. Moh. 2015. DBSCAN on Resilient Distributed Datasets. In *Proc. of the Int. Conf. on High Performance Computing Simulation (HPCS)*. 531–540.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.).

[11] B. Dai and I. Lin. 2012. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *Proc. of the IEEE Int. Conf. on Cloud Computing*. 59–66.

[12] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An Efficient Partition Based Method for Exact Set Similarity Joins. *Proc. of the VLDB Endowment* 9, 4 (Dec. 2015), 360–371.

[13] Dong Deng, Yufei Tao, and Guoliang Li. 2018. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 905–920.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. 1998. Incremental Clustering for Mining in a Data Warehousing Environment. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*. 323–333.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. 226–231.

[16] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set Similarity Joins on Mapreduce: An Experimental Survey. *Proc. of the VLDB Endowment* 11, 10 (June 2018), 1110–1122.

[17] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 519–530.

[18] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: Highly Parallel DBSCAN. In *Proc. of the Workshop on Machine Learning in High-Performance Computing Environments*. Article 2, 10 pages.

[19] D. Han, A. Agrawal, W. Liao, and A. Choudhary. 2016. A Novel Scalable DBSCAN Algorithm with Spark. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1393–1402.

[20] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. 2014. MR-DBSCAN: A Scalable MapReduce-Based DBSCAN Algorithm for Heavily Skewed Data. *Frontiers of Computer Science* 8, 1 (2014), 83–99.

[21] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. 2011. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*. 473–480.

[22] B.F.A. Hompes, J.C.A.M. Buijs, W.M.P. van der Aalst, P.M. Dixit, and J. Buurman. 2015. Discovering Deviating Cases and Process Variants Using Trace Clustering. In *Benelux Conf. on Artificial Intelligence*.

[23] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. 2004. Scalable Density-Based Distributed Clustering. In *Knowledge Discovery in Databases (PKDD)*. 231–244.

[24] Heinrich Jiang, Jennifer Jang, and Jakub Łącki. 2020. Faster DBSCAN via subsampled similarity queries. *CoRR* (2020).

[25] Marzena Kryszkiewicz and Piotr Lasek. 2010. TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality. In *Rough Sets and Current Trends in Computing (RSCTC)*. 60–69.

[26] K. Mahesh Kumar and A. Rama Mohan Reddy. 2016. A fast DBSCAN clustering algorithm by accelerating neighbor searching using Groups method. *Pattern Recognition* 58 (2016), 39 – 48.

[27] Yinghua Lv, Tinghuai Ma, Meili Tang, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. 2016. An efficient and scalable density-based clustering algorithm for datasets with complex structures. *Neurocomputing* 171 (2016), 9 – 22.

[28] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Proc. of the Workshop Grundlagen von Datenbanken (CEUR Workshop Proceedings)*, Vol. 1313. 89–94.

[29] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *Proc. of the VLDB Endowment* 9, 9 (2016), 636–647.

[30] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proc. of the ACM Int. Conf. on Internet Measurement (SIGCOMM)*. 29–42.

[31] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[32] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey. 2014. Pardicle: Parallel Approximate Density-Based Clustering. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. 560–571.

[33] Martin Pichl, Eva Zangerle, and Günther Specht. 2014. Combining Spotify and Twitter Data for Generating a Recent and Public Dataset for Music Recommendation. In *Proc. of the Workshop Grundlagen von Datenbanken (CEUR Workshop Proceedings)*, Vol. 1313. 35–40.

[34] Jianbin Qin and Chuan Xiao. 2018. Pigeonring: A Principle for Faster Thresholded Similarity Search. *Proc. of the VLDB Endowment* 12, 1 (2018), 28–42.

[35] Leonardo Andrade Ribeiro and Theo Härder. 2011. Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems* 36, 1 (2011), 62–78.

[36] A. Sarma, P. Goyal, S. Kumari, A. Wani, J. S. Challa, S. Islam, and N. Goyal. 2019. μDBSCAN: An Exact Scalable DBSCAN Algorithm for Big Data Exploiting Spatial Locality. In *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER)*. 1–11.

[37] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems* 42, 3 (2017), 21.

[38] Hwanjun Song and Jae-Gil Lee. 2018. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 1173–1187.

[39] Minseok Song, Christian W. Günther, and Wil M. P. van der Aalst. 2009. Trace Clustering in Process Mining. In *Business Process Management Workshops*. 109–120.

[40] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2019. Balance-aware Distributed String Similarity-based Query Processing System. *Proc. of the VLDB Endowment* 12, 9 (2019), 961–974.

[41] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM* 22, 2 (1975), 215–225.

[42] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient Parallel Set-Similarity Joins Using MapReduce. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 495–506.

[43] P. Viswanath and V. Suresh Babu. 2009. Rough-DBSCAN: A fast hybrid density based clustering method for large data sets. *Pattern Recognition Letters* 30, 16 (2009), 1477 – 1488.

[44] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can We Beat the Prefix Filtering? An Adaptive Framework for Similarity Join and Search. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 85–96.

[45] Pei Wang, Chuan Xiao, Jianbin Qin, Wei Wang, Xiaoyang Zhang, and Yoshiharu Ishikawa. 2016. Local Similarity Search for Unstructured Text. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 1991–2005.

[46] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2017. Leveraging Set Relations in Exact Set Similarity Join. *Proc. of the VLDB Endowment* 10, 9 (May 2017), 925–936.

[47] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2555–2571.

[48] Y. Wu, J. Guo, and X. Zhang. 2007. A Linear DBSCAN Algorithm Based on LSH. In *Proc. of the Int. Conf. on Machine Learning and Cybernetics*, Vol. 5. 2608–2614.

[49] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient Similarity Joins for Near-Duplicate Detection. *ACM Transactions on Database Systems* 36, 3 (2011), 41.

[50] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. 1999. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery* 3, 3 (1999), 263–290.

[51] K. Yang, Y. Gao, R. Ma, L. Chen, S. Wu, and G. Chen. 2019. DBSCAN-MS: Distributed Density-Based Clustering in Metric Spaces. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*. 1346–1357.

[52] Zijian Zheng, Ron Kohavi, and Llew Mason. 2001. Real World Performance of Association Rule Algorithms. In *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. 401–406.