Massive Computations of Online Social Neighborhoods from Mobile Big Data

Georgios Chatzimilioudis and Constantinos Costa and Demetrios Zeinalipour-Yazti and Wang-Chien Lee and Evaggelia Pitoura

Abstract In the age of smart urban and always-connected mobile environments, the mobile crowd offers an unprecedented source for computational paradigms motivated by living organisms. Computationally exploiting the mobility intelligence inherent in such a crowd has the potential of enhancing science and services, e.g., social network, emergency and crisis management services. The data generated and consumed by the mobile crowd is usually massive (Volume) and heterogeneous (Variety), which characterize Big Data. This chapter studies how the Big Data generated from the mobile crowd can be efficiently used to compute online social neighborhoods that evolve as the crowd moves. These neighborhoods can be defined as a kNN graph that needs to be computed over the arriving big spatial data every few seconds (Velocity), which is another characteristic of Big Data. Particularly, it analyzes models and metrics that are the foundation of efficient distributed computations for kNN graphs in Big Data scenarios, and underlines that intelligent data space partitioning and data replication among the processing units are imperative best design practices towards this goal. It goes on with a detailed survey on the state-of-the-art distributed solutions provided in literature so far. A detailed presentation is provided for one of these algorithms, namely *Spitfire*, together with an experimental evaluation and a case study of its application in a dynamic social network, called *Rayzit*, which studies social behavior.

Key words: Location, Mobile, Social, kNN, Big Data, Anonymous, Distributed, Parallel, Algorithms, Crowd Networks

Demetrios Zeinalipour-Yazti (Corresponding Author), Department of Computer Science, University of Cyprus, Email: dzeina@cs.ucy.ac.cy, Tel: +357-22-892755, Fax: +357-22-892701, Address: 1 University Avenue, P.O. Box 20537, 1678 Nicosia, Cyprus; G. Chatzimilioudis, C. Costa, University of Cyprus, 1678 Nicosia, Cyprus; W.-C. Lee, Penn State University, University Park, PA 16802, USA; E. Pitoura, University of Ioannina, 45110 Ioannina, Greece.

1 Introduction

The proliferation of communication and positioning technologies in today's always-connected mobile environment, in combination with the trend to share such information, lead to an explosion of high *velocity* streams of geo-tagged data [11] that include a great *variety* of information. The increasing frequency and detail of information mandate efficient management of the collected data, as its sheer *volume* (i.e., millions of users world-wide) can not be processed in due time using conventional solutions. Based on these characteristics, namely *velocity*, *variety*, and *volume*, such data is classified as Big Data.

Computational Intelligence techniques [59, 54, 6] such as fuzzy logic, evolutionary computation, neurocomputing and other machine learning techniques provide us with complementary searching and reasoning means to bring forward solutions to the Big Data challenges. Neural networks [40], cluster analysis [1], decision trees [14], evolutionary computing [16] and neuro-fuzzy systems can be applied for data reduction and projection when dealing with Big Data. Neuro-fuzzy methods [43], Genetic Algorithms [32, 49], and its combination with neural learning [2] and fuzzy clustering [55, 3] can be applied for discovering patterns in Big Data. Finally, swarm intelligence has been applied in creating a large-scale distributed storage and reasoning system [41]

The connected mobile crowd can be seen as a dynamic living organism. Considering the emergence of location-aware services and applications, it is evident that building computational paradigms based on the "intelligence" that this organism inherits from its mobile atoms and their position relative to each other is of great value, similar to swarm intelligence [24]. Social neighborhoods and spatiotemporal pattern discovery and modeling are key to many research areas, e.g., brain studies [31], UAV formation [25], robot navigation [35], engineering design [52], urban planning [34], and of course social mobility [51, 23] and nearest neighbor techniques [58].

A straightforward example is the computation of a dynamic network that evolves based on the atom's proximity, i.e., *social neighborhood*. Such a network can be described as a kNN graph, which is nothing more than a Continuous All k Nearest Neighbors (AkNN) query over millions of moving atoms. An application of such a biologically motivated computational paradigm can be found in cases of overloaded or partially broken (i.e., non-operational) mobile networks, where it is often important to enable communication over a kNN overlay graph that allows the crowd to connect to its geographically closest peers, those that can physically interact with the user and respond to an emergency crowdsourcing task, such as seeing/sensing similar things as the user (e.g., collect video, photos, etc.) [9][18].

Another application scenario based on dynamic social neighborhoods is *Rayzit* [18][1], an award-winning, dynamic and anonymous crowd messaging architecture, that has been developed to connect worldwide users instantly to their *k Nearest Neighbors (kNN)* as they move in space (see Figure 1.1). Similar to other social network applications (e.g., Twitter, Facebook), scalability is key in making *Rayzit* functional

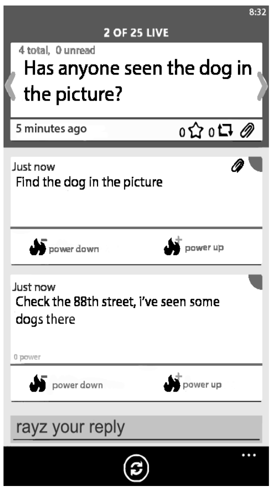
[1] Rayzit: `http://rayzit.com/`

and operational. Therefore we are challenged with the necessity to perform a fast computation of an AkNN query every few seconds in a scalable architecture.

The *k Nearest Neighbor (kNN)* search is one of the simplest non-parametric machine learning approaches mainly used for classification [19] and regression [5]. kNN aims at finding k objects that are the most similar to another object. Extensions of kNN include the Condensed nearest neighbor algorithm that reduces the data set for kNN classification [29] and the fuzzy-kNN [56] that deals with uneven and dense training datasets.

Most existing techniques for computing AkNN queries are centralized, lacking both scalability and efficiency [12, 13], which are mandatory when dealing with Big Data. The wide availability of off-the-shelf, shared-nothing, cloud infrastructures brings a natural framework to cope with Big Data in processing AkNN queries. Only recently researchers have proposed algorithms for optimizing AkNN queries in such infrastructures.

Recent distributed techniques achieve scalability using shared-nothing cloud infrastructures. However, they are inefficient due to sub-optimal data space partitioning and data replication among processing units. In such distributed environments, the communication cost is the main bottleneck and load balancing is the main speed-up factor. These issues need to be addressed thoroughly in order to exploit the computational resources to the fullest. The lower bound communication cost in this case is achieved when the total input of the servers is the initial data without overlapping instances. Note that synchronization, handshake, and/or header data are considered negligible in Big Data environments. The primary communication cost, also termed as overhead, comes from intermediate results that need to be exchanged between servers or from objects of the initial data that need to be input in more than one



Has anyone seen the dog in the picture?

Find the dog in the picture

Check the 88th street, I've seen some dogs there

Fig. 1 Our Rayzit crowd messenger enabling users to interact with their k geographic Nearest Neighbors.

server. This overlap dictates the scalability of such a distributed system and is the main issue to be addressed by solutions that optimize queries on Big Data.

This chapter analyzes models and metrics that are the foundation of efficient distributed solutions for AkNN queries in Big Data scenarios. It goes on with a detailed study on the distributed solutions provided for k Nearest Neighbor queries (kNN). The chapter then follows with the description and analysis of *Spitfire* [10], a novel, parallel, shared-nothing algorithm that provides a scalable and high-performance AkNN processing engine for Big Data employed on a cloud infrastructure. *Spitfire* deploys a fast *load-balanced* partitioning scheme along with an *efficient replication-set* selection algorithm, to provide fast main-memory computations of the exact AkNN results in a batch-oriented manner. It encapsulates a number of innovative internal components that make optimal use of the resource network: (i) a novel linear-time partitioning algorithm that achieves sufficient load-balancing independent of data skewness, (ii) a new replication algorithm that exploits geometric properties towards minimizing the candidates to be exchanged between servers, and (iii) optimizations added to the local AkNN computation proposed in [13].

Spitfire is particularly useful to large-scale main-memory data processing platforms (e.g., Apache Spark), which have not realized AkNN operators to this date. Therefore, *Spitfire* allows messaging platforms, like Rayzit.com, to scale to millions of active users around the world, dealing with both the *volume* and the *velocity* of this Big Data. This chapter also presents the backend architecture of Rayzit as a cloud-like infrastructure that implements algorithms like *Spitfire* in real-world scenarios.

The remainder of this chapter is organized as follows. Section 1.2 provides the problem definition, system model and desiderata, as well as an overview of the related work on distributed AkNN query processing. Section 1.3 presents the *Spitfire* algorithm with a particular emphasis on its partitioning and replication strategies. It discusses possible extensions to support more gracefully higher-rate AkNN scenarios with streaming data, as well as AkNN queries over high-dimensional data, approximate AkNN solutions, and online geographic hashing techniques at the network load-balancing level to speed up the distribution process.

2 Background and Related Work

This section formalizes the problem, describes the general principles needed for efficiency, and overviews existing research on distributed algorithms for computing AkNN queries. Such solutions can be categorized as “bottom-up” or “top-down” approaches. We shall express the AkNN query as a kNN Self-Join introduced earlier. Our main notation is summarized in Table 1.1.

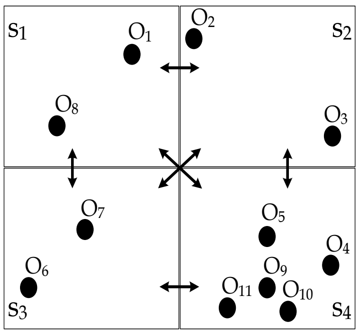


Fig. 2 Distributed main-memory AkNN computation in Rayzit is enabled through the *Spitfire* algorithm.

2.1 Goal and Design Principles

In this section we outline the desiderata and design principles for efficient distributed AkNN computation.

Definition 1. Problem Statement. Given a set of objects O in a bounding area A and a cloud computing infrastructure S, compute the AkNN result of O using S, maximizing *performance*, *scalability* and *load balancing*.

Formally, the kNN of an object o from some dataset O, denoted as $kNN(o, O)$, are the k objects that have the most similar attributes to o [53]. Specifically, given objects $o_a \neq o_b \neq o_c$, $\forall o_b \in kNN(o_a, O)$ and $\forall o_c \in O - kNN(o_a, O)$ it always holds that $dist(o_a, o_b) \leq dist(o_a, o_c)$[2]. An *All kNN (AkNN)* query generates a kNN graph. It computes the $kNN(o, O)$ result for every $o \in O$ and has a quadratic worst-case bound. An AkNN query can alternatively be viewed as a *kNN Self-Join*: *Given a dataset O and an integer k, the kNN Self-Join of O combines each object* $o_a \in O$ *with its k nearest neighbors from O, i.e.,* $O \bowtie_{kNN} O = \{(o_a, o_b) | o_a, o_b \in O$ *and* $o_b \in kNN(o_a, O)\}$.

Solving the AkNN problem efficiently in a distributed fashion requires the object set O be partitioned into disjoint subsets O_i corresponding to m servers (i.e., $O = \bigcup_{1 \leq i \leq m} O_i$). To facilitate local computations on each server and ensure correctness of the global AkNN result, servers need to compute distances across borders for the objects that lie on opposite sides of the border and are close enough to each other. Consider the example illustrated in Figure 1.2, where 11 objects are partitioned over 4 spatial quadrants, each being processed by one of four servers $\{s_1 \ldots s_4\}$. Now assume that we are interested in deriving the 2NN for each object $\{o_1 \ldots o_{11}\}$. By visually examining the example, we can identify that the $2NN(o_1, O)$ are $\{o_2, o_8\}$. Although o_8 indeed resides along with o_1 on s_1, the same does not apply to o_2, which resides on s_2. Consequently, in order to calculate $dist(o_1, o_2)$, we will first

[2] In our discussion, *dist* can be any L_p-norm distance metric, such as Manhattan (L_1), Euclidean (L_2) or Chebyshev (L_∞).

need to transfer o_2 from s_2 to s_1. The same problem also applies to other objects (e.g., $2NN(o_8, O) = \{o_7, o_1\}$ and $2NN(o_6, O) = \{o_7, o_8\}$).

In any performance-driven distributed algorithm, the efficiency is determined predominantly by the network messaging cost (i.e., network I/O). Therefore, in this work we address the *problem of minimizing the number of objects transferred (replicated) between servers during the computation of the AkNN query*.

Another factor in a distributed system is balancing the workload assigned to each computing node s_i, such that each s_i will require approximately the same time to compute the distances among objects. By examining Figure 1.2, we can see that s_4 would require to compute 15 distances among 6 objects (i.e., local objects $\{o_4, o_5, o_9, o_{10}, o_{11}\}$ and transferred object $\{o_3\}$), while s_3 would need to compute only 3 distances among 3 objects (i.e., local objects $\{o_6, o_7\}$ and transferred object $\{o_8\}$). This asymmetry, means that s_3 will complete 5 times faster than s_4. In fact, s_4 lies on the critical path of the computation as it has the highest load among all servers. Consequently, we also need to address the *problem of quickly deriving a fair partitioning of objects between s_i that would yield a load-balanced execution and thus minimize synchronization time*.

Next we list the desiderata that any AkNN solution should have.

Performance: This can be measured by *Response Time* representing the actual time required by a distributed AkNN algorithm to compute its result. In a distributed system the main bottleneck for the response time is the communication cost, which is affected by the size of the input dataset for each server. Synchronization, handshake, and/or header data are considered negligible in such environments [4]. Therefore, the lower bound of the communication cost is achieved when the total input of the servers equals to the size of the initial data set O. However, additional communication cost is incurred when some objects need to be transmitted (replicated) to more than one server. Thus, the input is augmented with a number of replicated objects, which is denoted as *replication factor f*.

Scalability: This can be measured by the *Replication Factor (f)* representing the number of times the n objects are replicated between servers to guarantee correctness of the AkNN computation. To accommodate the growth of data in volume, an efficient data processing algorithm should exploit the computing power of as many workers as possible. Unfortunately, increasing the number of workers usually comes with an increased communication cost. A scalable solution would require that the *replication factor f* increases slower than the *performance gain* with respect to the number of servers.

Load Balancing: This can be measured by the *Standard Deviation* of the number of objects received in servers. To fully exploit the computational power of all servers and minimize response time, an efficient algorithm needs to distribute work load equally among servers. In the worst case, a single server may receive the whole load, making the algorithm slower than its centralized counterpart. The work load is determined by the number of objects that are assigned to a server. Therefore, load balancing is achieved when the object set is partitioned equally.

Table 1 Summary of Notation

Notation	**Description**
o, O, n	Object o, set of all o, $n = \|O\|$
s_i, S, m	Server s_i, set of all s_i, $m = \|S\|$
$kNN(o, O)$	k nearest neighbors of o in O
$dist(o_a, o_b)$	L_p-norm distance between o_a and o_b
A, A_i, O_i	Area, Sub-Area i, Objects in sub-area i
b, B_i	A border edge of A_i, set of all $b \in A_i$
Adj_i	Set of all A_j adjacent (sharing b) to A_i
EC_i	External Candidates of A_i

2.2 *Parallel AkNN Algorithms*

There is a significant amount of previous work in the field of computational geometry, where parallel AkNN algorithms for special multi-processor or coarse-gained multicomputer systems are proposed. The algorithm proposed in [8] uses a quad-tree and the well-separated pair decomposition to answer an AkNN query in $O(\log n)$ using $O(n)$ processors on a standard CREW PRAM shared-memory model. Similarly, [21] proposes an algorithm with time complexity $O(n \cdot \log\frac{n}{m} + t(n,m))$, where n is the number of points in the data set, m is the number of processors, and $t(n,m)$ is the time for a global-sort operation. Nevertheless, none of the above algorithms is suitable for a shared-nothing cloud architecture, mainly due to the higher communication cost inherent in the latter architectures.

2.3 *Distributed AkNN Algorithms: Bottom-Up*

The first category of related work on distributed solutions solve the AkNN problem bottom-up by applying existing kNN techniques (e.g., iterative deepening from the query point [65]) to find the kNN for each point separately. The authors in [48] propose a general distributed framework for answering AkNN queries. This framework uses any centralized kNN technique as a black box. It determines how data will be initially distributed and schedules asynchronous communication between servers whenever a kNN search reaches a server border. In [45], the authors build on the same idea, but optimize the initial partitioning of the points onto servers and the number of communication rounds needed between the servers. Nevertheless, it has been shown in [13] that answering a kNN query for each object separately restricts possible optimizations that arise when searching for kNNs for a group of objects that are in close proximity.

2.4 Distributed AkNN Algorithms: Top-Down

The second category of related work on distributed solutions solve the AkNN problem top-down by first *partitioning* the object set into subsets and then computing kNN *candidates* for each area in a process we call *replication*. These batch-oriented algorithms are directly comparable to our proposed solution, therefore we have summarized their theoretical performance in Table 1.2. All existing algorithms in this category happen to be implemented in the MapReduce framework, therefore we overview basic MapReduce concepts before we describe these algorithms.

Background: *MapReduce [20] (MR)* is a well established programming model for processing large-scale data sets with commodity shared-nothing clusters. Programs written in MapReduce can automatically be parallelized using a reference implementation, such as the open source Hadoop framework[3], while cluster management is taken care of by YARN or Mesos [30]. The Hadoop MapReduce implementation allows programmers to express their query through `map` and `reduce` functions, respectively. For clarity, we refer to the execution of these MapReduce functions as *tasks* and their combination as a *job*. For ease of presentation, we adopt the notation *MR#.map* and *MR#.reduce* to denote the tasks of MapReduce job number #, respectively. Main-memory computations in Hadoop can be enforced using in-memory file systems such as Tachyon [57].

Hadoop Naive kNN Join (*H-NJ* [39]). This algorithm is implemented with 1 MapReduce job. In the map task, O is transferred to all m servers triggering the reduce task that initiates the nested-loop computation $O_i \ltimes_{kNN} O$ (O_i contains n/m objects logically partitioned to the given server). *H-NJ* incurs a heavy $O(\frac{n^2}{m})$ processing cost on each worker during the reduce step, which needs to compute the distances of O_i to O members. It also incurs a heavy $O(mn)$ communication cost, given that each server receives the complete O. The replication factor achieved is $f_{\text{H-NJ}} = m$.

Hadoop Block Nested Loop kNN Join (*H-BNLJ* [64]). This algorithm is implemented with 2 MapReduce jobs, MR1 and MR2, as follows: In MR1.map, O is partitioned into $\sqrt{m}$ disjoint sets, creating m possible pairs of sets in form of (O_i, O_j), where $i, j \leq \sqrt{m}$. Each of the m pairs (O_i, O_j) is sent to one of the m servers. The communication cost for this action is $O(\sqrt{m}n)$, attributed to the replication of m pairs each of size $\frac{n}{\sqrt{m}}$. The objective of the subsequent MR1.reduce task is to allow each of the m servers to derive the "local" kNN results for each of its assigned objects. Particularly, each s_i performs a local block nested loop kNN join $O_i \ltimes_{kNN} O_j$. The results of MR1.reduce have to go through a MR2 job, in order to yield a "global" kNN result per object. Particularly, MR2.map hashes the possible $\sqrt{m}$ kNN results of an object to the same server. Finally, MR2.reduce derives the global kNN for each object using a local top-k filtering. The CPU cost of *H-BNLJ* is $O(\frac{n^2}{m})$, as each server performs a nested loop in MR1.reduce. The replication factor achieved is $f_{\text{H-BNLJ}} = 2\sqrt{m}$.

[3] Apache Hadoop. `http://hadoop.apache.org/`

Table 2 Algorithms for Distributed Main-Memory AkNN Queries
[n: objects | m: servers | f: replication factor | $f << m < n$]

Algorithm	**Preproc.**	**Part. & Repl.**	**Refinement**	**Communic.**
H-NJ [39]	-	$O(n)$	$O(\frac{n^2}{m})$	$O(mn)$
H-BNLJ [64]	-	$O(n)$	$O(\frac{n^2}{m})$	$O(\sqrt{m}n)$
H-BRJ [64]	-	$O(n)$	$O(\frac{n}{\sqrt{m}}\log\frac{n}{\sqrt{m}})$	$O(\sqrt{m}n)$
PGBJ [39]	$O(\sqrt{n})$	$O(n^{1.5}/m)$	$O(f_{\text{PGBJ}}\frac{n^2}{m^2})$	$O(f_{\text{PGBJ}}n)$
Spitfire	-	$O(n)$	$O(f_{\textit{Spitfire}}\frac{n^2}{m^2})$	$O(f_{\textit{Spitfire}}n)$

Hadoop Block R-tree Loop kNN Join (*H-BRJ* [64]). This is similar to *H-BNLJ*, with the difference that an R-tree on the smaller O_i set is built prior to the MR1.reduce task, to alleviate its heavy processing cost shown above. This reduces the join processing cost during MR1.reduce to $O(\frac{n}{\sqrt{m}}\log\frac{n}{\sqrt{m}})$. The communication cost remains $O(\sqrt{m}n)$ and the incurred replication factor is again $f_{\text{H-BRJ}} = 2\sqrt{m}$.

Hadoop Partitioned Grouped Block kNN Join (*PGBJ* [39]): This is the state-of-the-art Hadoop-based AkNN query processing algorithm that is implemented with 2 MapReduce jobs, MR1$'$ and MR2$'$, and 1 pre-processing step according to the following logic: in a preprocessing step, a set of approximately $\sqrt{n}$ random pivots in space is generated [39].

During MR1$'$.map, each object in O is mapped to its closest pivot, thus partitioning O into $\sqrt{n}$ sets (i.e., $O=\bigcup_{1\leq j\leq\sqrt{n}} O_j$). This takes $O(n^{\frac{3}{2}}/m)$ time, since on each server the distance of n/m objects is measured against $\sqrt{n}$ pivots. At the same time, the maximum and minimum distances between O_j objects and its corresponding pivot are recorded as lower and upper pruning thresholds for subsequent filtering. In MR2$'$.map, the given bounds define a set of objects around each partition O_j that must be replicated to O_j (coined F_j). MR1$'$.reduce in PGBJ is void.

During MR2$'$.map, the $\sqrt{n}$ subsets defined during MR1$'$.map are geographically grouped together into m clusters (i.e., $O=\bigcup_{1\leq i\leq m} O_i$) using a grouping strategy, which greedily attempts to generate clusters of equal population around some m geometrically dispersed pivots. For each generated cluster O_i, a set F_i is derived based on the union of the respective F_j sets of the cluster defined earlier. Having F_i defined, it allows MR2$'$.reduce to perform a straightforward $O_i \bowtie_{kNN} (O_i \cup F_i)$ to generate the final result.

The replication factor is $f_{\text{PGBJ}}=\frac{1}{n}\sum_{i=1}^{m}|F_i| + 1$. The CPU cost is $O(\sqrt{n})$ for the preprocessing step, O($\frac{n}{m}\sqrt{n}$) for MR1$'$ and $O(f_{\text{PGBJ}}\frac{n^2}{m^2})$ in MR2$'$. PGBJ only distributes O over m servers and then exchanges $f_{\text{PGBJ}}n$ candidates between servers, therefore its communication cost is $O(f_{\text{PGBJ}}n)$.

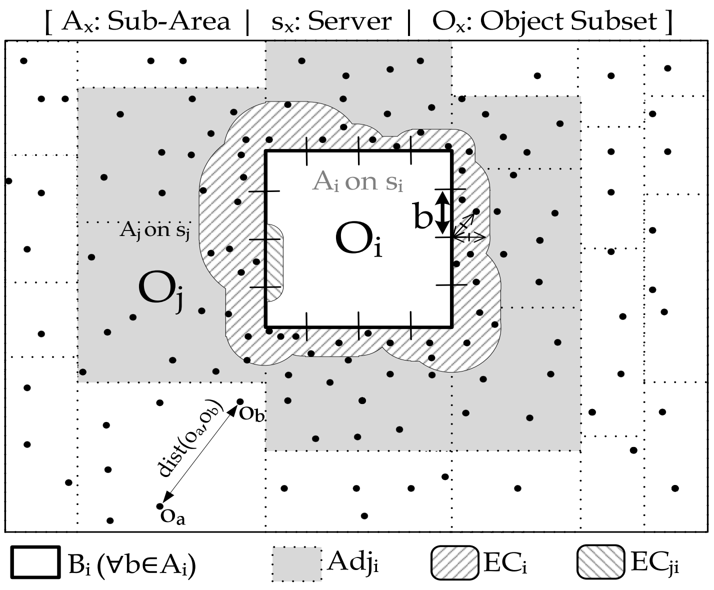


Fig. 3 *Spitfire* Overview: i) Space partitioning to equi-depth quadrants; ii) Replication between neighboring O_i and O_j using EC_{ji} and EC_{ij}, respectively; and iii) Local refinement within each $O_i \cup EC_i$.

3 The Spitfire Algorithm

In this section we propose *Spitfire*, a high-performance distributed main-memory algorithm. We outline its operation and intrinsic characteristics and then detail its three internal steps that capture the core functionality, namely *partition* (Partitioning/Splitting), *computeECB* (Replication) and *localAkNN* (Refinement). The name *Spitfire* is derived by a syllable play using the meaning of these three steps, namely **Split**, Re**fi**ne **re**plicate, which implies good mechanical performance.

3.1 Spitfire: Overview and Highlights

As shown in [13], grouping the points geographically and computing common kNN candidates per group, instead of computing the kNN for each point separately, significantly improves performance. Furthermore, partitioning is necessary for distribution, thus such algorithms, which geographically group their points, inherently lend themselves as distributed solutions. For the above reasons, the solution we propose in this work also belongs to the category of "top-down" distributed AkNN solutions.

Overview: In *Spitfire*, the input O is processed in a single communication round, involving the three discrete steps shown in Algorithm 1 and explained below (please see Figure 1.3 along with the description):

- **Step 1 (Partitioning):** Initially, O is partitioned into (disjoint) sub-areas with an approximately equal number of objects, i.e., $O = \bigcup_{1 \leq i \leq m} O_i$. We use a simple but fast centralized hash-based adaptation of equi-depth histograms [42] to achieve good load balancing in $O(n + \sqrt{nm})$ time. It first hashes the objects based on their locations into a number of sorted equi-width buckets on each axis and then partitions each axis sequentially by grouping these buckets in an equi-depth fashion. Let each O_i belong to area A_i handled by server s_i, B_i be the border segmentations surrounding A_i and Adj_i be the adjacent servers to s_i (handling adjacent areas to A_i).
- **Step 2 (Replication):** Subsequently, each s_i computes a subset of O_i, coined *External Candidates* EC_{ji}, which is possibly needed by its neighboring s_j for carrying out a local AkNN computation in Step 3 (refinement). The given set EC_{ji} is transmitted by s_i to s_j (i.e., left-dashed area within O_i, as depicted in Figure 1.3). Since each s_j applies the above operation as well, we also have the notion of EC_{ij}. The union of EC_{ij} for all neighboring $s_j \in Adj_i$ defines the External Candidates of O_i, i.e., $EC_i = \bigcup_{1 \leq j \leq Adj_i} EC_{ij}$. The cardinality of all EC_i defines the *Spitfire replication factor*, i.e.,

$$f_{Spitfire} = \frac{1}{n} \sum_{i=1}^{m} |EC_i| + 1 \tag{1}$$

- **Step 3 (Refinement):** Finally, each s_i performs a local $O_i \bowtie_{kNN} (O_i \cup EC_i)$ computation, which is optimized by using a heap structure along with internal geographic grouping and bulk processing.

The CPU time complexity of *Spitfire* is $T(n) = n + \sqrt{nm} + f_{Spitfire}\frac{n^2}{m^2}$, as this will be shown in Sections 1.3.2 and 1.3.4, respectively. The communication complexity is expressed as a function of the total number of objects communicated over the network, i.e., $C(n) = f_{Spitfire}n$.

Highlights: *Spitfire*'s main advantages to prior work follow:

- **Fast Batch Processing:** *Spitfire* is suitable for *online operational* AkNN workloads as opposed to *offline analytic* AkNN workloads. Particularly, it is able to compute the AkNN result-set every few seconds as opposed to minutes required by state-of-the-art AkNN algorithms configured in main-memory.
- **Effective Pruning:** *Spitfire* uses a pruning strategy that achieves replication factor $f_{Spitfire}$, which is shown analytically and experimentally to be always better than that achieved by state-of-the-art AkNN algorithms.
- **Single Round:** *Spitfire* is a single round algorithm as opposed to state-of-the-art AkNN algorithms that require multiple rounds.

Algorithm 1 - *Spitfire* Distributed AkNN Algorithm

```
Input: $n$ Objects $O$ in Area $A$, $m$ Servers $S$, Parameter $k$
Output: AkNN of $O$

    ▷ Step 1: Partitioning (centrally)
 1: $Areas = partition(A, m)$                                  ▷ (Algo. 2, Sec. 1.3.2)
 2: for all $A_i \in Areas$ do
 3:     determine $O_i, Adj_i$ and $B_i$
 4:     transmit $O_i$ to server $s_i \in S$
 5: end for
    ▷ Step 2: Replication (parallel on each $s_i$)
 6: for all $b \in B_i$ do                          ▷ Find candidates needed by each $A_j$
 7:     $EC_b$ =computeECB($b$,$O_i$)                           ▷ (Algo. 3, Sec. 1.3.3)
 8:     $EC_{ji}$=$EC_{ji} \cup EC_b$                           ▷ Append to $EC_{ji}$ results.
 9: end for
10: for all $A_j \in Adj_i$ do                          ▷ Exchange External Candidates
11:     Asynchronous send $EC_{ji}$ to adjacent server $s_j$
12:     Asynchronous receive $EC_{ij}$ from adjacent server $s_j$
13:     $EC_i = EC_i \cup EC_{ij}$                              ▷ Append to $EC_i$ results.
14: end for
▷ Step 3: Refinement (parallel on each $s_i$)
15: localAkNN($O_i$,$EC_i$)                                    ▷ (Algo. 4, Sec. 1.3.4)
```

3.2 Step 1: Partitioning

To fully utilize the processing power of the available servers in the cluster, it is desirable to allocate an evenly balanced workload. This functionality has to be carried out in a fast batch-oriented manner, in order to accommodate the real-time nature of crowd messaging services such as those offered by Rayzit. Particularly, our execution has to be carried out every few seconds. As the partitioning is to be used by each AkNN query, the result will not be useful after a few seconds (i.e., when the next AkNN query is executed). We consequently have not opted for traditional space partitioning index structures (e.g., k-d trees, R-trees, etc.), as these require a wasteful $O(nlogn)$ construction time.

Our *partition* function runs on a master node centrally and uses a hash-based adaptation of equi-depth histograms [42] for speed and simplicity. Instead of ordering the objects on each axis and then partitioning each axis sequentially for a time complexity of $O(nlogn)$, our *partition* function first hashes the objects, based on their location, into $p_{axis} < n$ sorted equi-width buckets on each axis, and then partitions each axis by grouping these buckets for $O(n + \sqrt{mn})$ time.

Particularly, our *partition* function (Algorithm 2) splits the x-axis into p_x equi-width buckets and hashes each object o in O in the corresponding x-axis bucket (Line 3). Then it groups all x-axis buckets into $\lceil\sqrt{m}\rceil$ vertical partitions (*xpartition*) so that no group has more than $\frac{n}{\sqrt{m}}+\frac{1}{2}|bucket|$ objects (Line 4-9). The last x-axis partition gets the remaining buckets. Next, it splits the y-axis into p_y equi-width buckets. For each generated vertical partition $xpartition_i$ it hashes object $o \in$

Algorithm 2 - $\mathit{partition}(A, m)$ Algorithm

```
Given: object space $A$, number of partitions $m$, set of objects $O$, number of buckets per axis $p_{axis}$
 1: $partition_s = \emptyset$, $1 \leq s \leq m$                  ▷ initialize final partitions
 2: $xpartition_r = \emptyset$, $1 \leq r \leq \lceil\sqrt{m}\rceil$            ▷ initialize x-axis partitions
 3: xbuckets = equi-width hash $\forall o \in O$ into $p_x$ buckets
 4: for all bucket in xbuckets do
 5:     if $|xpartition_r| + \frac{1}{2}|bucket| > \frac{n}{\sqrt{m}}$ and $r < \sqrt{m}$ then
 6:         $r = r + 1$
 7:     end if
 8:     $xpartition_r \leftarrow bucket$
 9: end for
10: for all part in xpartitions do
11:     empty all ybuckets
12:     ybuckets=equi-width hash $\forall o \in part$ into $p_y$ buckets
13:     for all bucket in ybuckets do
14:         if $|partition_s| + \frac{1}{2}|bucket| > \frac{n}{m}$ and $s < \sqrt{m}$ then
15:             $s = s + 1$
16:         end if
17:         $partition_s \leftarrow bucket$
18:     end for
19: end for
```

$xpartition_i$ into the corresponding bucket (Line 12). Then it groups all y-axis buckets into $\lceil\sqrt{m}\rceil$ *partitions* so that no group has more than $\frac{n}{m} + \frac{1}{2}|bucket|$ objects (Line 13-18). The last y-axis partition gets the remaining buckets.

The result is m partitions of approximately equal population, i.e., $\frac{n}{m} + \frac{1}{2}|bucket|$. The more buckets we hash into, i.e., larger values for p_x and p_y, the more "even" the populations will be. The time complexity of the partition function is determined by the number n of objects to hash into each bucket $(p_x + p_y)$ (Lines 3 and 12) and the nested-loop over all $\sqrt{m}$ xpartitions (Lines 10-19). In our setting, $p_x < \sqrt{n}$ and $p_y < \sqrt{n}$ are used in the internal loop (Lines 13-18). Thus, the total time complexity is $O(n + \sqrt{mn}) = O(n)$, since $n > m$.

3.3 Step 2: Replication

The theoretical foundation of our replication algorithm is based on the notion of "hiding". Intuitively, given the kNNs of a line segment or corner b and a set of points O_i on one side of b, it is guaranteed that any point belonging to the opposite side of b, other than the given kNNs of b, is not a kNN of O_i.

Each server s_i computes the *External Candidates* EC_{ji} for each of its adjacent servers $s_j \in Adj_i$ (Algorithm 1, Line 6-9). It runs the *computeECB* algorithm for each border segment or corner $b \in B_i$ (Line 7) and combines the results according to the adjacency between b and Adj_i (Line 8).

computeECB (Algorithm 3) scans all the objects in O_i once to find the $kNN(b, O_i)$, i.e., the k objects with the smallest *mindist* to border b (Line 2), where $mindist(o, b) =$

Algorithm 3 - *computeECB(b,O_i)* Algorithm

```
Given: border segment (or corner) $b$, object set $O_i$
1: construct Min Heap $H_b$ from $O_i$ based on $mindist(o,b)$
2: $kNN(b,O_i)$ = extract top k objects from $H_b$
3: $\theta_b \leftarrow max_{p \in kNN(b,O_i)}\{maxdist(p,b)\}$
4: for all $o \in O_i$ do
5:     if $mindist(o,b) < \theta_b$ then
6:         $EC_b = EC_b \cup o$
7:     end if
8: end for
9: return $EC_b$
```

$min_{p \in b}\{dist(o,p)\}$ and p is any point on b. Note, that the partitioning step guarantees that each server will have at least k objects if $m < \frac{n}{k} - |bucket|$.

A pruning threshold θ_b is determined by $kNN(b,O_i)$ and used to prune objects that should not be part of EC_b. Specifically, threshold θ_b is the worst (i.e., largest) $maxdist(o,b)$ of any object $o \in kNN(b,O_i)$ to border b (Line 3), where $maxdist(o,b)$ is defined as $maxdist(o,b)=max_{p \in b}\ \{dist(o,p)\}$.

$$\theta_b = argmax_{p \in kNN(b,O_i)}\{maxdist(p,b)\} \tag{2}$$

Given θ_b, an object $o \in O_i$ is part of EC_b if and only if its *mindist* to b is smaller than θ_b (Line 4-8). Formally,

$$EC_b = \{o | o \in O_i \wedge mindist(o,b) < \theta_b\} \tag{3}$$

As s_i completes the computation of EC_{ji} for an adjacent server $s_j \in Adj_i$, it sends EC_{ji} to s_j and receives $EC_{ij'}$ from some $s'_j \in Adj_i$ that has completed the respective computation in an asynchronous fashion (Algorithm 1, Line 10-14). When all servers complete the replication step, each s_i have received set $EC_i = \bigcup_{s_j \in Adj_i} EC_{ij}$.

In the example of Figure 1.4, server s_1 has $O_1 = \{o_1, o_2, o_3, o_4\}$ and wants to run *computeECB* for $b = \overline{be}$. The 2 neighbors $2NN(b,O_1)$ of border b are $\{o_1, o_2\}$ and therefore $\theta_b = maxdist(o_1,b)$ (since $maxdist(o_1,b) > maxdist(o_2,b)$). Objects o_3 and o_4 do not qualify as part of EC_b, since $mindist(o_3,b) > \theta_b$ and $mindist(o_4,b) > \theta_b$, thus $EC_b = \{o_1, o_2\}$.

3.4 Step 3: Refinement

Having received EC_i, each server s_i computes $kNN(o, O_i \cup EC_i), \forall o \in O_i$ (Algorithm 1, Line 15). Any centralized main-memory AkNN algorithm [13, 17] that finds the kNNs from $O_i \cup EC_i$ for each object $o \in O_i$ (a.k.a. kNN-Join between sets O_i and $O_i \cup EC_i$) can be used for this step.

Algorithm 4 - *localAkNN*(O_i, EC_i) Algorithm

```
Given: External Candidates $EC_i$ and set of objects $O_i$
 1: partition the area $A_i$ into a set of cells $C_i$
 2: for all cells $c \in C_i$ do
 3:     construct Min Heap $H_c$ from $O_i$ on $mindist(o, c)$
 4:     $kNN(c, O_i)$ = extract top k objects from $H_c$
 5:     $\theta_c \leftarrow max_{p \in kNN(c,O_i)}\{maxdist(p, c)\}$
 6:     for all $o \in O_i$ do
 7:         if $mindist(o, c) < \theta_c$ then
 8:             $EC_c = EC_c \cup o$
 9:         end if
10:     end for
11:     compute $kNN(o, O_c \cup EC_c), \forall o \in O_c$
12: end for
```

In *Spitfire*, we partition sub-area A_i of server s_i into a grid of *equi-width* cells C_i. Each cell $c \in C_i$ contains a disjoint subset $O_c \subset O_i$ of objects. Next, we compute locally a correct external candidate set EC_c for each cell $c \in C_i$ (similarly to the replication step). Finally, we find the kNNs for each object $o \in O_i$ by computing $kNN(o, O_c \cup EC_c)$.

In Algorithm 4, objects $o \in O_i$ are scanned once to build a k-min heap H_c for each cell $c \in C_i$ based on the minimum distance $mindist(o, c)$ between o and the cell-border (Line 3). The first k objects are then popped from H_c to determine threshold θ_c, based on Equation (1.2) (Lines 4-5). Objects $o \in O_i$ are scanned once again to determine the *External Candidates* EC_c that satisfy the threshold as in Equation (1.3) (Lines 6-10). Finally, s_i computes $kNN(o, O_c \cup EC_c), \forall o \in O_c$ (Line 11).

Given optimal load balancing, the building phase (heap construction and External Candidates) completes in $O(\frac{n}{m})$ time, whereas *finding* the kNN within $O_c \cup EC_c$ completes in $O(f_{Spitfire} \frac{n}{m})$ time, where $\frac{n}{m} = |O_i|$.

3.5 Running Example

Given an object set O, assume that a set of servers $\{s_1, s_2, s_3, s_4\}$ have been assigned to sub-areas $\{abde, bcfe, efih, dehg\}$, respectively (see Figure 1.4). In the following, we discuss the processing steps of server s_1. The objects of s_1 are $O_1 = \{o_1, o_2, o_3, o_4\}$, its adjacent servers are $Adj_i = \{s_2, s_3, s_4\}$, and its border segments are $B_1 = \{a, \overline{ab}, b, \overline{be}, e, \overline{ed}, d, \overline{da}\}$. For simplicity we have defined the border segments to be a one-to-one mapping to the corresponding adjacent servers. As shown, border segment $\overline{be}$ is adjacent to server s_2, corner e is adjacent to server s_3, and segment $\overline{ed}$ is adjacent to server s_4.

Server s_1 locally computes EC_b for each $b \in B_i$. It does so by scanning all objects $o \in O_1$ and building a heap H_b for each border segment b based on $mindist(o, b)$. The k closest objects to each b are popped from H_b as a result. In

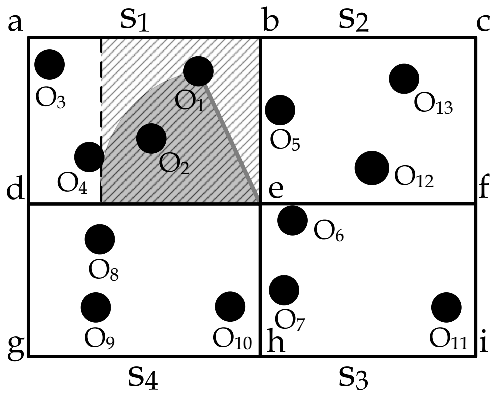


Fig. 4 Server s_1 sends $\{o_1, o_2\}$ to s_2, $\{o_1, o_2\}$ to s_3, and $\{o_2, o_4\}$ to s_4.

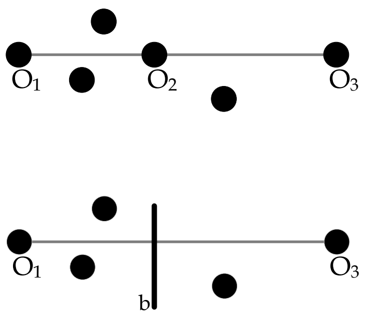


Fig. 5 *(Top)* o_2 *hides* o_1 from o_3, *(Bottom)* Segment b *hides* o_1 from o_3.

our example, $\{o_1, o_2\}$ are the k closest objects to segment $\overline{be}$, $\{o_1, o_2\}$ are the k closest objects to e, and $\{o_2, o_4\}$ are the k closest objects to segment $\overline{ed}$.

For each segment b, its pruning threshold θ_b is determined by the largest *maxdist* of its closest objects computed in the previous step. For instance, for segment $\overline{be}$ this is $\theta_b = \mathit{maxdist}(o_1, \overline{be})$, since $\mathit{maxdist}(o_1, \overline{be}) > \mathit{maxdist}(o_2, \overline{be})$. Given the thresholds θ_b, all objects $o \in O_1$ are scanned again and the condition $\text{mindist}(o, b) < \theta_b$ is checked for each segment b. If this condition holds then object o is part of EC_b. In our example, $EC_{\overline{be}} = \{o_1, o_2\}$, $EC_e = \{o_1, o_2\}$ and $EC_{\overline{ed}} = \{o_2, o_4\}$. Now s_1 sends $EC_{\overline{be}}$ to s_2, EC_e to s_3, and $EC_{\overline{ed}}$ to s_4, based on the adjacency described earlier.

Similarly, the above steps take place in parallel on each server. Therefore, s_1 receives from s_2 the $EC_{\overline{be}}$ of set O_2, from s_3 the $EC_{\overline{e}}$ of set O_3, and from s_4 the $EC_{\overline{ed}}$ of set O_4. Hence, server s_1 will be able to construct its $EC_1 = \bigcup_{b \in B_i} EC_b = \{o_5, o_6, o_7, o_8\}$. The External Candidate computation completes and the local kNN refinement phase initiates computing $\mathit{kNN}(o, O_i \cup EC_i), \forall o \in O_i$ on each server s_i.

3.6 Replication Factor: Spitfire vs. PGBJ

In this section, we qualitatively explain the difference of the replication factors f_{Spitfire} and f_{PGBJ} achieved by the replication strategies adopted in *Spitfire* and PGBJ, respectively. We use Figure 1.6 to illustrate the discussion.

In *Spitfire*, the cutoff distance for the candidates $\theta_{\mathit{Spitfire}}$ (defining the shaded bound) is determined by the maximum distance of the k closest external objects to the border segment b (let this be of length d). Now assume that all external objects are located directly on the border b. In this case, $\theta_{\mathit{Spitfire}} = d$. On the other

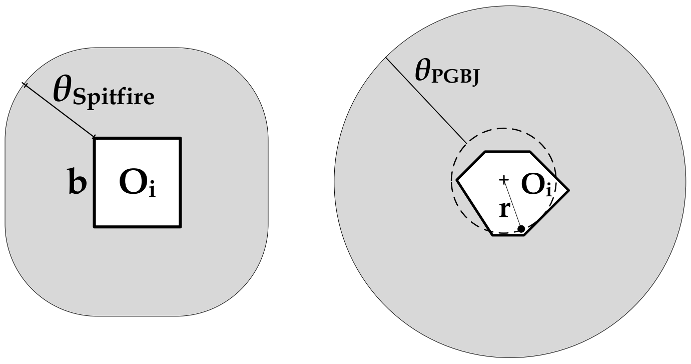


Fig. 6 Replication factor f in *Spitfire* (left) and PGBJ (right) shown as shaded areas.

extreme, assume that the external objects are exactly d distance from the border b where their worst case maximum distance to a border point would be $\sqrt{2} \cdot d$. In this case, $\theta_{Spitfire} = \sqrt{2} \cdot d$.

In PGBJ, the maximum distance between a pivot (+) and its assigned objects defines the radius r of a circular bound (dashed line), centered around the pivot. θ_{PGBJ} is determined by the maximum distance of the k closest objects to the pivot plus r. Now assume that all objects are located directly on the pivot. In this case, $r = 0$ and $\theta_{\text{PGBJ}} = 0$. On the other extreme, assume that all objects are on the boundary of the given Voronoi cell. In this case, $\theta_{\text{PGBJ}} = 2 \cdot r$. When d=r, $\theta_{Spitfire}$ has a $\sqrt{2}$ advantage over θ_{PGBJ}.

4 Experimental Evaluation

We show the evaluation of *Spitfire* using a real testbed on which all presented algorithms have been implemented. Here we present a summary of our experimental evaluation carried out in [10].

4.1 Experimental Testbed

Hardware: The evaluation is carried out on the DMSL VCenter[4] IaaS datacenter, a private cloud, which encompasses 5 IBM System x3550 M3 and HP Proliant DL 360 G7 rackables featuring single socket (8 cores) or dual socket (16 cores) Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, respectively. These hosts have collectively 300GB of main memory, 16TB of RAID-5 storage. The datacenter is man-

[4] DMSL VCenter @ UCY. `http://goo.gl/dZfTE5`

aged through a VMWare vCenter Server 5.1 that connects to the respective VMWare ESXi 5.0.0 hosts.

4.2 Datasets

The experiments use the following synthetic, realistic and real datasets:

Random (synthetic): This dataset was generated by randomly placing objects in space, in order to generate uniformly distributed datasets of 10K, 100K and 1M users.

Oldenburg (realistic): The initial dataset was generated with the Brinkhoff spatio-temporal generator [7], including 5K vehicle trajectories in a 25km x 25km area of Oldenburg, Germany. The generated spatio-temporal dataset was then decomposed on the temporal dimension, in order to generate realistic spatial datasets of 10K, 100K and 1M users.

Geolife (realistic): The initial dataset was obtained from the Geolife project at Microsoft Research Asia [66], including 1.1K trajectories of users moving in the city of Beijing, China over a life span of two years (2007-2009). Similarly to Oldenburg, the generated spatio-temporal dataset was decomposed on the temporal dimension, in order to generate realistic spatial datasets of 10K, 100K and 1M users.

Rayzit (real): This is a real spatial dataset of 20K coordinates captured by our Rayzit service during February 2014. We intentionally did not scale this dataset up to more users, in order to preserve the real user distribution.

4.3 Evaluated Algorithms

One centralized and four distributed algorithms are compared. They have been confirmed to generate identical correct results to the AkNN query.

Proximity [13]: This centralized algorithm runs on a single server and groups objects using a given space partitioning of cellular towers in a city. It computes the candidates kNNs of each area and scans those for each object within the area. Although this centralized algorithm is not competitive, we use it as a baseline for putting scalability into perspective.

H-BNLJ [64]: This is the two-phase MapReduce algorithm analyzed in Section 1.2.4, which partitions the object set randomly in $\sqrt{m}$ disjoint sets and creates their m possible pairs. Each server performs a kNN-join among each pair. Finally, the local results are gathered and the top-k results are returned as the final k nearest neighbors of each object.

H-BRJ [64]: This is the same algorithm as H-BNLJ, only it exploits an R-tree when performing the kNN-join to reduce the computation time.

PGBJ [39]: This is the two-phase MapReduce algorithm analyzed in Section 1.2.4, which partitions the space based on a set of pivot points generated in a preprocessing step. The candidate set is then computed based on the distance of each point to each pivot. We use the original implementation kindly provided by the authors of PGBJ that comes with the following configurations: (i) the number of pivots used is set to $P = 4000$ (i.e., $\approx \sqrt{n}$, for $n = 1M$ objects).

Spitfire: This is the algorithm proposed in our earlier work [10]. The only configuration parameter we use is the optimal border segment size d_b, which can be derived analytically, given the provided cluster of m nodes, the preference k, a dataset (A,n), the *CPU* speed of the servers and the *LAN* speed as shown in [10].

The traditional Hadoop implementation transfers intermediate results between tasks through a *disk-oriented* Hadoop Distributed File System (HDFS). For fair comparison we port all MapReduce algorithms to UC Berkeley's Tachyon [57] in-memory file system to enable *memory-oriented* data-sharing across MapReduce jobs. As such, the algorithms presented in this section have no Disk I/O operations, i.e., we are thus only concerned with minimizing Network I/Os (NI/Os).

4.4 Metrics and Configuration Parameters

Response Time: *This represents the actual time required by a distributed AkNN algorithm to compute its result.* We do not include the time required for loading the initial objects to main memory of the m servers or writing the result out. We use this setting to capture the processing scenarios deployed in our real Rayzit system architecture. Times are averaged over five iterations measured in seconds and plotted in log-scale, unless otherwise stated.

Replication Factor (f): *This represents the number of times the n objects are replicated between servers to guarantee correctness of the AkNN computation.* f determines the communication overhead of distributed algorithms, as described in Section 1.2. A good algorithm is expected to have a low replication factor (when f=1 there is no replication of objects).

Load Balancing: *This represents the standard deviation of the number of objects each server receives in its partition.* Higher deviations suggest that the load has not been balanced correctly, whereas a good algorithm is expected to have a low standard deviation.

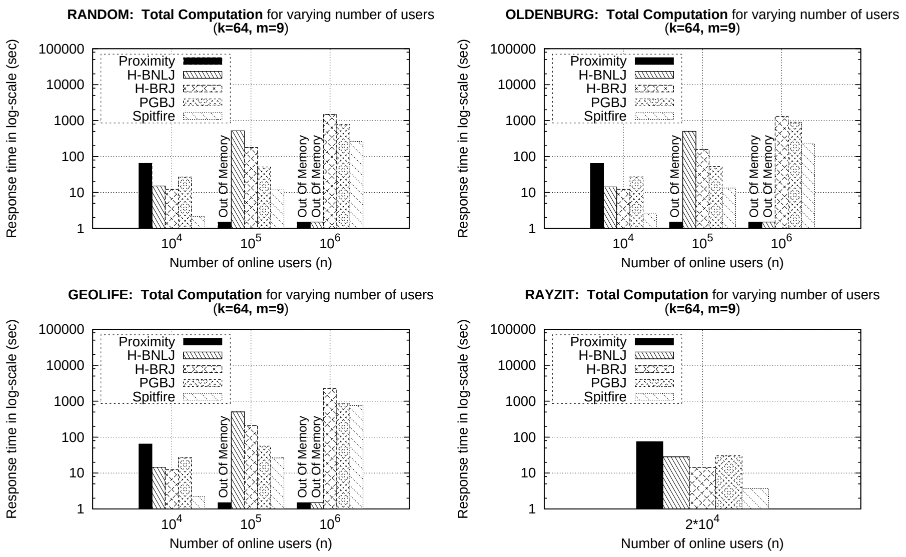


Fig. 7 AkNN query response time with increasing number of users. We compare the proposed *Spitfire* algorithm against the three state-of-the-art AkNN algorithms and a centralized algorithm on four datasets.

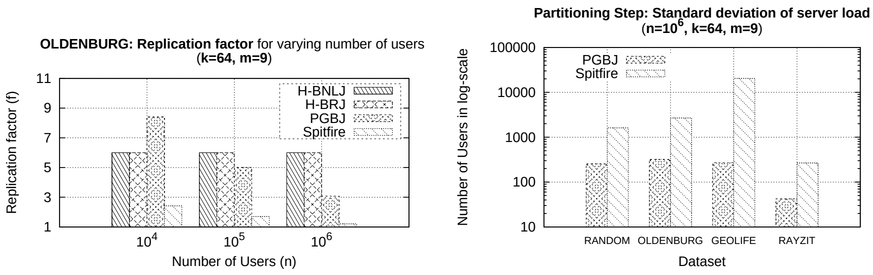


Fig. 8 Left: Replication factor f with increasing number of users. The optimal value for f is 1, signifying no replication. Right: Partitioning step: load balancing achieved (less is better). H-BNLJ and H-BRJ achieve optimal load balancing (standard deviation among server load ≈ 0).

4.5 Results

We increase the workload of the system by growing the number of online users (n) exponentially and measure the response time of the algorithms under evaluation.

Total Computation: In Figure 1.7, we measure the total response time for all algorithms, datasets and workloads. We can clearly see that *Spitfire* outperforms all other algorithms in every case. It is also evident that H-BNLJ and H-BRJ do not scale. H-BRJ achieves the worst time for 10^6 users. Most of H-BRJ's response time is spent in communication, which is indicated theoretically by its communication complexity of $O(\sqrt{mn})$ shown in Table 1.2. For 10^4 online users, *Spitfire* outperforms all algorithms by at least 85% for all dataset, whereas for 10^5 *Spitfire* outperforms PGBJ, by 75%, 75% and 53% for the Random, Oldenburg and Geolife datasets, respectively.

Spitfire and PGBJ are the only algorithms that scale. For a million online users (n=10^6), *Spitfire* and PGBJ are the fastest algorithms, but *Spitfire* still outperforms PGBJ by 67%, 75%, 14% for the Random, Oldenburg and Geolife datasets, respectively.

Replication Factor: In Figure 1.8 (left) we measure the replication factor for the distributed algorithms. It is noteworthy that the replication factor $f_{Spitfire}$ of *Spitfire* is always close to the optimal value 1. *Spitfire* only selects a very small candidate set around the border of each server (Algorithm 3 in Section 1.3.3). As analyzed in Section 1.3.6, in the worst case scenario $f_{Spitfire}$ is only $\sqrt{2}$ times smaller than f_{PGBJ}, but we see that for real datasets $f_{Spitfire}$ is at least half of f_{PGBJ}. Finally, $f_{\text{H-BNLJ}} = f_{\text{H-BRJ}} = 2\sqrt{m} = 6$ independently of n, as described in Section 1.2.4.

This experimental series demonstrates the algorithmic advantages offered by Spitfire free from any effect that the implementation framework might add.

Load-balancing: In the final result we focus only on *Spitfire* and PGBJ, which expose the best performance. Figure 1.8 (right) shows that the partitioning technique used by PGBJ achieves almost full load-balancing (i.e., $\pm$ 270 for 10^6 objects), while *Spitfire* achieves a less balanced workload among servers (i.e., $\pm$ 20,315 for 10^6 objects). Clearly, such a workload distribution will force certain servers to perform more distance calculations and will require higher synchronization time. Note, that the load balancing achieved by H-BNLJ and H-BRJ is optimal (standard deviation of object load on servers $\approx$ 0 not depicted in the figure), because they do not perform spatial partitioning but rather arbitrarily split the original object set into equally sized subsets.

5 Case Study on Rayzit Application

In this section we study Rayzit as a case of mobile social network that is based on dynamic social neighborhood that inherit their structure from the movement of the actual mobile crowd.

5.1 How Rayzit works

Rayzit was launched in 2014 and has redefined anonymous messaging and social interactions. The main functionality of an anonymous social network is to provide ways for users to publish opinions, thoughts or gossip while preserving anonymity. Connecting each user with the k geographical closest peers (strangers) allows them to communicate anonymously. These connections are updated as users move.

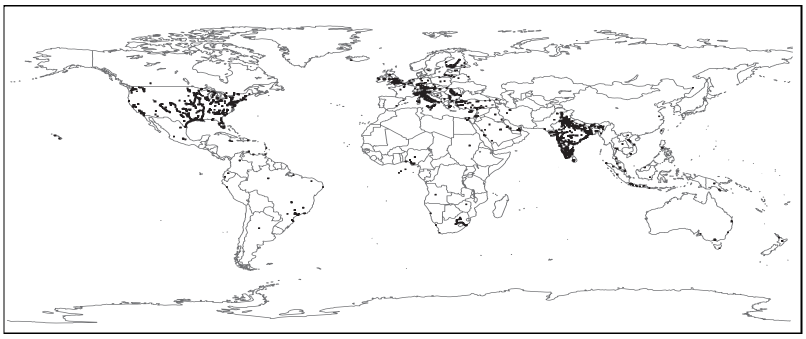


Fig. 9 *Rayzit* global user community.

The *Rayzit* mobile application allows users to send messages (rayzes) with media attachments, and reply to rayz posts based on a kNN network. In more details, the messages will be sent to the k nearest users. For each rayz that a user sends, the power of the user is decreasing. When the power of a user reaches 0 they can not post or reply any more. This functionality was incorporated in order to minimize spamming. The power of the user is fully reset every 24 hours or if the user's rayz is powered up by another user. In addition, a user can star or re-rayz any rayz which they consider interesting. Re-rayzing is the procedure of re-posting an existing rayz to one's own k nearest peers (similar to the re-tweet functionality in Twitter) and therefore propagating the post to more users.

Figure 1.9 shows the collection of active users in *Rayzit*. It is evident that most users have their nearest neighbors inside a small area. Now consider the scenario, where somebody has lost her dog and wants to start looking in a neighborhood in order to find it and at the same time asks for help from the social network. *Rayzit* offers this functionality out of the box. The user can post a picture of the dog, which the users in the vicinity will see. If someone sees a dog that matches the attached picture (see in Figure 1.1), they can reply to the post with whereabouts and/or photos of the lost pet.

In order to comprehend how the *Rayzit* works, consider the aforementioned scenario in terms of communication interactions. First of all, the client sends the rayz attaching a photo of the lost dog. The post, the photo, and their location is sent to the load balancer, which assigns them to a web server. The responsible web server stores the rayz through the Big data module. Then the social network module uses the dynamic social AkNN network computed by the AkNN engine in order to propagate the rayz to the k nearest neighbors. The k nearest neighbors will receive the rayz into their live feed and they can reply if they have any information.

The *Rayzit* backend architecture features a HAProxy[5] HTTP load balancer to distribute the load to respective Apache HTTP servers (see Figure 1.10). Each server also features a Couchbase NoSQL document store[6] for storing the messages posted

[5] HAProxy. `http://haproxy.1wt.eu/`

[6] Couchbase. `http://www.couchbase.com/`

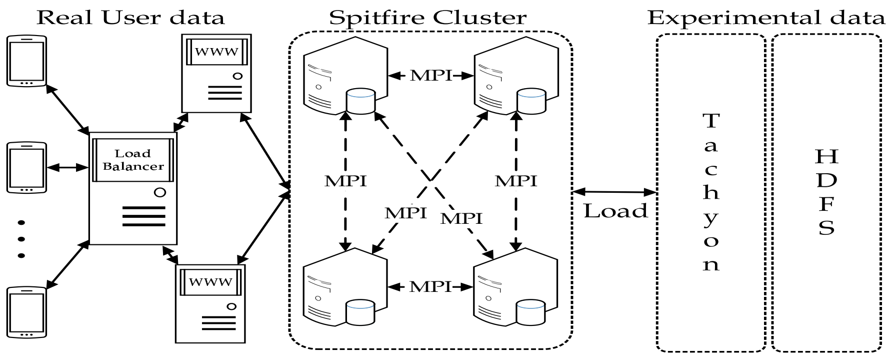


Fig. 10 Our Rayzit architecture.

by our users. In Couchbase, data is stored across the servers in JSON format, which is indexed and directly exposed to the Rayzit Web 2.0 API[7]. In the backend, we run the computing node cluster that carries out the AkNN computation as discussed in this work. The results are passed to the servers through main memory (i.e., MemCached) every few seconds.

5.2 *Data analysis and Evaluation*

In this section, we have collected data produced by our *Rayzit* application and provide a brief data analysis. The data consists of several elements such as UUID and timestamp. Furthermore, we conduct experiments for evaluating and analyzing the performance of *Rayzit*.

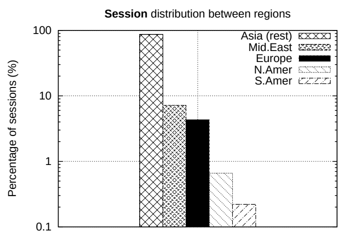
Our evaluation focuses on two aspects: (1) the average percentage of replies per rayz at a specific time; and (2) how the distance affects the number of replies. We used Flurry.com analytics to get usage and distribution statistics of *Rayzit* app.

Session measurements

In this experiment we study the usage of *Rayzit* app in respect to region and duration of interaction.

The histograms in Figure 1.11 show the distribution of sessions in percentage. The session length is defined simply as the length of time between the start application event and the end application event. The Figure 1.11 (right) plots the distribution of the sessions by displaying the number of sessions for which the session length falls into predefined duration slots.

[7] Rayzit API. `http://api.rayzit.com/`



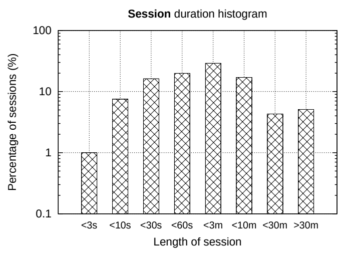
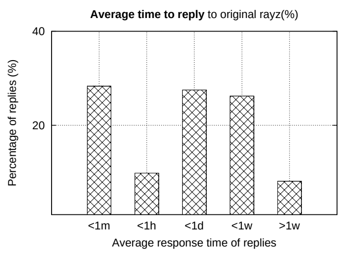


Fig. 11 (Left) Distribution of *Rayzit* users across regions in log scale. (Right) Average duration of each user session (reading and/or posting) in log scale.



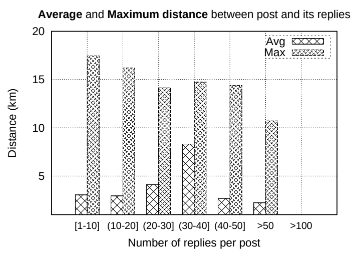


Fig. 12 (Left) Average time elapse between a post (rayz) and all its replies. (Right) The average and maximum distance between the locations of a post (rayz) and its replies, grouped by number of replies per post.

Figure 1.11 (left) plots the distribution of sessions in percentage for each region. More than 87.6% of sessions are created in Asia, and more than 7.2% of them in Middle East. 4.3% of all sessions are created in Europe. Only 0.7% of sessions are created in North America. Only 0.2% of the sessions have been created in South America.

Figure 1.11 (right) plots the distribution of *Rayzit*'s sessions across duration time slots. It is noticeable that 1% of the sessions is completed within 3 seconds. 7.50% completed between 3 and 10 seconds and 16.20% observed between 10 and 30 seconds. 19% of the sessions completed between 30 and 60 seconds. The time slot of 1 to 3 minutes has the maximum percentage with 29%. Furthermore, 17% completed within 10 minutes and 4.3% between 10 and 30 minutes. Finally, only the 4.3% completed after 30 minutes. Consequently, the AkNN engine needs to deliver the dynamic kNN graph in less than a minute to satisfy most users.

Interaction measurements

In this experiment we study how users interact through *Rayzit*, and in particular how fast a user replies to a post and how the distance affects interaction.

We used real data that was collected through the *Rayzit* back-end. The data include the time and location stamp of a post or a reply. The time between a rayz and a reply is presented in time slots: $(< 1m)$ less than a minute, $(< 1h)$ between a minute and an hour, $(< 1d)$ between an hour and a day, $(< 1w)$ between a day and a week and $(> w)$ more than a week. The average and maximum distances of the replies for each rayz were measured in order to depict how the distance affects the number of replies.

Figure 1.12 (left) plots the distribution of the reply arrival time, which is the time between each reply and the original rayz. 28% of replies arrive within the first minute of the original rayz, and more than 28% of replies arrive within a day. Only 10% of replies arrive with in the first hour. 26% of replies arrive between the first day and the first week. Only the 8% of replies arrive one week or more after the rayz creation. This confirms our intuition for the temporal dimension of a rayz: it is very unlikely to get attention after a while.

Figure 1.12 (right) plots the distance that a rayz can reach. It is noticeable that rayz posts with 1 - 50 replies have a maximum distance of almost 15km. Rayz posts with more than 50 replies have 10km maximum distance and 2km average distance. Moreover, rayz posts with more than 100 replies have only 8m maximum and 6m average distance. This validates our initial claim that the nearest users are more likely to be related to a specific subject than the faraway users.

6 Conclusions and Future Directions

In this chapter we present algorithms for massive computations of social neighborhoods from mobile Big Data. *Spitfire* is a scalable and high-performance distributed algorithm that solves the AkNN problem using a shared-nothing cloud infrastructure. This algorithm offers several advantages over the state-of-the-art algorithms in terms of efficient partitioning, replication and refinement. Theoretical analysis and experimental evaluation show that *Spitfire* outperforms existing algorithms reported in recent literature, achieving scalability both on the number of users and on the number of k nearest neighbors.

Future directions of this research include: i) studying the temporal extensions to support more gracefully higher-rate AkNN scenarios with streaming data; ii) studying AkNN queries over high-dimensional data; iii) providing an approximate AkNN version of *Spitfire*; iv) developing online geographic hashing techniques at the network load-balancing level; and finally v) porting our developments to general open-source large-scale data processing architectures (e.g., Apache Spark [63] and Apache Flink [26]).

References

1. S. Abe, R. Thawonmas, Y. Kobayashi, "Feature selection by analyzing regions approximated by ellipsoids," *IEEE Trans. on Systems, Man, and Cybernetics, Part. C*, vol. 28, no. 2, pp.

282–287, 1998.

2. A. Abraham, "Evonf: A framework for optimization of fuzzy inference systems using neural network learning and evolutionary computation," In *IEEE International Symposium on Intelligent Control*, ISIC02, IEEE Press, ISBN 0780376218, pp. 327–332, Canada, 2002.
3. A. Abraham, "i-miner: A web usage mining framework using hierarchical intelligent systems," In *The IEEE International Conference on Fuzzy Systems* FUZZ-IEEE03, IEEE Press, ISBN 0780378113, pp. 1129–1134, USA, 2003.
4. F.N. Afrati, A.D. Sarma, S. Salihoglu and J.D. Ullman. "Upper and lower bounds on the cost of a map-reduce computation". *In Proceedings of the 39th international conference on Very Large Data Bases (PVLDB'13)*, VLDB Endowment, 277–288, 2013.
5. N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," In *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.
6. C. Boehm and F. Krebs, "The k-nearest neighbour join: Turbo charging the kdd process," *Knowledge Information Systems*, vol. 6, no. 6, pp. 728–749, Nov. 2004.
7. T. Brinkhoff. "A framework for generating network-based moving objects". *Geoinformatica*, Vol. 6, 153–180, 2002.
8. P.B. Callahan. "Optimal parallel all-nearest-neighbors using the well-separated pair decomposition". *In Proceedings of the 34th IEEE Annual Foundations of Computer Science (SFCS'93)*, 332–340, 1993.
9. G. Chatzimilioudis and C. Costa and D. Zeinalipour-Yazti and W.-C. Lee. "Crowdsourcing emergency data in non-operational cellular networks". *In Elsevier Science Information Systems*, 2015
10. G. Chatzimilioudis and C. Costa and D. Zeinalipour-Yazti and W. C. Lee and E. Pitoura. "Distributed In-Memory Processing of All k Nearest Neighbor Queries" *IEEE Transactions on Knowledge and Data Engineering*, vol.28, 925–938, 2016.
11. G. Chatzimilioudis and A. Konstantinidis and C. Laoudias and D. Zeinalipour-Yazti. "Crowdsourcing with Smartphones". *In IEEE Internet Computing*, vol.16, 36–44, 2012
12. G. Chatzimilioudis and A. Konstantinidis and D. Zeinalipour-Yazti. "Nearest Neighbor Queries on Big Data" *Information Granularity, Big Data, and Computational Intelligence*, 3–22, 2015.
13. G. Chatzimilioudis, D. Zeinalipour-Yazti, W.-C. Lee and M.D. Dikaiakos. "Continuous all k-nearest neighbor querying in smartphone networks". *In Proceedings of the 13th IEEE International Conference on Mobile Data Management (MDM'12)*, 79–88, 2012.
14. A. A. S. Chebrolu, J. Thomas, "Feature deduction and ensemble design of intrusion detection systems," *Computers and Security*, Elsevier, 2004.
15. Y. Chen and J.M. Patel. "Efficient evaluation of all-nearest-neighbor queries". *In Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE'07)*, 1056–1065, 2007.
16. Y. Chen, B. Yang, J. Dong, A. Abraham, "Time-series forecasting using flexible neural tree model," *Information Sciences*, vol. 174, no. 3–4, pp. 219–235, 2005.
17. K.L. Clarkson. "Fast algorithms for the all nearest neighbors problem". *In Proceedings 24th Annual Symposium on Foundations of Computer Science (FOCS'83)*, 226–232, 1983.
18. C. Costa, C. Anastasiou, G. Chatzimilioudis and D. Zeinalipour-Yazti. "Rayzit: An Anonymous and Dynamic Crowd Messaging Architecture". *In Proceedings of the 3rd IEEE Intl. Workshop on Mobile Data Management, Mining, and Computing on Social Networks (Mobisocial'15)*, Vol. 2, Pages: 98-103, IEEE Computer Society, 2015.
19. T. Cover and P. Hart. "Nearest neighbor pattern classification" *in IEEE Transactions of Information Theory*, vol. 13, no. 1, pp. 21–27, Sep. 2006.
20. J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". *In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*, Vol. 6, USENIX Association, Berkeley, CA, USA, 10–23, 2004.
21. F. Dehne, A. Fabri and A. Rau-Chaplin. "Scalable Parallel Computational Geometry for Coarse Grained Multicomputers". *International Journal on Computational Geometry*, Vol. 6, 379–400, 1996.

22. D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker and D. Wood. “Implementation techniques for main memory database systems” *In Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD’84)*, 1–8, 1984.
23. G. Ditzler, M. Roveri, C. Alippi and R. Polikar. “Learning in Nonstationary Environments: A Survey” *in IEEE Computational Intelligence Magazine*, vol. 10, no. 4, pp. 12–25, Nov. 2015.
24. M. Dorigo, M. Birattari and T. Stutzle. “Ant colony optimization” *in IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, Nov. 2006.
25. H. Duan, Q. Luo, Y. Shi and G. Ma. “Hybrid Particle Swarm Optimization and Genetic Algorithm for Multi-UAV Formation Reconfiguration” *in IEEE Computational Intelligence Magazine*, vol. 8, no. 3, pp. 16–27, Aug. 2013.
26. S. Ewen, S. Schelter, K. Tzoumas, D. Warneke and V. Markl. “Iterative parallel data processing with stratosphere: an inside look”. *In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD’13)*, 1053–1056, 2013.
27. H.N. Gabow, J.L. Bentley and R.E. Tarjan. “Scaling and related techniques for geometry problems”. *In Proceedings of the 16th ACM symposium on Theory of computing (STOC’84)*, 135–143, 1984.
28. L. 0. Hall, N. Chawla , K. W. Bowyer, “Decision Tree Learning on Very Large Data Sets,” In *IEEE International Conference on System, Man and Cybernetics (SMC)* , pp. 187–222, Oct 1998.
29. P. E. Hart, “The Condensed Nearest Neighbor Rule” In *IEEE Transactions on Information Theory*, no. 18, pp. 515-516, 1968.
30. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker, and I. Stoica. “Mesos: a platform for fine-grained resource sharing in the data center”. *In Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI’11)*, 22–35, 2011.
31. J. Hu, H. Tang, K. C. Tan and H. Li. “How the Brain Formulates Memory: A Spatio-Temporal Model Research Frontier” *in IEEE Computational Intelligence Magazine*, vol. 11, no. 2, pp. 56–68, May 2016.
32. H. Ishibuchi, T. Nakashima, T. Murata, “Performance evaluation of fuzzy classifier systems for multidimensional pattern classification problems,” *IEEE Trans. SMCB*, vol. 29, pp. 601–618, 1999.
33. E.H. Jacox and H. Samet. “Spatial join techniques”. *ACM Transactions of Database Systems*, Vol. 32, 1–8, 2007.
34. T. W. Jayyousi and R. G. Reynolds. “Extracting Urban Occupational Plans Using Cultural Algorithms [Application Notes]” *in IEEE Computational Intelligence Magazine*, vol. 9, no. 3, pp. 66–87, Aug. 2014.
35. P. Jiang, Z. Feng, Y. Cheng, Y. Ji, J. Zhu, X. Wang, F. Tian, J. Baruch and F. Hu. “A Mosaic of Eyes” *in IEEE Robotics & Automation Magazine*, vol. 18, no. 3, pp. 104–113, Sept. 2011.
36. T.H. Lai and M.-J. Sheng. “Constructing euclidean minimum spanning trees and all nearest neighbors on reconfigurable meshes”. *IEEE Transactions of Parallel Distributed Systems*, Vol. 7, 806–817, 1996.
37. Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, A. Ng, “Building high-level features using large scale unsupervised learning,” In *International Conference in Machine Learning*, 2012.
38. Y.-L. Lu and C.-S. Fahn, “Hierarchical Artificial Neural Networks for Recognizing High Similar Large Data Sets,” In *International Conference on Machine Learning and Cybernetics*, vol. 7, pp. 1930–1935, 2007.
39. W. Lu, Y. Shen, S. Chen and B.C. Ooi. “Efficient processing of k nearest neighbor joins using mapreduce”. *In Proceedings of the 38th international conference on Very Large Data Bases (PVLDB’12).* VLDB Endowment 5, 1016–1027, 2012.
40. J. Mao, K. Jain, “Artificial neural networks for feature extraction and multivariate data projection,” *IEEE Trans. on Neural Networks*, vol. 6, no. 2, pp. 296–317, 1995.
41. H. Muhleisen and K. Dentler “Large-Scale Storage and Reasoning for Semantic Data Using Swarms” *in IEEE Computational Intelligence Magazine*, vol. 7, no. 2, pp. 32–44, May 2012.

42. M. Muralikrishna, D.J. DeWitt. "Equi-depth multidimensional histograms". *In Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD'88)*, 28–36, 1988.
43. D. Nauck, R. Kruse, "Obtaining interpretable fuzzy classification rules from medical data," *Artificial Intelligence in Medicine*, vol. 16, pp. 149–169, 1999.
44. E.C. Ngai, M.B. Srivastava, L. Jiangchuan. "Context-aware sensor data dissemination for mobile users in remote areas". *In Proceedings of the IEEE international conference on computer communication (INFOCOM'12)*, 2711–2715, 2012.
45. N. Nodarakis, E. Pitoura, S. Sioutas, O. Tsakalidis, D. Tsoumakos, G. Tzimas. "Efficient Multidimensional AkNN Query Processing in the Cloud". *In Proceedings of the 25th international conference on database and expert systems applications (DEXA'14)* , LNCS 8644, 477–491, 2014.
46. P. Pacheco. "Parallel Programming with MPI". Morgan Kaufman, 1997.
47. D. V. Patil, R. S. Bichkar, "A Hybrid Evolutionary Approach To Construct Optimal Decision Trees With Large Data Sets," In *IEEE International Conference on Industrial Technology*, pp. 429–433, 2006.
48. E. Plaku and L. E. Kavraki, "Distributed Computation of the Knn Graph for Large High-dimensional Point Sets". *Journal of Parallel Distributed Computing*, Vol. 67 (3), 346–359, 2007.
49. C. Reeves, "Genetic algorithms," *Handbook of Metaheuristics*, Kluwer, pp. 65–82, 2003.
50. M. Renz, N. Mamoulis, T. Emrich, Y. Tang, R. Cheng, A. Zufle, and P. Zhang. "Voronoi-based nearest neighbor search for multi-dimensional uncertain databases", *In Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE'13)*, 158–169, 2013.
51. L. Resnyansky "Social Modeling and Simulation for National Security" *in IEEE Technology and Society Magazine*, vol. 31, no. 1, pp. 17–25, Spring 2012.
52. R. G. Reynolds and M. Ali "Computing with the social fabric: The evolution of social intelligence within a cultural framework" *in IEEE Computational Intelligence Magazine*, vol. 3, no. 1, pp. 18–30, February 2008.
53. N. Roussopoulos, S. Kelley and F. Vincent. "Nearest neighbor queries". *In Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD'95)*, 71–79, 1995.
54. U. Seiffert, F.-M. Schleif, D. Zhlke, "Recent trends in computational intelligence in life sciences," In *ESANN*, 2011.
55. M. Setnes, R. Babuska, H. Verbruggen, "Rule-based modeling: precision and transparency," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* vol. 28, pp. 165–169, 1998.
56. W. Shang, H. Huang, H. Zhu, Y. Lin, Z. Wang, Y. Qu, "An Improved kNN Algorithm - Fuzzy kNN" In *Computational Intelligence and Security*, Lecture Notes in Computer Science, vol. 3801, pp. 741–746, 2005.
57. Tachyon, AMPLab UC Berkeley, `http://tachyon-project.org/`
58. B. Tang and H. He. "ENN: Extended Nearest Neighbor Method for Pattern Recognition [Research Frontier]" *in IEEE Computational Intelligence Magazine*, vol. 10, no. 3, pp. 52–60, Aug. 2015.
59. S. Thomas, and Y. Jin, "Reconstructing biological gene regulatory networks: where optimization meets Big Data," *Evolutionary Intelligence*, pp. 1-19, 2013.
60. P.M. Vaidya. "An o(n log n) algorithm for the all-nearest-neighbors problem". *Discrete Computational Geometry*, Vol. 4, 101–115, 1989.
61. C. Xia, H. Lu, B. Chin Ooi and J. Hu. "Gorder: an efficient method for knn join processing". *In Proceedings of the 30th international conference on Very large data bases (VLDB'04)*, VLDB Endowment 30, 756–767, 2004.
62. B. Yao, F. Li and P. Kumar. "K nearest neighbor queries and knn-joins in large relational databases (almost) for free". *In Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE'10)*, 4–15, 2010.

63. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing”. *In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, 10–16, 2012.
64. C. Zhang, F. Li and J. Jestes. “Efficient parallel knn joins for large data in mapreduce”. *In Proceedings of the 15th International Conference on Extending Database Technology (EDBT'12)*, 38–49, 2012.
65. J. Zhang, N. Mamoulis, D. Papadias and Y. Tao. “All-nearest-neighbors queries in spatial databases”. *In Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, 297–306, 2004.
66. Y. Zheng, L. Liu, L. Wang and X. Xie. “Learning transportation mode from raw gps data for geographic applications on the web”. *In Proceedings of the 17th international conference on World Wide Web (WWW'08)*, 247–256, 2008.