Distributed In-Memory Processing of All k Nearest Neighbor Queries

Georgios Chatzimilioudis, Constantinos Costa, Demetrios Zeinalipour-Yazti, *Member, IEEE*, Wang-Chien Lee, *Member, IEEE*, and Evaggelia Pitoura, *Member, IEEE*

Abstract—A wide spectrum of Internet-scale mobile applications, ranging from social networking, gaming and entertainment to emergency response and crisis management, all require efficient and scalable All k Nearest Neighbor (AkNN) computations over millions of moving objects every few seconds to be operational. Most traditional techniques for computing AkNN queries are centralized, lacking both scalability and efficiency. Only recently, distributed techniques for shared-nothing cloud infrastructures have been proposed to achieve scalability for large datasets. These batch-oriented algorithms are sub-optimal due to inefficient data space partitioning and data replication among processing units. In this paper we present *Spitfire*, a distributed algorithm that provides a scalable and high-performance AkNN processing framework. Our proposed algorithm 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. We evaluate, both analytically and experimentally, how the pruning efficiency of the *Spitfire* algorithm plays a pivotal role in reducing communication and response time up to an order of magnitude, compared to three other state-of-the-art distributed AkNN algorithms executed in distributed main-memory.

Index Terms—All kNN Queries, Space Partitioning, Data Replication, Main-Memory Processing, Shared-Nothing Architectures

1 Introduction

In the age of smart urban and mobile environments, the mobile crowd generates and consumes massive amounts of heterogeneous data [18]. Such streaming data may offer a wide spectrum of enhanced science and services, ranging from mobile gaming and entertainment, social networking, to emergency and crisis management services [7]. However, such data present new challenges in cloud-based query processing.

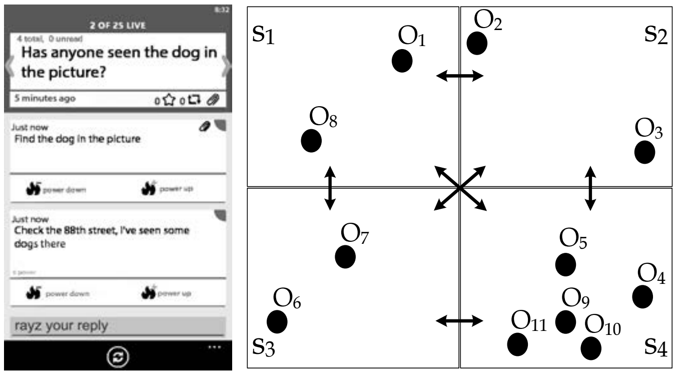
One useful query for the aforementioned services is the *All kNN (AkNN)* query: *finding the k nearest neighbors for all moving objects*. Formally, the kNN of an object o from some dataset O, denoted as *kNN*(o, O), are the k objects that have the most similar attributes to o [23]. Specifically, given 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). In our discussion, *dist* can be any L_p-norm distance metric, such as Manhattan (L_1), Euclidean (L_2) or Chebyshev (L_∞). 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$(o_a, O)\}$.*

A real-world application based on such a query is *Rayzit.com* [7], our award-winning crowd messaging architecture, that connects users instantly to their k *Nearest Neighbors (kNN)* as they move in space (Figure 1, left). Similar to other social network applications (e.g., Twitter, Facebook), scalability is key in making *Rayzit* functional 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 wide availability of off-the-shelf, shared-nothing, cloud infrastructures brings a natural framework to cope with scalability, fault-tolerance and performance issues faced in processing AkNN queries. Only recently researchers have proposed algorithms for optimizing AkNN queries in such infrastructures.

Specifically, the state-of-the-art solution [16] consists of three phases, namely *partitioning* the geographic area into sub-areas, computing the kNN *candidates* for each sub-area that need to be *replicated* among servers in order to guarantee correctness and finally, computing locally the global AkNN for the objects within each sub-area taking the *candidates* into consideration. The given algorithm has been designed with an offline (i.e., analytic-oriented) AkNN processing scenario in mind, as opposed to an online (i.e., operational-oriented) AkNN processing scenario we aim for in this work. The *performance* of [16] can be greatly improved, by introducing an optimized partitioning and replication strategy. These improvements, theoretically and experimentally shown to be superior, are critical in dramatically reducing the AkNN query processing cost yielding results within in a few seconds, as opposed to minutes, for million-scale object scenarios.

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

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, PA 16802, USA; E. Pitoura, University of Ioannina, 45110 Ioannina, Greece.



Has anyone seen the dog in the picture?

Just now
Find the dog in the picture

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

Fig. 1. (Left) Our Rayzit crowd messenger enabling users to interact with their k geographic Nearest Neighbors. (Right) Distributed main-memory AkNN computation in Rayzit is enabled through the *Spitfire* algorithm.

sides of the border and are close enough to each other. Consider the example illustrated at the right side of Figure 1, where 11 objects are partitioned over 4 spatial quadrants, each being processed by one of four servers $\{s_1 \dots s_4\}$. Now assume that we are interested in deriving the 2NN for each object $\{o_1 \dots o_{11}\}$. 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 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 (right), 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, in this work we also 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*.

In this paper we present *Spitfire*, a scalable and high-performance distributed algorithm that solves the AkNN problem in a *fast batch* mode using a shared-nothing cloud infrastructure of m servers. To address the aforementioned load balancing and communication issues, *Spitfire* starts out by partitioning O into disjoint sub-areas of approximately equal population using a fast equi-depth partitioning scheme. It then uses a threshold-based pruning algorithm to determine minimal *replication sets* to be exchanged between servers. Particularly, each server s_i receives from its neighboring servers a set of replicated objects potentially of interest, coined *External Candidates* (EC_i). EC_i supplements server s_i with all the needed external objects to compute the *correct* kNN for every $o \in O_i$, i.e., $kNN(o, EC_i \cup O_i) = kNN(o, O)$.

Particularly, *Spitfire* completes in three discrete phases. First, we devise a simple but fast centralized hash-based adaptation of equi-depth histograms [17] to *partition* the input O into disjoint subsets achieving good load balancing in $O(n + \sqrt{nm})$ time. To do this we first hash the objects based on their locations into a number of sorted equi-width buckets on each axis and then partition each axis sequentially by grouping these buckets in an equi-depth fashion. 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 the next phase. The given set EC_{ji} is *replicated* from s_i to s_j. 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.

Spitfire completes in only one communication round, as opposed to two communication rounds needed by the state-of-the-art [16], and its precise replication scheme has better pruning power, thus minimizing the communication cost/time as it is shown both analytically and experimentally in this work. The CPU time of *Spitfire* is $O(f_{Spitfire} \frac{n^2}{m^2})$ and its communication cost $O(f_{Spitfire} n)$, as this will be shown in Sections 3. We show that factor $f_{Spitfire}$ is always smaller than the factor achieved by the state-of-the-art [16]. Finally, *Spitfire* is implemented using the Message Passing Interface (MPI) framework [20]. This makes it particularly useful to large-scale main-memory data processing platforms (e.g., Apache Spark [27]), which have no dedicated AkNN operators.

In our previous work [4], we have presented a centralized algorithm named Proximity, which deals with AkNN queries in continuous query processing scenarios. In this work, we completely refocus the problem formulation to tackle the distributed in-memory AkNN query processing problem and propose the *Spitfire* algorithm. Our new contributions are summarized as follows:

- We devise *Spitfire*, a distributed algorithm that solves the AkNN problem in a *fast batch* mode, offering both *scalability* and *efficiency*. It encapsulates a number of innovative internal components, such as: (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 [4].
- We provide a formal proof of the correctness of our algorithm and a thorough analytical study of its performance.
- We conduct an extensive experimental evaluation that validates our analytical results and shows the superiority of *Spitfire*. Particularly, we use four datasets of various skewness to test real implementations of AkNN algorithms on our 9-node cluster, and report an improvement

of at least 50% in the pruning power of replicated objects that have to be communicated among the servers.

The remainder of the paper is organized as follows. Section 2 provides our problem definition, system model and desiderata, as well as an overview of the related work on distributed AkNN query processing. Section 3 presents our *Spitfire* algorithm with a particular emphasis on its partitioning and replication strategies, whereas Section 4 analyzes its correctness and complexity. Section 5 presents an extensive experimental evaluation and Section 6 concludes the paper.

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.

2.1 Goal and Design Principles

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

Research Goal. *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.

Performance: 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 [1]. 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: 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: 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 [3] 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, [9] 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 [29]) to find the kNN for each point separately. The authors in [21] 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 [19] 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 [4] 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 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 [8] (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[1], while cluster management is taken care of by YARN or Mesos [13]. 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 [2].

Hadoop Naive kNN Join (*H-NJ* [16]). 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 $O_i \bowtie_{kNN} O$ (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* [28]). 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 $O_i \bowtie_{kNN} O_j$. 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 $f_{\text{H-BNLJ}} = 2\sqrt{m}$.

Hadoop Block R-tree Loop kNN Join (*H-BRJ* [28]). 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* [16]): 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 [16].

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 $O(f_{\text{PGBJ}} n)$.

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 [16]	-	$O(n)$	$O(\frac{n^2}{m})$	$O(mn)$
H-BNLJ [28]	-	$O(n)$	$O(\frac{n^2}{m})$	$O(\sqrt{m}n)$
H-BRJ [28]	-	$O(n)$	$O(\frac{n}{\sqrt{m}} \log \frac{n}{\sqrt{m}})$	$O(\sqrt{m}n)$
PGBJ [16]	$O(\sqrt{n})$	$O(n^{1.5}/m)$	$O(f_{\text{PGBJ}} \frac{n^2}{m^2})$	$O(f_{\text{PGBJ}} n)$
Spitfire	-	$O(n)$	$O(f_{\textit{Spitfire}} \frac{n^2}{m^2})$	$O(f_{\textit{Spitfire}} n)$

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 [4], 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

1. Apache Hadoop. `http://hadoop.apache.org/`
2. Tachyon: `http://tachyon-project.org/`

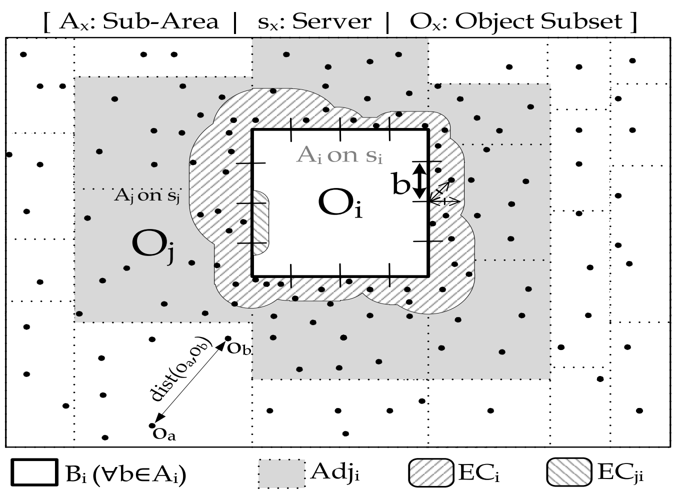


Fig. 2. *Spitfire* Overview: i) Space partitioning to equi-depth quadrants; ii) Replication between neighboring O_i and O_j using EC_{ji} and EC_{ij}, respectively; and iii) Local refinement within each $O_i \cup EC_i$.

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 2 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 [17] 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 2). 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.,

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

- **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.

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. 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. 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. 3.4)
```

The CPU time of *Spitfire* is $T(n) = n + \sqrt{nm} + f_{Spitfire}\frac{n^2}{m^2}$, as this will be shown in Sections 3.2 and 3.4, respectively. Its communication cost is 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.

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 [17] 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

Algorithm 2 - *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
```

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 $o \in xpartition_i$ 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", analyzed in detail later in Section 4.1. 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 $mindist(o, b) = min_{p \in b}\{dist(o, p)\}$ and p is any point on b. Note, that the partitioning step guarantees that each server will have at least k objects if $m < \frac{n}{k} - |bucket|$.

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 $EC_b$
```

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
```

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) (based on Theorem 1, Section 4). 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 3, 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 [4], [6] 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.

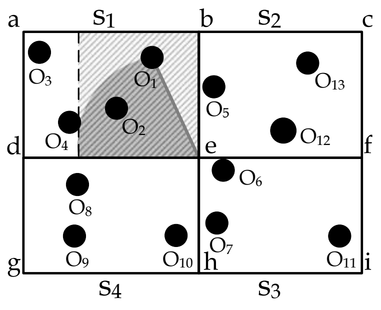


Fig. 3. Server s_1 sends $\{o_1, o_2\}$ to s_2, $\{o_1, o_2\}$ to s_3, and $\{o_2, o_4\}$ to s_4.

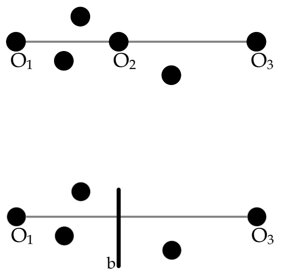


Fig. 4. *(Top)* o_2 *hides* o_1 from o_3, *(Bottom)* Segment b *hides* o_1 from o_3.

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 (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 (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 $O(f_{\textit{Spitfire}}\frac{n}{m})$ 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 3). 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 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 $\theta_b = \textit{maxdist}(o_1, \overline{be})$, since $\textit{maxdist}(o_1, \overline{be}) > \textit{maxdist}(o_2, \overline{be})$. Given the thresholds θ_b, all objects $o \in O_1$ are scanned again and the condition 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 *kNN*$(o, O_i \cup EC_i), \forall o \in O_i$ on each server s_i.

4 Correctness and Analysis

In this section we first show that our algorithm leads to a correct AkNN result, i.e., $\sum_i^m$ *kNN*$(o, EC_i \cup O_i) =$ *kNN*(o, O), based on the External Candidates determined by *computeECB*. Then, we analyze its computational and communication cost.

4.1 Correctness of the computeECB function

To prove correctness, we show that it suffices to compute the External Candidates EC_{B_i} to border B_i in order to find the External Candidates EC_i of the whole area A_i, given area A_i, its border B_i, and the necessary objects around B_i. In the following, we first define the notion of *point hiding*.

Definition 1 (Point Hiding)**.** *Given three points o_1, o_2, o_3 on a line, which holds the following relationship dist$(o_1, o_3) =$ dist$(o_1, o_2) +$ dist(o_2, o_3), we say that o_2 hides o_1 and o_3 from each other.*

In Figure 4 *(top)* point o_2 *hides* o_1 and o_3 from each other.

Lemma 1. *Given three points o_1, o_2, o_3 where o_2 hides o_1 from o_3 and the fact that o_1 is not a kNN of o_2, it holds that o_1 is not a kNN of o_3, and vice versa.*

Proof: To prove that o_1 is not a kNN of o_3 it suffices to prove that there are k points closer than o_1 is to o_3. The fact that o_1 is not a kNN of o_2 means that there are k other points, $\{p_1, p_2, ..., p_k\}$, in space that are closer to o_2 than is o_1, *dist*$(p_i, o_2) \leq$ *dist*(o_1, o_2). It holds that *dist*$(p_i, o_3) \leq$ *dist*$(o_1, o_2) +$ *dist*$(o_1, o_3) -$ *dist*(o_2, o_1) based on trigonometry, which gives *dist*$(p_i, o_3) \leq$ *dist*(o_1, o_3). Therefore there are k points, namely $\{p_1, p_2, ..., p_k\}$, that are closer to o_3 than is o_1 $\square$

Similarly, we can extend the notion of hiding from a point to a line segment, i.e., border. In Figure 4 *(bottom)* segment b *hides* o_1 and o_3 from each other.

Definition 2 (Segment Hiding)**.** *Given two points o_1, o_3, and a segment b, we say that b hides o_1 and o_3 from each other, when there is always a point $o \in b$ that hides o_1 and o_3 from each other.*

Lemma 2. *Given two points o_1 and o_3, a segment b that hides o_1 from o_3, and the fact that o_1 is not a kNN of any point on b, it holds that o_1 is not a kNN of o_3, and vice versa.*

Proof: It suffices if it holds that *dist*$(k_i, o_3) \leq$ *dist*(o_1, o_3) for $1 \leq i \leq k$. Given that o_1 is not a kNN of any point $p \in b$, then for each p there are k other points $\{k_1, k_2, ..., k_k\}^p$ in space. It holds that $dist(k_i^{p'}, o_3) \leq dist(o_1, o_3) - dist(o_1, p') + dist(o_1, p')$ based on trigonometry, which gives $dist(k_i^{p'}, o_3) \leq dist(o_1, o_3)$ for $1 \leq i \leq k$. □

Since border B_i of area A_i hides every point that is outside A_i from the points inside A_i, we can easily extend Lemma 1 and Lemma 2 into Lemma 3:

Lemma 3. *Given an area A_i, its objects O_i and its border segments B_i, then any object x outside area A_i, i.e., $x \notin O_i$, that is not a kNN of some point of border B_i is guaranteed not to be a kNN of any object inside A_i.*

4.2 Correctness of *Spitfire*

The correctness of computeECB performed on a single server assumes that for a given border b the server has access to all the kNN candidates of b. In a distributed environment this may not be the case as some candidates of b might span over several servers. More specifically, this happens when a server has less than k objects or when there is a θ_b that is greater than the side of the sub-area A_i assigned to the server.

The result of *Spitfire* is always correct since it deals with both cases gracefully. In particular, given a dataset of size n, *Spitfire* does not allow m to be set such that $\frac{n}{m} < k$ and furthermore, at the end of the partitioning step (Algorithm 2) it iterates through the m generated partitions $partition_s$, $1 \leq s \leq m$, to check whether $|partition_s| < k$. If this is the case, *Spitfire* re-instantiates itself using only $m/2$ servers in order to produce partitions with larger population.

To handle the second case, *Spitfire* also computes the side lengths of each partition during the partitioning step (Algorithm 2) and checks on each server during the replication step whether for any $b \in B_i$ it holds that $\theta_b >$ *partition_side_length* in Algorithm 3. If this is the case, *Spitfire* re-instantiates itself using only $m/2$ servers in order to produce partitions with bigger side lengths. The above controls are not shown in the Algorithms for clarity of presentation.

Given that each server s_i receives EC_i computed by function *computeECB* over all adjacent servers $s_j \in Adj_i$ we get:

Theorem 1. *Given an object set O that is geographical partitioned into disjoint subsets $O = \bigcup_{1 \leq i \leq m} O_i$, the bounding border B_i of each O_i, and the segmentation of B_i into segments $b \in B_i$, it holds that $kNN(o, O) = \sum_i^m kNN(o, O_i \cup EC_i)$, $\forall o \in O_i$ if and only if $EC_i = EC_{B_i} = \bigcup_{b \in B_i} EC_b, \forall 1 \leq i \leq m$.*

Proof: Directly from Equation (3) and Lemma 3 □

4.3 Computational Cost of computeECB

The computational cost is directly affected by the replication factor $f_{Spitfire}$ achieved by *Spitfire*. Assume that the border B_i of area A_i is divided into $|B_i|$ equi-width border segments $b \in B_i$, with width d_b.

Lemma 4. *Given n objects, m servers, $|B|$ number of segments for each area border, and the optimal allocation of objects to the servers $\frac{n}{m}$, the time to compute the candidates EC_i for each sub-area is $O(|B| \cdot (\frac{n}{m} + klog\frac{n}{m}))$.*

Proof: Assuming optimal partitioning and an equal number of segments for each server, it holds that $|O_i| = \frac{n}{m}$ and $|B| = |B_i|$ for each s_i, respectively. In Algorithm 3 *computeECB* is invoked for each border segment $b \in B_i$ in order to compute the candidates EC_i. Determining $kNN(B_i, O_i)$ (Line 1) and θ_i (Line 3) has time complexity $O(\frac{n}{m} + klog\frac{n}{m})$. Scanning set O_i to determine EC_b using θ_b (Lines 4 - 8) has time complexity $O(\frac{n}{m})$. Therefore, each server spends $O(|B|(\frac{n}{m} + klog\frac{n}{m}))$ time to compute the candidates to be transmitted to its neighbors. □

Theorem 2. *Given n objects, m servers, parameter k, the perimeter P_A of area A, the length d_b of each border segments, and the optimal allocation of objects to the servers $\frac{n}{m}$, the time to compute the candidates EC_i for each sub-area is $O(\frac{P_A\sqrt{m}}{d_b}(\frac{n}{m} + klog\frac{n}{m}))$.*

Proof: In Lemma 4 we can replace the number of segments $|B|$ by the total length L over the length of the segments d_b as follows: $|B| = L/d_b$. The total length of all borders based on the *partition* algorithm is $L = \sqrt{m} * A_x + \sqrt{m} * A_y$, as each axis is partitioned $\sqrt{m}$ times. A_{axis} represents the length of area A along the given *axis*. Therefore, $|B| = \frac{P_A\sqrt{m}}{2d_b}$ □

4.4 Communication Cost of Replication

The computational cost is directly affected by the replication factor $f_{Spitfire}$, which is the cardinality of the External Candidate set EC_i for each server s_i (see Equation (1) in Section 3). Each EC_i consists of the k closest objects to its border B_i plus the objects alt_i whose *mindist* is smaller than θ_i, as described Section 3.3:

$$|EC_i| = k + |alt_i| \tag{4}$$

We can only analyze the replication factor f further if we make an assumption about the distribution of objects. Hereafter, we assume that the distribution is uniform. Further, w.l.o.g. we assume that we use border segments of the same diameter d_b to compose the borders between sub-areas.

Lemma 5. *Given a uniform distribution of n objects over area A, m servers with border segment diameter d_b, and an AkNN query, the alternative external candidate population is $|alt_i| \approx \frac{n}{A} \cdot (d_b + \sqrt{\frac{kA}{n\pi}})^2 - k$.*

Proof: The proof is omitted due to space limitation. □

Theorem 3. *Given a uniform distribution of n objects over area A, m servers with border segment diameter d_b, and an AkNN query, the replication factor is*

$$f_{Spitfire} \approx \frac{m}{A} \cdot (d_b + \sqrt{\frac{kA}{n\pi}})^2 + 1$$

Proof: Follows from Equations (1), (4) and Lemma 5. □

4.5 Optimal border segment size

Given a cluster setup (m), a dataset (A, n), the *CPU* speed of the servers, the *LAN* speed, an AkNN query (k) and the

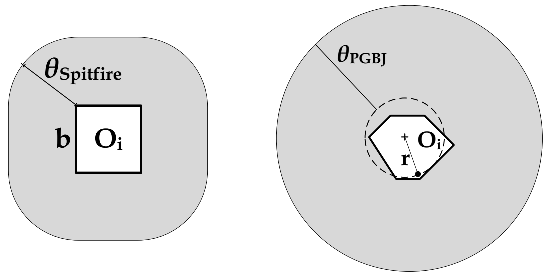


Fig. 5. Replication factor f in *Spitfire* (left) and PGBJ (right) shown as shaded areas in both figures.

border segment size d_b used in *Spitfire*, we can estimate the total response time as follows:

$$T = \frac{CPU \cdot P_A \cdot n}{d_b\sqrt{m}} + \frac{LAN \cdot n \cdot m}{A} \cdot (d_b + \sqrt{\frac{kA}{n\pi}})^2$$

Given that the only parameter we can fine-tune is the segment length d_b, we find the optimal value for d_b that minimizes the above equation as follows:

$$d_b = argmin_{d_b} T \tag{5}$$

4.6 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 5 to illustrate the discussion.

In *Spitfire*, the cutoff distance for the candidates $\theta_{Spitfire}$ (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, $\theta_{Spitfire} = d$. On the other 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}.

5 Experimental Evaluation

To validate our proposed ideas and evaluate *Spitfire*, we conduct a comprehensive set of experiments using a real testbed on which all presented algorithms have been implemented. We show the evaluation results of *Spitfire* in comparison with the state-of-the-art algorithms.

5.1 Experimental Testbed

Hardware: Our evaluation is carried out on the DMSL VCenter[3] IaaS datacenter, a private cloud, which encompasses

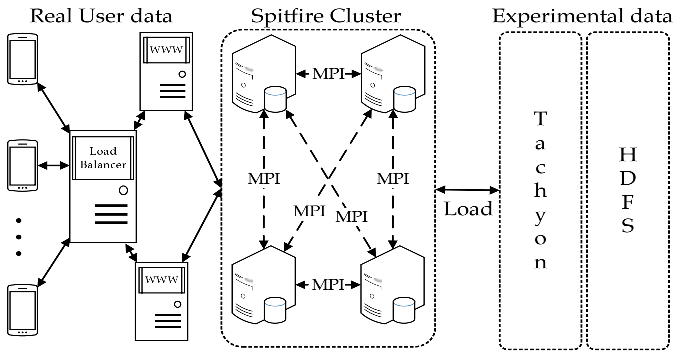


Fig. 6. Our Rayzit and experimental architecture.

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 on an IBM 3512 and are interconnected through a Gigabit network. The datacenter is managed through a VMWare vCenter Server 5.1 that connects to the respective VMWare ESXi 5.0.0 hosts.

Computing Nodes: The computing cluster, deployed over our VCenter IaaS, comprises of 9 Ubuntu 12.04 server images (i.e., denoted earlier as s_i), each featuring 8GB of RAM with 2 virtual CPUs (@ 2.40GHz). The images utilize fast local 10K RPM RAID-5 LSILogic SCSI disks, formatted with VMFS 5.54 (1MB block size). Each node features Hadoop v0.20.2 along with memory-centric distributed file system Tachyon v0.5.0. It also features the Parallel Java Library[4] to accommodate MPI [20] message passing in *Spitfire*.

Rayzit Service [7]: Our service, outlined in Section 1, features a HAProxy[5] HTTP load balancer to distribute the load to respective Apache HTTP servers (see Figure 6). Each server also features a Couchbase NoSQL document store[6] for storing the messages posted 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 Datasets

In our experiments we use the following synthetic, realistic and real datasets (depicted in Figure 7):

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 [2], including 5K vehicle trajectories in a 25km x 25km area of Oldenburg,

3. DMSL VCenter @ UCY. `http://goo.gl/dZfTE5`

4. Parallel Java. `http://goo.gl/uOQsDX`
5. HAProxy. `http://haproxy.1wt.eu/`
6. Couchbase. `http://www.couchbase.com/`
7. Rayzit API. `http://api.rayzit.com/`

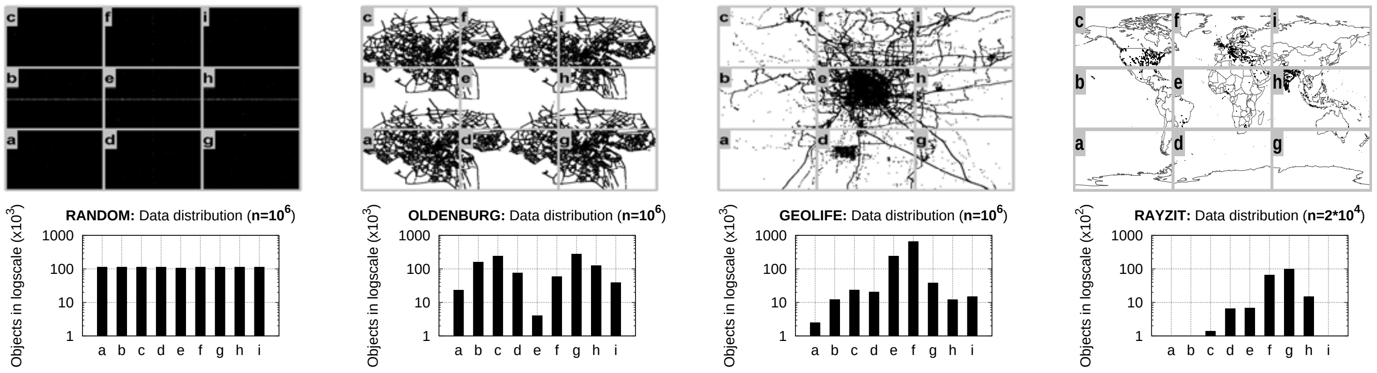


Fig. 7. Datasets (top row) and population histograms (bottom row) for an indicative 3x3 partitioning.

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 [30], 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.

Figure 7 (second row) shows the population histograms for the four respective datasets, when split into nine equi-width partitions. The standard deviation among the buckets for a total population of 1M objects is: i) 2K in Random; ii) 90K in Oldenburg; and iii) 200K in Geolife. For Rayzit, which has a population of 20K, the standard deviation is 3.3K.

5.3 Evaluated Algorithms

We compare one centralized and four distributed algorithms, which have been confirmed to generate identical correct results to the AkNN query.

Proximity [4]: 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 [28]: This is the two-phase MapReduce algorithm analyzed in Section 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 [28]: 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 [16]: This is the two-phase MapReduce algorithm analyzed in Section 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 this work. The only configuration parameter we use is the optimal border segment size d_b, which is derived with Equation (5), given the provided cluster of m nodes, the preference k, a dataset (A,n), the *CPU* speed of the servers and the *LAN* speed.

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 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).

5.4 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 2. A good algorithm is expected to have a low replication factor (when f=1 there is no replication of objects).

We also extend our presentation with additional *Network I/O (NI/O)* and Server Load Balancing measurements. Table 3 summarizes all parameters used in the experiments.

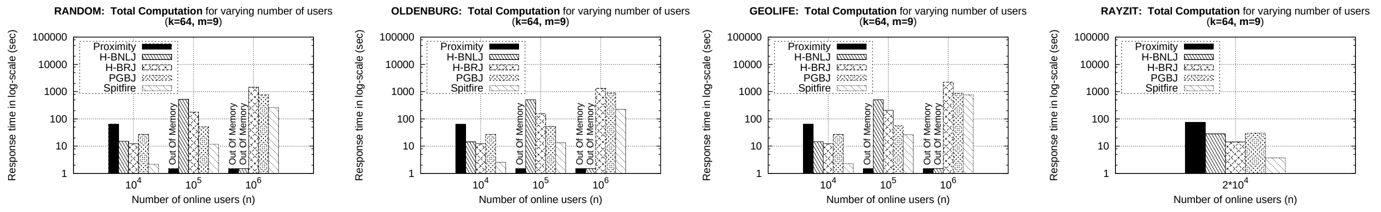


Fig. 8. AkNN query response time with increasing number of users. We compare the proposed *Spitfire* algorithm against the three state-of-the-art AkNN algorithms and a centralized algorithm on four datasets.

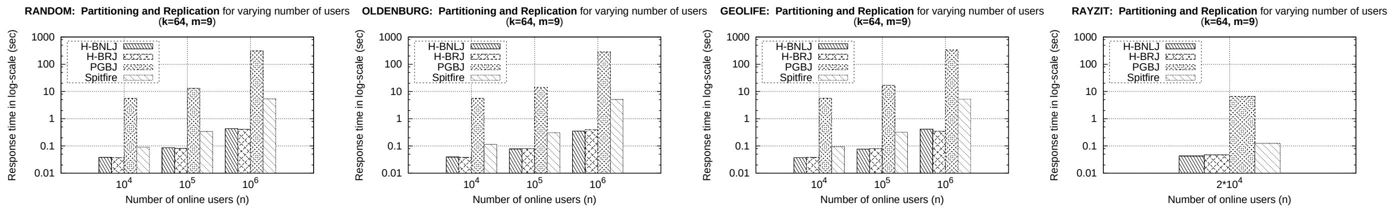


Fig. 9. Partitioning and Replication step response time with increasing number of users.

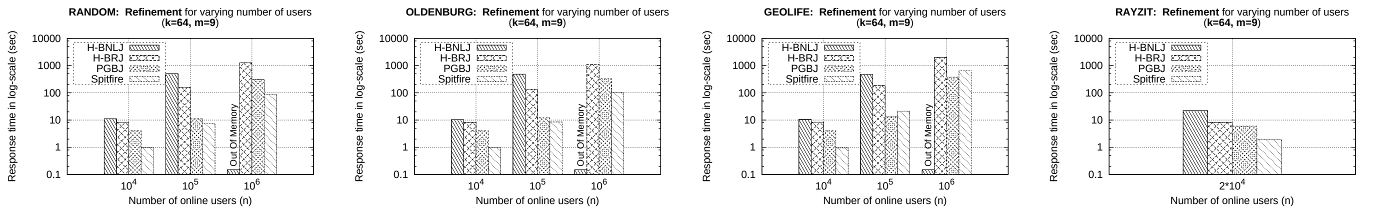


Fig. 10. Refinement step response time with increasing number of users and for each available dataset.

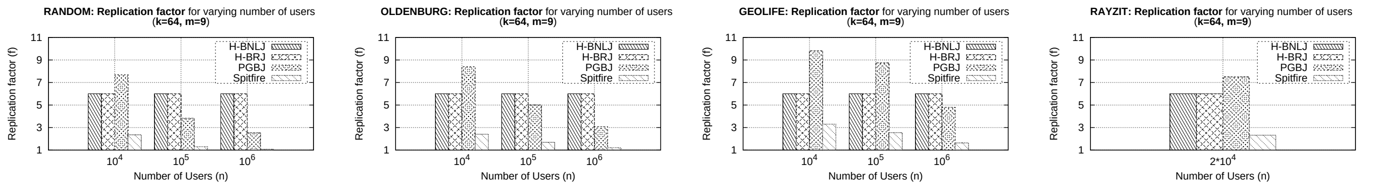


Fig. 11. Replication factor f with increasing number of users. The optimal value for f is 1, signifying no replication.

TABLE 3
Values used in our experiments

Section	Dataset	n	k	m
5.5	ALL	$[10^4, 10^5, 10^6]$	64	9
5.6	Random	10^6	64	9
5.7	ALL	10^6 (10^4 Rayzit)	64	9
5.8	Random, Rayzit	10^6 (10^4 Rayzit)	$4^i, 1 \leq i \leq 5$	9
5.9	Random	10^6	64	$[3, 6, 9]$

5.5 Varying Number of Users (n)

In this experimental series, we increase the workload of the system by growing the number of online users (n) exponentially and measure the response time and replication factor of the algorithms under evaluation.

Total Computation: In Figure 8, 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. Adding up the values shown in Figures 9 and 10 and comparing to the total response time in Figure 8, it becomes obvious that most of H-BRJ's response time is spent in communication, which is indicated theoretically by its communication complexity of $O(\sqrt{m}n)$ shown in Table 2. We focus on comparing only *Spitfire* and PGBJ for the rest of our evaluation.

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. The small percentage noted for the Geolife dataset is attributed to the fact that this dataset is highly skewed (as observed in Figure 7), and that PGBJ achieves better load balancing (as shown later in Section 5.7), which in turn leads to a faster refinement step.

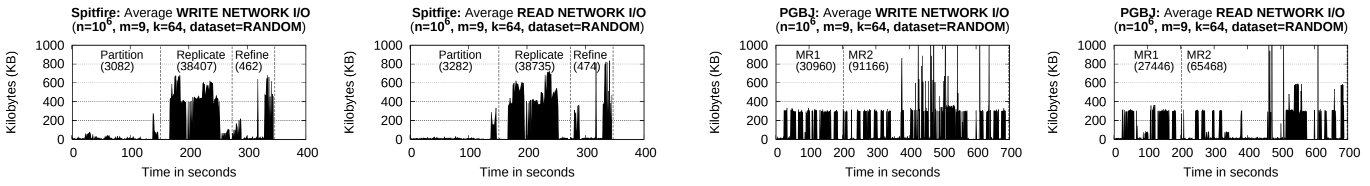


Fig. 12. Low level Network I/O (NI/O) measurements for *Spitfire* and PGBJ. *Spitfire* consumes 2.5x less NI/O.

Partitioning and Replication: In Figure 9 we measure the response time for the partitioning and replication steps in isolation. The theoretical time complexities, as presented in Table 2, confirm the outcomes: PGBJ is growing faster with the number of users n, while the other algorithms have only linear growth. These plots also show that the partitioning step of *Spitfire* features an important advantage: *speed*. *Spitfire* requires only $\sim$91 milliseconds, as opposed to PGBJ $\sim$263 milliseconds. In *Spitfire* we have opted for a much faster partitioning algorithm, even if that results in a slightly longer refinement process. Finally, it is also evident that the response time of these steps is independent of the dataset skewness.

Refinement: Figure 10 shows that the response time for the refinement step in PGBJ is independent of the dataset skewness, as opposed to *Spitfire*. Specifically, PGBJ achieves a response time of approximately 200 seconds for 10^6 users using any dataset. For the same amount of users *Spitfire* achieves a response time of 90, 100, or 800 seconds depending on the skewness of the dataset. The partitioning step in PGBJ is more sophisticated and produces a more even distribution. This means greater computational cost (Figure 9) but reduced response times for refinement (Figure 10) due to better load balancing. On the other hand, *Spitfire* strikes a better balance in these two steps, i.e., the much faster partitioning step makes up for the slower refinement step to achieve a much better overall performance.

Replication Factor: In Figure 11 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 3.3). As analyzed in Section 4.6, 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 2.4.

This experimental series demonstrates the algorithmic advantage that Spitfire offers, free from any effect that the implementation framework might add.

5.6 Network I/O Performance

We examine the underlying *Network I/O (NI/O)* activity taking place in PGBJ and *Spitfire* in order to better explain the results of Section 5.5. For brevity, we only present the Random dataset with $n{=}10^6$ online users, using $m{=}9$ servers and searching for $k{=}64$ NN. The other datasets produce similar results. We measured the Network I/O cost using *nmon*[8].

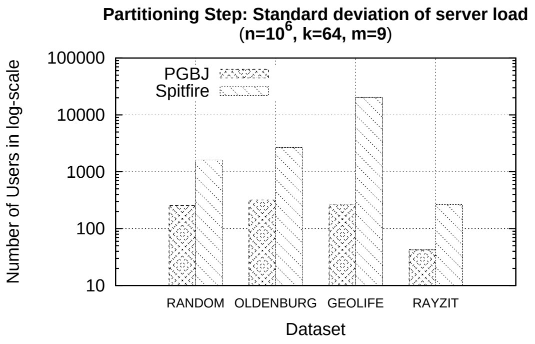


Fig. 13. Partitioning step: load balancing achieved (less is better). H-BNLJ and H-BRJ achieve optimal load balancing (standard deviation among server load $\approx$ 0).

Figure 12 shows that Spitfire features almost no NI/O in its partitioning step, while the respective step for PGBJ is quite intensive and lengthy. In fact, the total network traffic for PGBJ is 215 MB while for *Spitfire* it is only 84 MB. The above observations are compatible with our analysis, where we showed that f_{Spitfire} has a $\sqrt{2}$ advantage over f_{PGBJ} in the worst case. Here the advantage of Spitfire over PGBJ is even greater than $\sqrt{2}$ (i.e., 2.5x).

5.7 Partitioning and Load Balancing

In Section 5.5, we observe that for certain skewed datasets the competitive advantage of *Spitfire* over PGBJ is relatively small (e.g., in Geolife it is 14%). In this experimental series, we analyze in further depth the performance of the load balancing subroutines deployed in both PGBJ and *Spitfire*, respectively. Going back to our analysis in Section 2.4, we recall that PGBJ achieves a close to optimal partitioning using the $\sqrt{n}$ pivots, but at a higher computational cost. Here we experimentally validate these analytical findings.

Figure 13 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 ≈ 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.

8. nmon for Linux. `http://nmon.sourceforge.net/`

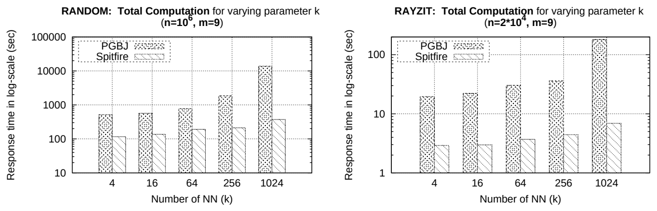


Fig. 14. The effect of k on response time.

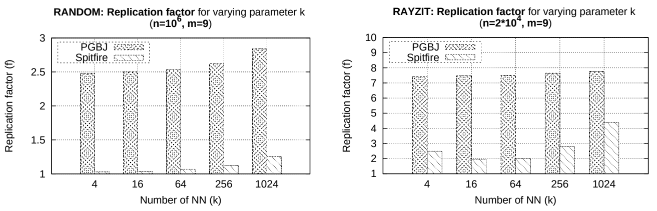


Fig. 15. The effect of k on the replication factor f.

5.8 Varying Number of Neighbors (k)

In this experiment, we exponentially increase the query parameter k by a factor of 4 and study its effect on the response time and the replication factor f of both *Spitfire* and PGBJ. We use the Random dataset of $n = 10^6$ online users and the $2{*}10^4$ Rayzit dataset. It is expected that an increasing k increases the workload for the distributed AkNN solutions, as the number of objects exchanged among servers is increased.

In Figure 14, we observe that *Spitfire* scales linearly with the increase in k for both datasets. This confirms our analytical result in Section 4, which shows *Spitfire*'s computational time and replication factor to be sub-linearly proportional to k. *Spitfire* is almost two orders of magnitude faster than PGBJ for $k = 1024$.

Figure 15 shows that the replication factor f of *Spitfire*, not only scales well with an increasing k, but also has a very low absolute value. Particularly, $f_{Spitfire}$ is less than 1.07 for $k \leq 64$, and it barely reaches 1.25 for $k = 1024$, showing more than a 95% improvement over f_{PGBJ}. This is one of the main reasons for the better response times exhibited by *Spitfire* in the previous experiments. Therefore, *Spitfire* outperforms PGBJ in scalability when the workload is increased by searching for more nearest neighbors.

5.9 Varying Number of Servers (m)

In this experiment we evaluate the effect that the number of servers (m) has on the response time and the replication factor of the distributed algorithms under evaluation.

In Figure 16 (left), we observe that with more servers *Spitfire* becomes faster than PGBJ, indicating that *Spitfire* utilizes the computational resources better than PGBJ.

Figure 16 (right) shows that the replication factor of *Spitfire* grows slightly faster than that of PGBJ. This experiment confirms Theorem 3 in Section 4, where $f_{Spitfire}$ is shown to increase as the number m of servers increases. Nevertheless, the absolute difference of the replication factor between *Spitfire* and PGBJ remains significantly large, making *Spitfire* the better choice. Comparing the two plots in Figure 16 it becomes

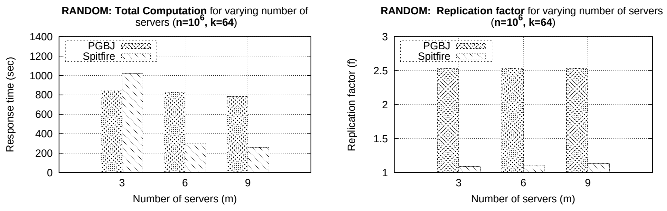


Fig. 16. The effect of m on response time and the replication factor f.

evident that the replication factor f increases slower than the performance gain with respect to the number of servers, a characteristic that proves the scalability of *Spitfire*.

6 Conclusions and Future Work

In this paper we present *Spitfire*, a scalable and high-performance distributed algorithm that solves the AkNN problem using a shared-nothing cloud infrastructure. Our 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.

In the future, we plan to study the temporal extensions to support more gracefully higher-rate AkNN scenarios with streaming data, as well as AkNN queries over high-dimensional data. We also plan to provide an approximate AkNN version of *Spitfire*. Finally, we are interested in developing online geographic hashing techniques at the network load-balancing level and also port our developments to general open-source large-scale data processing architectures (e.g., Apache Spark [27] and Apache Flink [11]). Finally, we intent to release our developments as an open-source project.

Acknowledgments: We would like to thank Lu et. al. (NUS, Singapore) for kindly providing the PGBJ [16] source code and Haoyuan Li (UC Berkeley, USA) for his assistance with Tachyon integration issues. This work was financially supported through an *Appcampus Award* by Microsoft, Nokia and Aalto University (Finland) as well as a industrial sponsorship by MTN (Cyprus). It has also been supported by the third author's startup grant at the Univ. of Cyprus, EU's FP7 *"Mobility, Data Mining, and Privacy"* project, EU's COST Action MOVE *"Knowledge Discovery for Moving Objects"*.

References

[1] 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.

[2] T. Brinkhoff. "A framework for generating network-based moving objects". *Geoinformatica*, Vol. 6, 153–180, 2002.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] E.H. Jacox and H. Samet. "Spatial join techniques". *ACM Transactions of Database Systems*, Vol. 32, 1–8, 2007.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] N. Nodarakis, E. Pitoura, S. Sioutas, . 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.

[20] P. Pacheco. "Parallel Programming with MPI". Morgan Kaufman, 1997.

[21] 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.

[22] 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.

[23] 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.

[24] P.M. Vaidya. "An o(n log n) algorithm for the all-nearest-neighbors problem". *Discrete Computational Geometry*, Vol. 4, 101–115, 1989.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

Georgios Chatzimilioudis received the BSc degree in Computer Science from the Aristotle University of Thessaloniki Greece, in 2004, the MSc and the PhD degrees in Computer Science and Engineering from University of California Riverside, in 2008 and 2010, respectively. He is a visiting Lecturer at the Department of Computer Science of the University of Cyprus and his research interests lie in the field of mobile crowdsourcing and public safety computing.

Costantinos Costa received the BSc and MSc degrees in computer science from the University of Cyprus, in 2011 and 2013, respectively. He is currently a PhD student at the Department of Computer Science, University of Cyprus. His research interests include databases and mobile computing, particularly distributed query processing for spatial and spatio-temporal datasets.

Demetrios Zeinalipour-Yazti received the BSc degree in Computer Science from the University of Cyprus, in 2000, the MSc and the PhD degrees in Computer Science and Engineering from University of California Riverside, in 2003 and 2005, respectively. He is an Assistant Professor at the Department of Computer Science of the University of Cyprus, where he leads the Data Management Systems Laboratory. He is a member of IEEE.

Wang-Chien Lee received the BSc degree from the Information Science Dept, National Chiao Tung University Taiwan, the MSc degree from the Computer Science Dept, Indiana University Bloomington, and the PhD degree from the Computer and Information Science Dept, Ohio State University. He is an Associate Professor of Computer Science and Eng. at Pennsylvania State University, leading the Pervasive Data Access research group. He is a member of IEEE.

Evaggelia Pitoura received the BSc degree in Computer Engineering from the University of Patras Greece, in 1990, the MSc and the PhD degrees in Computer Science from Purdue University, in 1993 and 1995, respectively. She is a Professor at the Department of Computer Science of the University of Ioannina Greece, where she leads the Distributed Data Management laboratory. She is a member of the IEEE Computer Society.