Scalable All-Pairs Similarity Search in Metric Spaces

Ye Wang
The Ohio State University
wangy@cse.ohio-state.edu

Ahmed Metwally
Google Inc.
metwally@google.com

Srinivasan Parthasarathy
The Ohio State University
srini@cse.ohio-state.edu

ABSTRACT

Given a set of entities, the all-pairs similarity search aims at identifying all pairs of entities that have similarity greater than (or distance smaller than) some user-defined threshold. In this article, we propose a parallel framework for solving this problem in metric spaces. Novel elements of our solution include: i) flexible support for multiple metrics of interest; ii) an autonomic approach to partition the input dataset with minimal redundancy to achieve good load-balance in the presence of limited computing resources; iii) an on-the-fly lossless compression strategy to reduce both the running time and the final output size. We validate the utility, scalability and the effectiveness of the approach on hundreds of machines using real and synthetic datasets.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Performance

Keywords

Similarity Joins; All-Pairs Similarity Search

1. INTRODUCTION

The all-pair similarity search (APSS) problem seeks to find all pairs of records (or entities) within a dataset that meet a user-defined similarity threshold, based on some definition of similarity. This problem has found applications in many domains including community discovery [29], duplicate detection [16], collaborative filtering [19] and as a preprocessing step for clustering [18].

The motivating application behind this work is the detection of click fraud rings [21]. Since an Internet content publisher collects revenue for each click on any ad displayed on her page, she has the incentive to launch click inflation attacks by generating fake clicks. The identification of publisher accounts that have similar features, from an enormous amount of accounts with legitimate traffic, is very effective in detecting such attacks. This can be achieved by representing each account as a point on a multi-dimensional space, and efficiently solving the APSS problem on all such points.

A significant amount of related work has focused on set similarity measures (e.g. Jaccard or Cosine) operating on binary or categorical data [31, 5, 27, 22]. The key to their scalability is effectively leveraging the inherent sparsity in such data [26, 8, 6, 31, 22].

In contrast to the aforementioned sparse APSS problem, we focus on the dense APSS problem [28]. In the motivating application, publishers are typically represented using vectors of numerical features that are reasonably dense i.e., each record has a non-zero value for a significant fraction of the dimensions. To make the problem more tractable, we focus on the MAPSS problem, the APSS flavor of the problem where the similarity measure is a Metric. MAPSS has applications from image search [20] to time series analysis [11]. To make the solution more scalable, we propose a parallel algorithm. While our implementation is MapReduce-based [9], the algorithm is generalizable to other forms of parallelizing frameworks, such as MPI and OpenMP.

The proposed MR-MAPSS (MapReduce-based MAPSS) algorithm relies on a carefully engineered partitioning scheme, where the backbone algorithm intelligently routes the input data records into worksets with minimal redundancy. These worksets are independent and each is processed by a MapReduce worker. This achieves load-balancing when handling a massive number of high-dimensional records. Repartitioning is employed to further split large worksets that occur when the input data is densely clustered. This improves the load balancing, and enables execution with limited per-machine resource budgets. For efficiency, MR-MAPSS also employs a multi-level indexing structure to combine computing the similar pairs of records with on-the-fly lossless compression of the results. MR-MAPSS is flexible, and caters to diverse measures of similarity, and even handles non-metric similarity measures as long as upper and lower bounds on the distances can be established. The main contributions are:

1. We propose MR-MAPSS, an intelligent divide-and-conquer partitioning scheme, where the data routing ensures each pair of records is evaluated at most once.
2. The MR-MAPSS framework automatically detects skewed workloads, and further distributes the computation among other machines.

3. The MR-MAPSS framework is agile and can adapt to streaming and memory constrained environments.

4. MR-MAPSS employs a novel compression strategy using a community-based approach, which requires no decompression in almost all of the follow-up jobs. This compression greatly reduces the running time and the output size, as verified in the experiments.

5. We establish the effectiveness of the optimizations on two real datasets, and demonstrate the scalability of MR-MAPSS on hundreds of machines.

2. BACKGROUND

Notation: We begin by briefly going over notation used in this paper (see Table 1). Note that the terms "record" and "point" will be used interchangeably in the rest of the paper.

Table 1: Notations

L	The dimensionality of the points
D	Database of points from $\mathbb{R}^L$
$\|D\|$	Number of points in the database
p_i	i^{th} point in the database
$dist(p_i, p_j)$	Distance between p_i and p_j
t	Distance threshold for a pair to be similar

MapReduce infrastructure: MapReduce [9] (MR) has become the de facto platform for big data processing in shared-nothing clusters due to its high scalability and built-in fault tolerance support. It borrows *Map* and *Reduce* concepts from functional programming. The computation can be represented using two functions:

Map : $\langle k_1, v_1\rangle \rightarrow [\langle k_2, v_2\rangle]$

Reduce : $\langle k_2, [v_2]\rangle \rightarrow [v_3]$

Each input record is a tuple $\langle k_1, v_1\rangle$. During the execution of a job, each mapper fetches a set of records from the distributed file system, and applies the map function on each single record to produce a list in the form of $[\langle k_2, v_2\rangle]$, where [.] represents a list of elements. The mappers can also output the tuples with a *secondary key*. The shuffler groups the output of the mappers by k_2 and sorts by the secondary key within each group, and sends all tuples with the same k_2 value to the same reducer. The reduce function receives the key and the list of values as input, and emits the results. A combiner, a function similar to reducer, can be used for partial reducing at the mapper's side to reduce the network load. The MR infrastructure allows for specifying initialization operations before the actual Mapper or Reducer starts. Examples of MR applications can be found in [9].

3. THE MR-MAPSS FRAMEWORK

This section describes the backbone algorithm of the MR-MAPSS framework. Optimizations to scale under tight computational constraints are discussed in Section 4 and 5.

3.1 Key Definitions and Intuitions

We start by introducing some notations.

DEFINITION 3.1. *SimSet: Given two sets of points S_1 and S_2, define $SimSet(S_1, S_2) = \{\langle p_1, p_2\rangle \mid dist(p_1, p_2) \leq t \wedge p_1 \in S_1 \wedge p_2 \in S_2 \wedge p1 \neq p2\}$.*

The MAPSS problem is to *find $SimSet(D, D)$ of Dataset D in a metric space.*

DEFINITION 3.2. *Inner sets, Outer sets, Worksets and APSS division: Assume there exist N sets $\{I_{i=1,\dots,N}\}$ such that $\cup_{i=1}^{N} I_i = D$, and N sets $\{O_{i=1,\dots,N}\}$ such that $O_i \subseteq D$. We call each $W_i = \langle I_i, O_i\rangle$ a workset of D. I_i, O_i are the inner set and outer set of W_i respectively. If $\cup_{i=1}^{N} SimSet(I_i, I_i \cup O_i) = SimSet(D, D)$, then $\{W_{i=1,\cdots N}\}$ is said to define an APSS division of D.*

From Definition 3.2, partitioning the APSS workload of D becomes a problem of finding an APSS division. There may

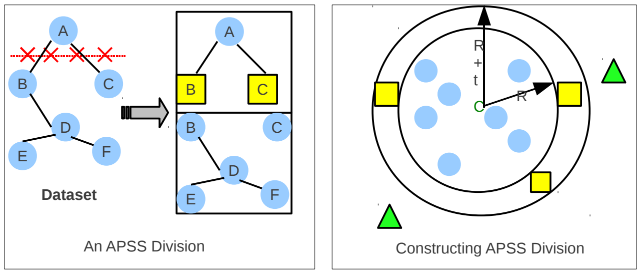


Figure 1: APSS Division and Their Constructions

be multiple APSS divisions. A trivial APSS division is to define inner and outer sets as $I_1 = \ldots = I_N = O_1 = \ldots = O_N = D$. The left part of Figure 1 illustrates the concept of APSS division. In the figure, the nodes are interconnected if similar. Circular and square points represent inner and outer points, respectively. Note that B and C serve as outer points for the upper workset, and serve as inner points for the lower one.

DEFINITION 3.3. *Partitions, Centroids, and Radii: D can be divided into N disjoint partitions, $P_i, \ldots, P_N$, such that $P_i \cap P_j = \emptyset\ \forall i \neq j$ and $\cup_{i=1}^{N} P_i = D$. Let N centroids, $c_1, \ldots, c_N$ be used to partition D. A point, p, belongs to partition P_i if and only if c_i is its closest centroid. The radius of P_i is defined as $r_i = max_{p_i \in P_i} dist(c_i, p_i)$.*

In metric space, one way to form worksets is to let I_i, the inner set of workset W_i, be P_i. O_i, the outer set of W_i, is given by $\{p_j \mid dist(p_j, c_i) \leq (r_i + t) \wedge p_j \notin P_i\}$. The right part of Figure 1 is an example of constructing workset from a partition consisting of all the circular points. All circles serve as inner points in the generated workset, while squares serve as the outer points. Triangles are irrelevant to this workset and thus discarded.

THEOREM 3.4. *The worksets formed above define a MAPSS division.*

PROOF. It suffices to show that all the pairs in $SimSet(I_i, I_i \cup O_i)$ are similar, and that no similar pair is missed. Each pair in $SimSet(I_i, I_i \cup O_i)$ is a similar pair, from Definition 3.1. This establishes that $\cup_{i=1}^{N} SimSet(I_i, I_i \cup O_i)$ only contains similar pairs. From Definition 3.3 and *Triangle Inequality (TI)*, $I_i \cup O_i$ contains all points similar to any point in I_i. Since from Definition 3.3, $D = \cup_{i=1}^{N} I_i$, it follows that $\cup_{i=1}^{N} SimSet(I_i, I_i \cup O_i) \supseteq SimSet(D, D)$. Hence $\{W_{i=1,\dots,N}\}$ defines an APSS division for D. □

Other approaches, such as the "generalized hyperplane partitioning", can also serve as alternatives to construct inner and outer sets. The "generalized hyperplane partitioning" is also implemented in our framework. Interested readers are referred to [30, 17] for more details. We omit this discussion here in the interest of space.

3.2 The Framework

Figure 2 depicts the MR-MAPSS framework. It has two major phases: data pre-processing and similarity computing. The data pre-processing phase comprises the CentroidSampling and CentroidStats steps. CentroidSampling selects random points as centroids. The output samples will be stored on disks and used in the follow-up step. CentroidStats step computes partition statistics, such as the radii of the centroids. The Similarity computation phase identifies all the similar pairs. It has either one or two chained MapReduce steps, depending on the computing resources and data characteristics. SimilarityMapper reads in the original dataset as well as the centroid statistics, and constructs the worksets for SimilarityReducer. SimilarityMapper uses secondary keys so that SimilarityReducer receives inner points before outer points. An optional repartitioning step is employed to split large worksets that do not fit in memory into smaller ones. The repartitioning work is done by the RepartitionReducer. The RepartitionMapper is almost an identity mapper, except for using secondary keys to provide the same guarantee of SimilarityMapper. Finally, SimilarityReducer computes $\cup_i SimSet(I_i, I_i \cup O_i)$.

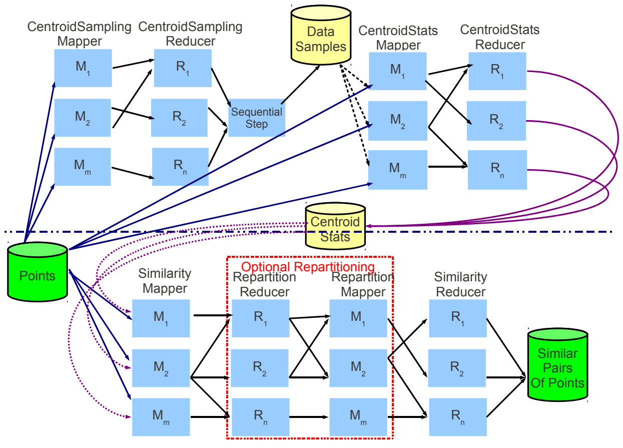


Figure 2: The MR-MAPSS Partitioning Framework

In the rest of the section, we describe the CentroidSampling, CentroidStats and Similarity steps without repartitioning. Repartitioning will be discussed in the Section 5.

3.2.1 *CentroidSampling and CentroidStats*

Selecting the right set of centroids(partitions) to minimize the *APSS computational cost* is a very difficult task. If each data point is treated as a vertex in the graph, the similar pair can be modeled as an edge connecting the corresponding vertices. When forming the partitions, a simple but intuitive partitioning approach is to find a set of N centroids for which the largest distance of any point to its closest centroid in the K-set is minimum. This problem is commonly known as *metric K-center*, a NP-complete problem [15]. In practice, the algorithm also needs to minimize the number of similar pairs formed by points in two different partitions. Considering the various system constraints makes the problem even harder.

The CentroidSampling step samples N centroids from the input. We tried methods such as KMeans++ [3], random sampling, as well as a seeding approach ensuring the distances between any pair of samples is no less than a threshold. It turned out random sampling and KMeans++ have better performance. We therefore chose random sampling due to the simplicity and scalability. CentroidStats computes the radius of the partitions. Centroid information is read in every mapper before the actual map operation starts. As a mapper processes a point, it outputs the closest centroid id as the key, and the corresponding distance as the value. For each centroid, a reducer receives a list of the distances of the inner points to that centroid, $[dist(p_j, c_j)]$. The reducer computes the radius for each centroid as the maximum distance in this list.

3.2.2 *SimilarityMapper*

Algorithm 1 Similarity

```
Input: $D$: the dataset; $\{\langle c_i, r_i \rangle\}$: the centroid statistics.
Output: $SimSet(D, D)$: the set of similar pairs.
  Map:
  $\langle i, p_i \rangle + \langle c_i, r_i \rangle \rightarrow [\langle c_i, \langle p_i, pointType, dist(p_i, c_i) \rangle \rangle]$

  Reduce:
  $\langle c_i, [\langle p_i, pointType, dist(p_i, c_i) \rangle] \rangle \rightarrow$
     $[\langle p_i, p_j \rangle \mid dist(p_i, p_j) \leq t \wedge p_i \text{ or } p_j \text{ is an inner point}]$
```

In the Similarity step (Algorithm 1), all the outer points of each workset are found. The mappers route the corresponding *inner* and *outer* points of each partition to the same reducer, and specify the *pointType* of each point as either inner or outer. Each reducer receives one of these worksets, computes all its similar pair of points and outputs them. Secondary keys that guarantee the reducer receives inner points before outer points are not shown for simplicity.

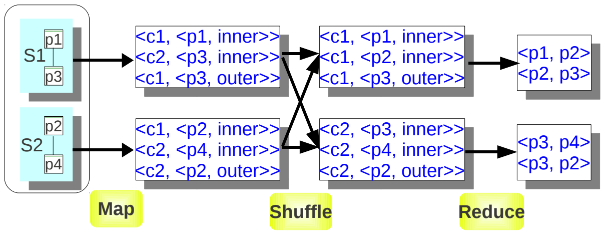


Figure 3: The MR-MAPSS Backbone Algorithm

Figure 3 provides an example of four data points that are aligned horizontally on increasing subscript order, such that the distance between any two neighboring points is 0.5. Partition P_1 has points p_1 and p_2. Its centroid, c_1, lies between p_1 and p_2. Similarly, partition P_2 contains p_3 and p_4, with centroid c_2 lying between them. Hence, $r_1 = r_2 = 0.25$. For simplicity, the centroids are assumed not to be part of the dataset. The distance threshold is set to 0.6. p_1 and p_3 reside in shard S_1, while the other two points reside in shard S_2. From Definition 3.3, p_2 and p_3 are outer points of P_1 and P_2, respectively. During the shuffle phase, p_1 and p_3 are routed to W_1, and p_2 and p_4 are routed to W_2 as inner points. In addition, p_2 and p_3 are routed as outer points to W_1 and W_2, respectively. The two reducers then compute all the pairs in $SimSet(I_i, I_i \cup O_i)$ for $i = 1, 2$.

4. OPTIMIZATIONS

Exploting Commutativity: So far, the mappers are assumed to send out the outer points of all the worksets. One drawback of this naïve routing scheme is the existence of redundant computation. In Figure 3, the two points p_2 and p_3 were sent to W_1 and W_2, respectively. This causes the pair $\langle p_2, p_3 \rangle$ to be computed by both reducers. More generally, assuming the dataset is N-way partitioned, and $OP(P_i, P_j) = \{a \mid a \in I_i \wedge a \in O_j\}$ then the number of unnecessary computations in the reduce phase is at least $\sum_{1 \le i < j \le N} |OP(P_i, P_j)|$.

The symmetry property of the metric space can be exploited to form worksets more efficiently as defined below.

Definition 4.1. *Contributing Partitions: Let $\{P_{i=1,\ldots,N}\}$ be the N partitions of D as defined in Definition 3.3. For any pair of partitions, $\langle P_i, P_j \rangle$, let either $OP(P_i, P_j)$ be routed to W_j as outer points, or $OP(P_j, P_i)$ be routed to P_i as outer points, but not both. Define the contributing partitions to P_i as the set $Q_i = \{P_j \mid OP(P_j, P_i)$ is routed to P_i as outer points of $P_i\}$. The worksets are formed such that $I_i = P_i$ and $O_i = \cup_{P_j \in Q_i} OP(P_j, P_i)$.*

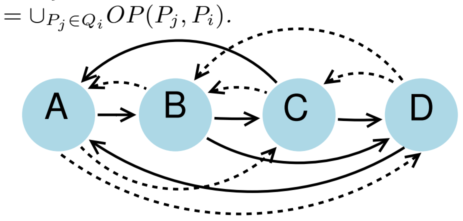


Figure 4: Naïve VS Optimized Routing.

Figure 4 illustrates the inefficiency by showing a 4-way partitioned dataset, where an edge represents $OP(P_i, P_j)$ for partitions P_i and P_j. The dotted edges represent redundant computation. The solid edges incident on any partition represent contributions of outer points from neighboring partitions that do not result in redundant computation.

Theorem 4.2. *The worksets formed according to Definition 4.1 define a non-redundant APSS division where any pair of points is considered in at most one workset.*

Proof. The output $\cup_{i=1}^{N} SimSet(I_i, I_i \cup O_i)$ is equal to $(\cup_{i=1}^{N} SimSet(I_i, I_i)) \cup (\cup_{i=1}^{N} SimSet(I_i, O_i))$. To prove by contradiction, assume two points $p_i \in P_i$ and $p_j \in P_j$ are similar, but not included in the output. We have $P_i \neq P_j$, otherwise, this point pair will be evaluated in $SimSet(P_i, P_i)$. From the symmetry of the space, since p_i and p_j are similar, $p_i \in OP(P_i, P_j)$ and $p_j \in OP(P_j, P_i)$. Without loss of generality, we assume $OP(P_i, P_j)$ is routed to P_j as an outer point, and the pair (p_i, p_j) is evaluated in $SimSet(I_j, O_j)$. Thus, such pair (p_i, p_j) cannot exist. Hence, the worksets define an APSS division. In addition, (p_i, p_j) will either be routed to workset W_i or W_j, but not both. Thus, any pair is only considered in at most one workset. $\square$

One way to construct the non-redundant APSS division is to route $OP(P_i, P_j)$ to P_j if $|P_i| < |P_j|$. This greedy construction minimizes the total computations for forming the APSS division, assuming that the time of evaluating $OP(P_i, P_j)$ is proportional to $|P_i|$. However, this skews the workload, since partitions with larger sizes receive more points. We propose a balancing approach, which is to route $OP(P_i, P_j)$ to P_j if $(((P_i.id + P_j.id)$ is odd) XOR $(P_i.id < P_j.id))$ is true[1]. Hence, each partition gets outer points from roughly half the other partitions, ensuring better load balancing. The SimilarityMapper is outlined in Algorithm 2.

[1]All ids are assumed to be fingerprint-able into integers.

The input of the SimilarityMapper is points rather than partitions. The algorithm first finds the partition P_i to which point p_i belongs (Line 1). In Line 2-6, the algorithm loops over all the partitions. It outputs p_i as an inner point of P_i (Line 3-4). Otherwise, the point is checked for being an outer point of P_j and output if it is (Line 5-6). The output is sorted based on the secondary key ("inner" or "outer"). This ensures the SimilarityReducer processes inner points first to guarantee the correctness of the results in case repartitioning takes place, as discussed in Section 5.

Algorithm 2 SimilarityMapper

Input: p_i: the current point; $PartitionSet$: the set of partitions.
Output: $\{\langle c_j, \langle p_i, PointType \rangle \rangle\}$: a set of mapper output tuples.

```
1: $P_i = GetPartition(p_i, PartitionSet)$
2: for $P_j$ in $PartitionSet$ do
3:     if $P_i$.id == $P_j$.id then
4:         $OutputWithKey(\langle P_i, \text{“inner”}, \langle p_i, \text{“inner”} \rangle \rangle)$
5:     else if $(((P_i.id + P_j.id)$ is $odd)$ XOR $(P_i.id < P_j.id))$
               && $dist(p_i, c_j \le (r_j + t))$ then
6:         $OutputWithKey(\langle P_j, \text{“outer”}, \langle p_i, \text{“outer”} \rangle \rangle)$
```

Compression of Pairs: To design an efficient SimilarityReducer, one major challenge is the online compression of enormous APSS output. This happens when the data is highly clustered given the similarity threshold. Existing graph compression algorithms such as WEBGRAPH [7] are not appropriate because they assume the entire data fits in memory. Our solution, *Community-based Lossless Compression*, compresses similar pairs efficiently on the fly, and provides good Interpretation of the resulting pairs better.

We treat the APSS output as a *graph*, where records are vertices and similar records are connected with edges. Three key community structures are identified – cliques, bicliques and hubs (implemented as adjacency lists), as illustrated in Figure 5. By exploiting the *community structures* in the graph, MR-MAPSS outputs compressed community structures on the fly. The fundamental idea of the compression scheme is to find the community structures when computing the similar pairs, and store them right away. If one record is similar to a set of records, then we say the record and the similar set form an adjacency list. We can thus compress the pairs by factoring the record out. For example, the four similar pairs $[(a, b)(a, c)(a, d)(a, e)]$ can be compressed as $[a, (b, c, d, e)]$. Biclique structure generalizes the adjacency list. We say two sets of records form a biclique if any pair of records, each from a different set, forms a similar pair. For instance, the four similar pairs $[(a, c)(a, d)(b, c)(b, d)]$ can be compressed as $[(a, b), (c, d)]$. To simplify the implementation, cliques are treated as a special bicliques. In terms of compression ratio, adjacency list can compress the data to half of its size, while the biclique representation can compress the data up to $min(m, n)$ times, with m and n being the two set sizes respectively. Clique representation compresses the data to the $O(\sqrt{n})$, where n is the clique size.

Achieving the optimal compression is very difficult - one criteria of optimality is to minimize the number of bicliques collections covering all edges, which is NP-complete [13]. In datasets where no huge cliques and bicliques exist, the ratio is usually between one and two. These compressed structures are identified by sorting the points in their distance to the centroid, and binning them. Then, the bins that have

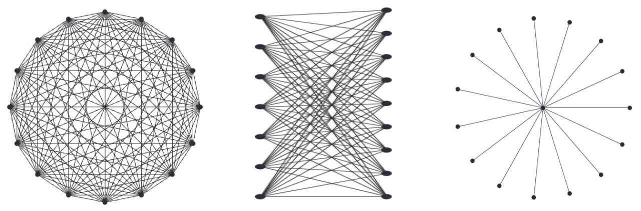


Figure 5: Compression Helps Exhibit the Community Structure

points that fit any of the structures are identified using the Triangle Inequality (TI). Compression results in an order of magnitude difference in the output size when the data is highly clustered given the similarity threshold.

The proposed compression earns us some extra bonus on processing the follow-up steps. For example, shingle clustering [14] and neighborhood density estimation can directly use the compressed data without decompressing, reducing both the IO and the CPU costs.

Similarity Evaluation: The SimilarityReducer employs the online compression approach while generating all similar pairs. Here we assume the whole workset fits in memory. If the workset cannot fit, our streaming solution takes care of it, as discussed in Section 5.

A tree-based indexing structure is built using hierarchical clustering. In our implementation, this creates partitions at different granularities, and makes both pruning and identifying bicliques at different levels easier.

Algorithm 3 EmitSimSet

```
Inputs: $P_A$: index $A$; $P_B$: index $B$.
Outputs: the similar pairs SimSet($P_A$, $P_B$).
 1: if $IsBiClique(P_A, P_B)$ then
 2:     $EmitBiClique(P_A, P_B)$
 3: else if $MayHavePairs(P_A, P_B)$ then
 4:     //Dissimilar partitions are ignored.
 5:     if $P_B.HasChildren()$ then
 6:         for $child in P_B.Children()$ do
 7:             $EmitSimSet(P_A, child)$
 8:     else if $P_A.HasChildren()$ then
 9:         for $child in P_A.Children()$ do
10:             $EmitSimSet(child, P_B)$
11:     else
12:         $EmitAdjLists(P_A, P_B)$
```

In Algorithm 3, EmitSimSet recursively outputs SimSet(P_A, P_B). The output formats are adjacency list and biclique. The algorithm first outputs the two partitions if they can form a biclique. The checking only needs radius and centroid information in the indexing structures (Line 1-2). Next, the algorithm continues only when the two partitions *may* have similar pairs, which prunes away the dissimilar partitions if they have no chance of forming similar pairs. The algorithm exploits the fine-granularity partitioning information by going to the child nodes, and calls itself recursively on one of the child level partitions. If we are already at a leaf node of the indexes, we directly compute and emit the similar pairs (Line 3-12). SimilarityReducer calls EmitSimSet to evaluate similar pairs in each partition.

Algorithm Applicability: Our MR-MAPSS framework can be applied to any metrics, including Earth Mover Distance(EMD), Hamming distances, string edit distances and so on. When computing pairwise distance, it is common practice to make use of filters. For example, with EMD metric, we can employ filters such as weighted L_p norms [4].

It is also worth noting that this generic partitioning framework can even cope with some non-metrics. In fact, both cosine similarity and the KS-statistics can work under our framework. For instance, although cosine similarity itself does not conform to TI, the angles between the vectors actually exhibit the TI property [2].

5. DATA REPARTITIONING

In an imbalanced workload [12], few machines take hours or even days to compute the large worksets while all other machines finish in seconds. It is also possible some worksets are too large to fit in memory, which may make the program crash in the middle of a job. Fortunately, both issues can be greatly relieved by repartitioning the large worksets into even smaller ones iteratively.

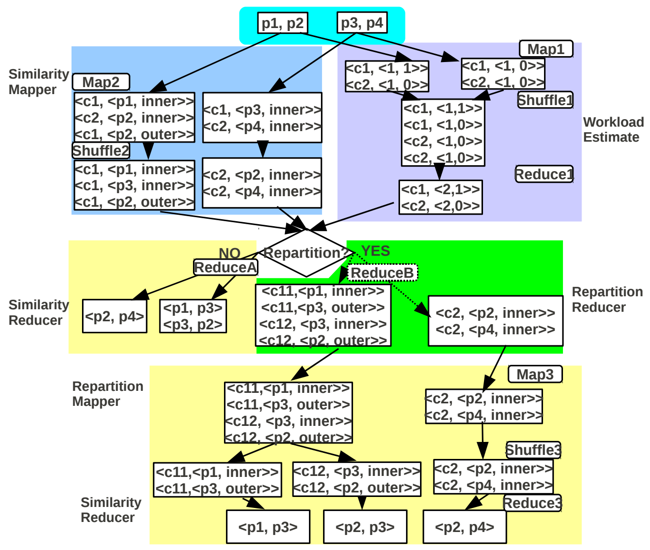
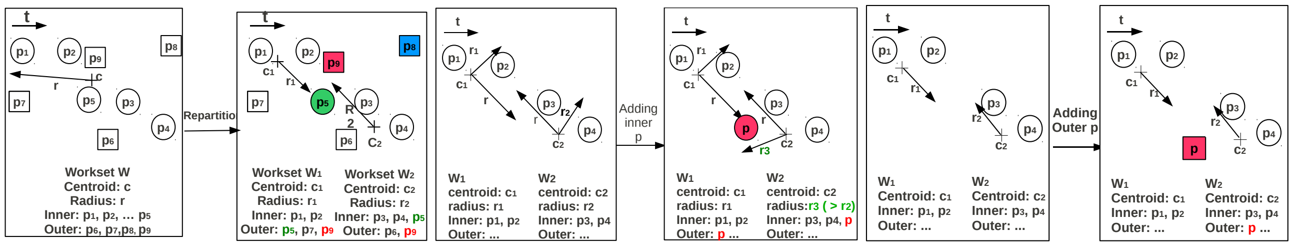


Figure 6: Improved Similarity Computation Task

Figure 6 highlights the more scalable Similarity computation task with repartitioning. The WorkloadEstimate step, Job_1 in Figure 6, computes the inner and outer sizes for each workset. The fact that partition c_1 has 2 inner and 1 outer points is denoted in Figure 6 as $\langle c_1, \langle 2, 1\rangle\rangle$. Then for each workset, it computes a simple estimate of the processing load. It sorts both inner and outer points first, and constructs a frequency histogram based on the distance to the centroid. By applying TI, WorkloadEstimate identifies the bins whose points have to be compared against each other. Hence, WorkloadEstimate can estimate an upper bound on the total number of comparisons. If there exists one partition whose estimated comparison number is larger than ut^2, or whose estimated data size cannot fit in memory, the worksets are repartitioned during the Repartition step. Otherwise, the similar pairs are computed directly.

[2]The value ut was set to 10^8 in our experiments.



(a) Repartitioning, r is the radius of the original worksetInner (b) Inner Streaming (c) Outer Streaming

Figure 7: Illustration of RepartitionReducer

$Map2$ is SimilarityMapper. Based on the estimated workload from WorkloadEstimate, it either continues running SimilarityReducer ($Reduce_A$) to compute all the similar pairs when the data is not highly clustered given the similarity threshold, or switches to the RepartitionReducer ($Reduce_B$) to split the large worksets into smaller ones. WorkloadEstimate and SimilarityMapper can run in parallel until the repartitioning decision is reached.

RepartitionMapper ($Map3$) is simple as it outputs the input as is. It parses the point list and outputs each point individually using the key generated by RepartitionReducer. It also applies a secondary key for the SimilarityReducer to process all the inner points before the outer points.

SimilarityReducer ($Reduce3$) finds the similar pairs and effectively compresses them on the fly. Streaming of data reading can be achieved because the MapReduce infrastructure can stream data from disk when necessary.

When the repartitioning is done, MR-MAPSS checks whether each workset is small enough and repartitions again if there are still some large worksets. We omit this iterative process for simplicity. Next, we study the design of repartitioning, and how it scales even under scarce system memory.

RepartitionReducer: A RepartitionReducer reads inner points before outer ones, due to the use of secondary keys by SimilarityMapper. Each reducer evaluates one workset. If the available memory can accommodate the whole workset and the workload is relatively small, the algorithm outputs all the points in the workset as is, deferring the pair evaluation to the SimilarityReducer. Otherwise, repartitioning is carried out, as will be discussed next.

To repartition the large worksets into smaller ones, the workset's inner points are first divided into multiple "inner sub-partitions" using randomly sampled centriods. Each inner sub-partition then forms a new sub-workset by picking up some outer points, that could be either inner or outer in the original workset. Figure 7a illustrates this concept by splitting the original workset into two smaller ones. The repartitioning operation results in two smaller sub-worksets by duplicating p_5, p_9. Interestingly, p_8 no longer exists in the sub-worksets. The correctness is ensured as $SimSet(I, I \cup O) = SimSet(I_1, I_1 \cup O_1) \cup SimSet(I_2, I_2 \cup O_2)$ holds.

So far, repartitioning assumed all inner points fit in memory. Next, we describe a streaming repartitioning scheme that relaxes this assumption, and prove its correctness.

Streaming Repartition: After identifying the worksets to be repartitioned, the algorithm processes points in a streaming way, thus avoiding crashing if the inner set does not fit in the RepartitionReducer memory. It first randomly samples points from the inner set. Each one of the samples will be a centroid to one of the new sub-worksets .

The streaming operation has two phases - inner point streaming and outer point streaming. If all the inner points fit in memory, calculating the radii of each inner sub-partition is trivial. Otherwise, the exact radii cannot be known until it finishes processing the disk-resident inner points. By adding the old centroid in the centroids list, the radius of the original workset is guaranteed to be an upper bound on all the radii of the new worksets. In this case, the original radius is assigned as the tentative radius when streaming the inner points. The observation on the radii upper bound allows us to avoid outputting the point with every centroid as an outer point. After all inner points are processed, the exact radii are adjusted based on the inner point assignment. This adjustment minimizes the redundancies for the outer points streaming.

Algorithm 4 ProcessInnerPoint

```
input: $workesetInfo$ array, which contains the centroid and the corresponding radius for each point; $point$: the input point to process.
1: $point$ is emitted as inner point of its closest workset $W_i$
2: for $W_j$ in $worksetInfo$ do
3:     $point$ is emitted as outer point of workset $W_j$, iff $dist(point, W_j) \leq R_{W_j} + t$ and $i \neq j$
```

The disk-resident inner (to the original centroid) points are scanned. For each point, the RepartitionReducer emits the point as an inner point to the sub-workset with the closest centroid. It also emits the point as an outer point with all the sub-worksets where the point is within $r + t$ of it centroid, and r is the radius of the original centroid. Figure 7b shows a state change of the worksets when a new inner point arrives after the memory is filled up. Point p is deemed an outer point for W_1 since $dist(p, c_1)$ does not exceed $r + t$.

After all such points are scanned, the radii of the new centroids are calculated. The new radii (instead of the radius of the original centroid) are used when scanning the disk-resident outer (to the original centroid) points and assigning them as outer points to the new centroids.

Processing disk-resident outer points is almost identical to the loop in Algorithm 4. If the distance of the point to any new centroid, W_i, does not exceed $R_{W_i} + t$, it is output as an outer point with W_i. Figure 7c provides an illustration.

THEOREM 5.1. *Streaming Correctness: StreamRepartition splits the workset W into P worksets $W_{1,2,\ldots,P}$, such that $SimSet(W.I, W.I \cup W.O) = \cup_{i=1}^{P} SimSet(I_i, W_i)$ holds.*

PROOF. If the inner points fit in memory, the proof is trivial. We only discuss the other case. Due to the sorting on point type, all the inner points are processed before any outer points. We divide the streaming into three phases. In the first (second) phase, the RepartionReducer scans the inner points of the original workset that fit (do not fit) in memory. In the third phase, it scans the outer points of the original workset. Denote the inner (outer) set for workset W_i by I_{ij} (O_{ij}) after phase j finishes. Denote all the inner (outer) points of W_i as I_i (O_i). Thus, $I_{i1} \subseteq I_i$, $O_{i1} \subseteq O_i$, $W.I = \cup_{i=1}^{P} I_{i2} = \cup_{i=1}^{P} I_i$, and $\cup_{i=1}^{P} O_{i1} \subseteq \cup_{i=1}^{P} O_{i2} \subseteq \cup_{i=1}^{P} O_{i3}$, hold. In the first phase, Algorithm 4 partitions $\cup_{i=1}^{P} I_{i1}$ into P smaller worksets. Also, it outputs $\cup_{i=1}^{P} O_{i1}$. Hence, the output of the first phase is used by SimilarityReducer to compute $\cup_{i=1}^{P} SimSet(I_{i1}, I_{i1})$. Similarly, the output of the second phase is used by SimilarityReducer to compute $\cup_{i=1}^{P} SimSet(I_{i2} - I_{i1}, I_{i2})$. Finally, the output of the third phase is used by SimilarityReducer to compute $\cup_{i=1}^{P} SimSet(I_{i2}, O_{i3})$. By combining the SimilarityReducer computed pairs from the above output, we get $SimSet(W.I, W.I \cup W.O)$. $\square$

The form $\langle worksetID, \langle point, pointType, distance\rangle\rangle$ is used for the output of RepartitionReducer. There could be multiple repartitioning steps, though we find repartitioning once is sufficient, in practice, for solving any scalability issues.

6. EVALUATION

We analyze the algorithm, then explain the experiments setup and results.

6.1 Algorithm Analysis

Given the data D and M machines, assume the data is partitioned with N centroids. We analyze the complexity of each MapReduce job. One notable feature of the framework is that it has minimal redundancy as each pair is considered only once in SimilarityReducer.

The Sampling step computes statistics on every partition. As each machine works independently on roughly $O(|D|/M)$ records, this step is embarrassingly parallel. Usually it could be done in tens of seconds.

The Statistics step computes the centroid statistics. A naïve approach would compare every point with all the centroids, thus having a complexity of $O(K \times |D|/M)$. However, by using some indexing on the centroids, we can significantly reduce the number of comparisons.

In RepartitionMapper, the IO cost takes most of the time. Let I denote the total number of repartitions. In most cases $I = 1$. Basically, the IO cost is $O(I * \sum_{i=1}^{K} |workset_i|)$ as we need to route the worksets to the same Reducer. RepartitionReducer time is linear to the workset size if the number of repartitioned worksets is fixed to a constant(20 in our implementation). Fortunately, most sorting can be done in main memory and the shuffling overlaps with the Mapper. The repartition step reduces the workset size at the expense of extra overhead. This tradeoff is evaluated in Section 6.3.

SimilarityReducer is often the most time-consuming operation in the framework. A naïve implementation would have $\Theta((N/(M * I))^2)$ complexity. However, because a workset is repartitioned when the workload is above the threshold ut, the time complexity is bounded by ut. Usually SimilarityReduer performance is even better given our indexing and compression techniques.

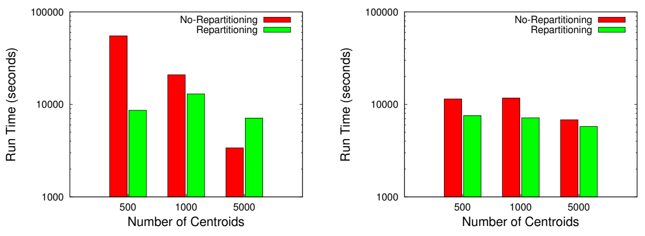
6.2 Experiments Setup

The principal motivation for this work is to find all similar pairs of records in a dataset of network traffic collected at Google. Each record corresponds to the distribution of a signal used for click ring detection[3]. For this evaluation, we used a subset of this dataset of approximately 2M records where each record is a point in a 30-dimensional space. Additionally, we report results on a publicly available dataset from the Netflix competition [23]. This dataset contains 480000 users and their ratings on 17770 movies. Each rating is an integer ranging from 1 to 5. We identified a useful subset, namely the 20 most rated movies (dimensions) and the 421144 users (records) who rated them.

We use these datasets to study five issues. First, we investigate the role of repartitioning as a survival mechanism when the computational resources are constrained. Second, we examine repartitioning as a mechanism to mitigate the *undersampling* problem, where the number of selected centroids is relatively small, and hence MR workers are overloaded. Third, we assess the effectiveness of the compression strategy. Forth, we study the scalability of the MR-MAPSS approach as the number of machines is varied, and finally, we compare our approach with similar efforts.

For all the experiments, a set of default settings is used unless otherwise stated. The MapReduce jobs are run on 500 machines (500 Mappers and 500 Reducers). The pairwise distances are normalized to $[0, 1]$[4]. The distance threshold assumes the values 0.5, 0.2, and 0.05.

6.3 The Effect of Repartitioning



(a) Threshold 0.5 (b) Threshold 0.2

Figure 8: Repartitioning on the Google Dataset

As mentioned above, repartitioning can be autonomously disabled when it is deemed not useful. However, for these experiments, repartitioning is always enabled to study its impact on the overall performance. Figure 8 illustrates this effect when the number of centroids is 500, 1000 and 5000 on the Google dataset. The algorithm with repartitioning is several times faster than the one without repartitioning for almost all configurations. The only exception, under the configuration of 0.5 threshold and 5000 centroids, is because we reached the *oversampling* zone. When oversampling, the number of selected centroids is relatively large, and the overhead of the extra repartitioning MapReduce step does not justify the extra little load balancing. On the other hand, from Figure 9, repartitioning was never useful on the Netflix data. This is because that data was not highly clustered, thus naturally highly load-balanced.

Another major motivation for repartitioning is when the computational resources are limited. An experiment was conducted on the Google dataset with 50 machines, each

[3]Further details of this dataset cannot be disclosed for proprietorial considerations.

[4]Although the current implementation supports Euclidean, Earth Movers Distance and KS-Statistic, we report results using the Euclidean distance. The reason is no significant trend differences were detected between the metrics.

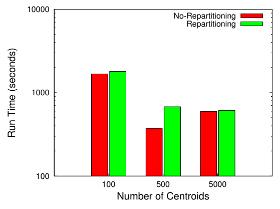


Figure 9: Repartitioning on the Netflix Dataset, $t = 0.05$

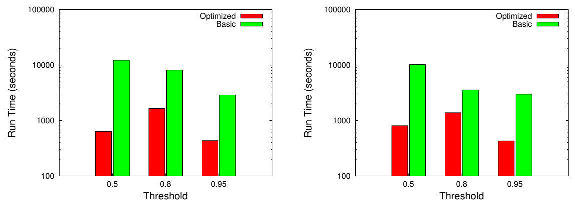
with a 500MB memory limit. When running with 100 centroids, the repartitioning method always outperformed the non-repartitioning counterpart, as shown in Table 2. The non-repartitioning runs were aborted when the threshold was set to 0.5, and failed several times before succeeding when the threshold was 0.2. When running the experiments on larger datasets, the non-repartitioning runs crashed immediately when the memory was constrained to 3GB, while the repartitioning case completed in reasonable time.

Table 2: Run Time (seconds) on the Google Dataset with Limited Memory (500MB)

Threshold	Non-Repartitioning	Repartitioning
0.5	Failure	23864
0.2	69010	28866
0.05	3019	3639

6.4 The Effect of Compression

To appreciate the effectiveness of compression, we designed a naïve algorithm, Basic, which does brute-force computations for each workset, and outputs the similar pairs without compression. For the Google dataset, Basic failed due to the sheer number of pairs. Thus, we only report results on the Netflix dataset in Figure 10. On average, compression resulted in an order of magnitude improvement.



(a) 500 Centroids (b) 1000 Centroids

Figure 10: Compression on the Netflix Dataset

On the Google dataset, we examined the effect of the semantic compression technique by comparing different output sizes as the threshold and the number of centroids, N, were varied (see Figure 11). The output sizes reported here are after MapReduce further compresses the output with bzip2. At 0.5 we have an interesting situation where most of the pairs are similar. So, there was little benefit from pruning, but there was massive benefit from compression. When the threshold was 0.2, many pairs had to be considered, and there was moderate benefit from compression. Notice that the output size when t was 0.5 is almost an order of magnitude smaller than its size when t was 0.2, even though the total output comprises more pairs. At 0.05 there was the least benefit from compression. However, given the threshold is very low, our pruning techniques worked well. It is also worth noting that towards the lower thresholds, as N increases, the compression power is restrained by the smaller workset sizes.

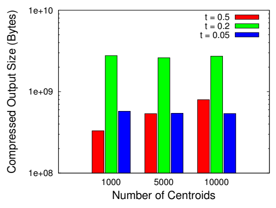
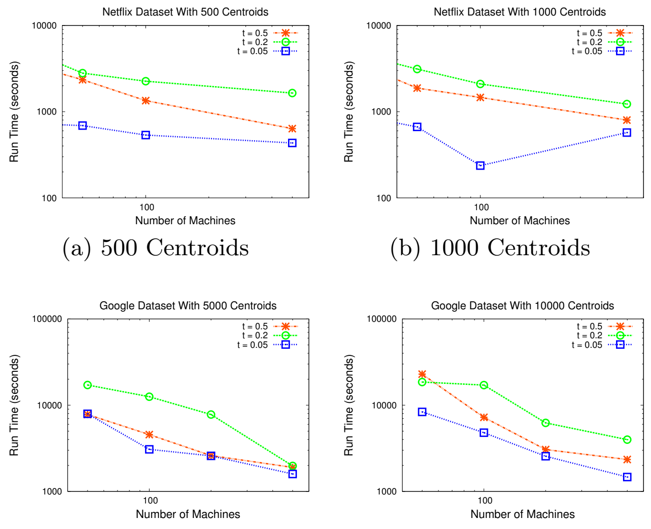


Figure 11: Compression on the Google Dataset

6.5 The Scalability Study

We present the scalability results on both the Netflix and Google datasets in Figures 12. We varied the threshold and the number of centroids, N. We selected 500 and 1000 centroids for the Netflix dataset and 5000 and 10000 centroids for the Google datasets. From Figures 12, for most of the different combinations of parameter values, the MR-MAPSS framework scaled well with the number of machines. The main exception was the case when 1000 centroids were selected for the Netflix dataset when the threshold was 0.05. This is mainly because the scale of Netflix dataset is relatively small. Running with 1000 centroids on 500 machines does not help much in reducing the overall running time but brings in extra system communication and overheads.



(a) 500 Centroids (b) 1000 Centroids

(c) 5000 Centroids (d) 10000 Centroids

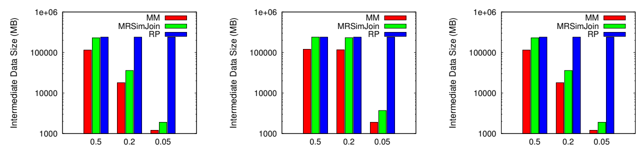
Figure 12: Scalability on the Netflix and Google Datasets

6.6 Comparison with MRSimJoin

We first briefly note that MRSimJoin [28] is distinct from our approach in that it does not address streaming data when input cannot fit in the reducer memory. Moreover, it does not consider compressing the output. The Theta-Join framework [24] enables the comparison of competing strategies when the join problem is: i) **input-size dominated**, when the reducer-input related costs dominate the join completion; ii) **output-size dominated**, when the reducer-output cost outweighs other costs and iii) **input-output balanced**, if neither dominates the other. Since MRSimJoin does not address the output size problem (see Section 6.4), here we focus only on the input-size comparison[5].

We also compare these two approaches with an approach called *record partitioning* (RP). This naïve approach first splits the data by record into P partitions. Each partition is then routed to all other partitions to form worksets and compute similar pairs. Two synthetic datasets and the Netflix dataset are considered for this evaluation. GAUSSIAN and CLUSTERED are both 20 dimensional datasets with 3000000 records. For GAUSSIAN, each element in the record follows normal distribution $N(1, 1)$. For CLUSTERED, each dimension following $1 + 2 \times Uniform(0, 1)$. For ease of comparison, we apply the same indexes proposed by Jacox and Samet [17], which ensures that the total number of evaluations is roughly identical in both methods.

We now consider the size of intermediate data that is fed to the reducers. As shown in Figure 13, on all datasets, our approach consistently reduces the intermediate data size by half even compared with MRSimJoin, resulting in significant gains. The rationale is that MRSimJoin is also an N-Way partitioning model with a different routing scheme. For any two partitions P_i and P_j, it sends both $OP(P_i, P_j)$ and $OP(P_j, P_i)$ to the same reducer. Thus, the amount of intermediate data to the Reduce phase is the same as the naïve approach in Figure 4. Essentially, MR-MAPSS delays about half of the overall evaluations to the Reduce phase, thus generating much fewer candidate pairs. Both MRSimJoin and Record Partitioning do not consider output compression at all (described in Section 4) which results in an additional performance benefit for our method.



(a) GAUSSIAN (b) CLUSTERED (c) Netflix

Figure 13: Reducer Input for MR-MAPSS(MM), MRSimJoin and Record Partitioning (RP). X axis is threshold.

7. RELATED WORK

The set APSS problem, also known as set similarity self-join, has been widely studied in the literatures due to its wide applicability. Several key optimizations, including inverted indexes [26], prefix filtering [8, 6], suffix filtering [32], were proposed. MapReduce versions also exist [10, 5, 22, 31]. Some optimizations in [32] are adopted in a MapReduce setting in [31] for database joins. One recent paper [22] discussed the inefficiencies in [31], and proposed an order of magnitude faster and a more scalable approach. The algorithms in [10, 5] approximate the set and multiset similarity using cosine similarity. Very recently, a partitioning approach [1] has applied cosine similarity on vectors.

The Metric APSS problem has many applications, e.g., recommender systems [19] and Internet-scale image search [25, 20]. Sequential solutions [17] employed effective indexes for metric similarity joins. Silva *et al* [28] demostrated a MapReduce-based metric APSS applications on EC2 by extending the "generalized hyperplane partition" idea. However, they did not consider surviving resource sparsity and output compression.

[5]It is difficult to compare with Theta-Join work as the authors do not mention a specific approach for estimating the join matrix efficiently for similarity self joins.

8. CONCLUSION

In this paper, we proposed the novel MR-MAPSS framework. The backbone of our approach intelligently routes the input data records into independent worksets processed by independent MapReduce workers such that no pair is evaluated twice. We further enhanced the load-balancing of the partitioning algorithm by using workset repartitioning, and employing compression of the output pairs. Finally, we demonstrated the efficiency, the effectiveness and the scalability of the framework using real datasets on hundreds of machines. Future work will study automatic parameter tuning and compare with approaches on Hamming and edit distances.

9. ACKNOWLEDGEMENTS

YW and SP acknowledge support from a Google Research Award and NSF Grant IIS-0917070, and the suggestions from Yang Zhang, Dave Fuhry and Tyler Clemons.

10. REFERENCES

[1] M. Alabduljalil, X. Tang, and T. Yang. Optimizing parallel algorithms for all pairs similarity search. In *WSDM Conference*, pages 203–212, 2013.
[2] D. A. Arbatsky. The Certainty Principle. http://arxiv.org/abs/quant-ph/0608138v1, 2006.
[3] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *SODA*, pages 1027–1035, 2007.
[4] I. Assent, A. Wenning, and T. Seidl. Approximation Techniques for Indexing the Earth Mover's Distance in Multimedia Databases. In *ICDE Conference*, pages 11–11, 2006.
[5] R. Baraglia, G. De Francisci Morales, and C. Lucchese. Document Similarity Self-Join with MapReduce. In *ICDM Conference*, pages 731–736, 2010.
[6] R. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. In *WWW Conference*, pages 131–140, 2007.
[7] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *WWW Conference*, pages 595–601, 2004.
[8] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE Conference*, pages 5–5, 2006.
[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.
[10] T. Elsayed, J. Lin, and D. Oard. Pairwise Document Similarity in Large Collections with MapReduce. In *ACL (Short Papers)*, pages 265–268, 2008.
[11] R. et. al. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, pages 262–270, 2012.
[12] R. Ferreira Cordeiro, C. Traina Junior, A. Machado Traina, J. López, U. Kang, and C. Faloutsos. Clustering Very Large Multi-Dimensional Datasets with MapReduce. In *SIGKDD Conference*, pages 690–698, 2011.
[13] P. C. Fishburn and P. L. Hammer. Bipartite dimensions and bipartite degrees of graphs. *Discrete Mathematics*, 160(1):127–148, 1996.
[14] D. Gibson, R. Kumar, and A. Tomkins. Discovering Large Dense Subgraphs in Massive Graphs. In *VLDB Conference*, pages 721–732, 2005.
[15] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.
[16] M. Henzinger. Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms. In *SIGIR Conference*, pages 284–291, 2006.
[17] E. Jacox and H. Samet. Metric space similarity joins. *TODS*, 33(2):7, 2008.
[18] R. Jarvis and E. Patrick. Clustering Using a Similarity Measure Based on Shared Near Neighbors. *TOC*, 100(11):1025–1034, 1973.
[19] Y. Koren. Collaborative Filtering with Temporal Dynamics. *CACM*, 53(4):89–97, 2010.
[20] B. Kulis and K. Grauman. Kernelized Locality-Sensitive Hashing for Scalable Image Search. In *ICCV*, pages 2130–2137, 2009.
[21] A. Metwally, D. Agrawal, and A. El Abbadi. DETECTIVES: DETEcting Coalition hiT InSSation attacks in adVertising nEtworks Streams. In *WWW Conference*, pages 241–250, 2007.
[22] A. Metwally and C. Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.
[23] Netflix Inc. Netflix competition.
[24] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD Conference*, pages 949–960, 2011.
[25] Y. Rubner, C. Tomasi, and L. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *IJCV*, 40(2):99–121, 2000.
[26] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.
[27] V. Satuluri and S. Parthasarathy. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *PVLDB*, 5(5):430–441, 2011.
[28] Y. Silva and J. Reed. Exploiting mapreduce-based similarity joins. In *SIGMOD*, pages 693–696, 2012.
[29] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating Similarity Measures: A Large-Scale Study in the Orkut Social Network. In *SIGKDD Conference*, pages 678–684, 2005.
[30] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information processing letters*, 40(4):175–179, 1991.
[31] R. Vernica, M. Carey, and C. Li. Efficient Parallel Set-Similarity Joins Using MapReduce. In *SIGMOD Conference*, pages 495–506, 2010.
[32] C. Xiao, W. Wang, X. Lin, and J. Yu. Efficient Similarity Joins for Near Duplicate Detection. In *WWW Conference*, pages 131–140, 2008.