IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 5 July 2023, accepted 22 August 2023, date of publication 25 August 2023, date of current version 7 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3308684

RESEARCH ARTICLE

ParSCL: A Parallel and Distributed Framework to Process All Nearest Neighbor Queries on a Road Network

AAVASH BHANDARI[1], PRINCE HAMANDAWANA[1], MUHAMMAD ATTIQUE[2], HYUNG-JU CHO[3], AND TAE-SUN CHUNG[1]

[1]Department of Artificial Intelligence, Ajou University, Suwon-si 16499, South Korea
[2]Department of Software, Sejong University, Seoul 05006, South Korea
[3]Department of Software, Kyungpook National University, Sangju-si 37224, South Korea

Corresponding author: Tae-Sun Chung (tschung@ajou.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development under Grant IITP-2023-RS-2023-00255968, and in part by the Information Technology Research Center (ITRC) Support Program funded by the Korean Government (MSIT) under Grant IITP-2021-0-02051. The work of Hyung-Ju Cho was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education under Grant NRF-2020R1I1A3052713.

ABSTRACT The proliferation of current and next-generation mobile and sensing devices has increased at an alarming rate. With these state-of-the-art devices, the global positioning system (*GPS*) has made remote sensing and location tracking more viable. One such query is the All Nearest Neighbor (*ANN*) query, which extracts and returns all data objects that are in close vicinity to all query objects. An *ANN* is a combination of k-nearest neighbors (*kNN*), and join queries. Hence, *ANN* has useful for applications in different domains such as transportation optimization, locating safe zones, and ride-sharing. An example of its applications is, "*find the nearest gas station for each car parking lot*". Because these applications are responsible for generating a massive number of query requests, a large amount of computation is required to return these query requests. As a single machine cannot meet this demand in this study, we propose a distributed query processing framework to process *ANN* queries using the Apache Spark framework. In an empirical study, our proposed framework achieved superior query efficiency and scalability compared to other methods and design alternatives.

INDEX TERMS All nearest neighbor queries, distributed and parallel processing, spatial query processing.

I. INTRODUCTION

The rapid evolution of smartphone and sensor technologies has enabled users to access the world in their hands. In particular, location-based services (*LBSs*) such as maps, and point-of-interest (*POIs*) recommendations have become an integral part of daily life. For instance, we use navigation applications to find the nearest restaurants or cafes or use ride-sharing applications to reach our destinations. The use of these *LBS* applications results in the collection of enormous amounts of location data known as spatial data, and the corresponding queries are known as spatial queries [1]. The most common instances of spatial queries are linked to *LBS* incorporate shortest path queries [2], [3], range queries [4], [5], k-nearest neighbor queries [6], [7], reverse k-NN queries [8], [9], keyword queries [10], [11], and preference queries [12], [13]. Location data usually consist of user query requests. Such large numbers of query requests must be accepted, analyzed, and evaluated efficiently.

Albeit, efficient algorithms for processing *ANN* exist for centralized data- bases [14], [15], [16], [17], [18], it is vital to provide a more efficient, distributed, and parallel framework that scales well with the increasing number of queries, where the computation is shared among multiple available servers.

The associate editor coordinating the review of this manuscript and approving it for publication was Thomas Canhao Xu.

The main goal of this study is to design and implement a robust distributed and parallel-processing *ANN* architecture that provides high scalability and performance for spatial query processing. We refer to *ANN* queries as variants of an exact k-nearest neighbor in a spatial network, and k is always equal to one. As *ANN* queries require the processing of a large number of *NN* query requests, evaluating a large number of query requests requires effective distribution of data into a suitable number of partitions and scheduling them across several machines so that the overall query is executed within a reasonable amount of time. Thus, we propose an algorithm called parallel standard clustered loops (*parSCL*), a parallel algorithm that involves shared execution and incorporates Apache Spark for effective and efficient parallel processing of *ANN* queries in spatial road networks. Solving *ANN* queries in a distributed environment requires that the road network graph G be partitioned into p sub-graphs. Each sub-graph is then distributed to p executors, each of which is assigned a local computation task.

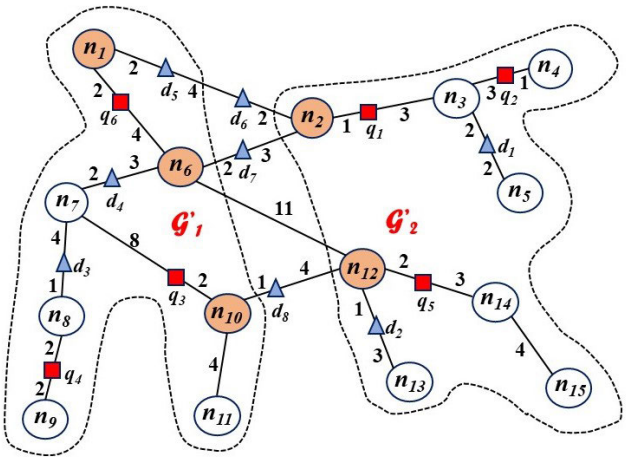


FIGURE 1. The road network graph is partitioned into two sub-graphs. *ANN* query is performed in each sub-graph. The orange colored circle represents the boundary nodes.

To facilitate local computation among different executors and guarantee the accuracy of the final result, clusters of executors are required to compute distances across partition boundaries for the data object that exists on the next side of the boundary; in any case, it might be the nearest neighbor. This can be explained further by an example, as depicted in Figure 1, which is partitioned into the two sub-graphs, G'_1 and G'_2, that are distributed among two executors. There are two types of objects, query, and data objects, which are denoted by red squares and blue triangles, respectively. We assume that we must find an *NN* for each query object. By looking at the example the *NN*s of q_1, q_2, q_3, q_4, q_5, and q_6 are d_6, d_1, d_8, d_3, d_2, and d_5, respectively. Because d_6 and d_8 do not reside in the same partition, q_1 and q_3 must communicate with G'_1 and G'_2 to request accurate *NN* data objects. This motivated us to propose *parSCL*, which is an efficient algorithm that enables the synchronization of the spatial query of objects across partitions to obtain accurate *NN* objects. *parSCL* is an Apache Spark [19] based framework that supports parallel execution of *ANN* queries. Apache Spark is an open-source, in-memory distributed big-data processing framework with fault tolerance. This provides a data structure known as a resilient distributed dataset (*RDD*). An *RDD* is an immutable collection of elements and can operate in parallel with a low-level API. Each *RDD* is created by parallelizing existing collections using operations such as (*map*, *filter*, *reduce*). The primary contributions of this study are as follows:

- We propose a parallel and distributed framework *parSCL* to efficiently process *ANN* queries on road networks. To the best of our knowledge, this is the first attempt to evaluate *ANN* queries in distributed environment settings.
- The proposed *parSCL* method is simple and easy to implement. From the generated sub-graphs, utilizing a pre-computed distance and shared execution techniques can reduce communication between workers.
- Extensive experiments with various settings are performed to measure the superiority of the proposed scheme for particular scenarios.

The remainder of this paper is organized as follows. Section II reviews the related works, and preliminary findings are presented in Section III. In Section IV, we explain the processing steps of the proposed framework. The theoretical analysis of the proposed algorithm is discussed in Section IV-G. The results of a detailed experimental evaluation and the discussions are presented in Sections V and VI. Finally, the paper is concluded with future research directions in Section VII.

II. RELATED WORKS

Most conventional location-based spatial queries concentrate on discovering the spatial proximity of data objects to query objects. Papadias et al. [20] proposed Incremental Euclidean Restriction (*IER*) and Incremental Network Expansion (*INE*). Both *IER* and *INE* use multi-step *kNN* that can operate in high dimensions. *IER* works based on the A* search algorithm, and *INE* expands the search region until the closest data point is identified. To overcome the storage costs incurred when processing large-scale road networks Samet et al. [21] proposed a spatially induced linkage cognizance *SILC* framework. The *SILC* uses a precomputation scheme and quadtrees to find possible pairs of objects and store the identified shortest paths. However, SILC uses a precise distance and has storage overhead. Lee et al. [22] introduced ROAD, in which this issue is addressed by using search-space pruning.

A spatial join query maintains a set of object pairs that is, each object in the R dataset has its pair in the S dataset, which satisfies some spatial conditions, for example, distance. Brinkhoff et al. [23] presented an early study on spatial joins using R-trees. The spatial join query was further extended by Mamoulis and Papadias [24] to a multiway

spatial join query, that combines a sequence of pairwise joins under the condition that the object is closer rather than overlappings. Xia et al. [25] suggested Gorder, which is a block-nested loop based on grid partitioning that utilizes techniques such as sorting, join scheduling, and distance computation filtering to reduce both CPU and I/O overheads. Yao et al. [26] proposed *z-KNN*, which allows the conversion of multi-dimensional data points into linear points without tampering with the database engine.

Graph analysis and computations play vital roles in various fields. Processing *ANN* queries require performing one *NN* query on d in the data object for each q of the query objects. Clarkson [27] presented a main-memory-based randomized technique in which the sizes of both data and query objects are identical and are often found in the same dataset. Vaidya [28] presented an optimal algorithm based on variants of kd-trees and utilized the box split method to build a box-split tree with time complexity of $\mathcal{O}(mnlogn)$. Zhang et al. [14] suggested two-phase hash-based algorithms using spatial hashing, which initially imported a pair of data and query objects, and divided them into a bucket of equal size. Overlapping buckets are used to identify the nearest neighbor of each query object. However, the skewness of the data distribution has a large impact on the performance of the algorithm. Xu et al. [17] suggested VIVET, which computes all nearest neighbor queries with a single traversal. It is an index-based algorithm that traverses a graph network from a virtual node to all the other existing nodes. An array table holds the nearest-neighbor result for all nodes. Bhandari et al. [18] proposed an algorithm named SCL based on a shared-execution technique aiming to reduce the run-time by reducing redundant query searches. However, these techniques aim to support efficient *ANN* processing in a centralized manner.

Chen and Patel [15] proposed an algorithm based on an R-tree or R*-tree, that traverses index trees in a depth-first manner by expanding the candidate search nodes bidirectionally. With this approach, each query object is often required to traverse more than one path in the tree structure and access redundant partitions, which results in computational overhead. A distributed algorithm called Spitfire, proposed by Chatzimilioudis et al. [29] performed centralized hash-based partitioning to divide the search space. Each node then calculates a set of candidate neighbors within each split and distributes them among the computing nodes to compute the local k-nearest neighbors. Zhang et al. [30] suggested the use of a Map-Reduce framework to perform *kNN* joins that utilize z-values to convert multi-dimensional points into a single dimension and employs a random shift to maintain spatial locality. This method produces approximate results; however, the post-processing step incurs additional overhead. Lu et al. [31] proposed a method for performing k-nearest neighbor join operations using Map-Reduce. Their approach involved efficiently mapping objects into groups by leveraging pruning rules to reduce the number of data replicas, resulting in improved overall performance. Luo et al. [32] presented a Hadoop MapReduce-based framework to process distributed group keyword queries called *DISKs*. It also assumes that vertices and edges also have spatial location information besides the graph connecting structure. Eldawy and Mokbel [33] focused on developing a MapReduce framework called *SpatialHadoop*. This framework includes a set of spatial index structures and provides built-in support for Hadoop distributed file systems (*HDFS*). Additionally, it supports various types of spatial queries, making it a versatile tool for processing large-scale spatial data sets in distributed environments. Yokoyama et al. [34] proposed a method that involved computing all k-nearest neighbor queries in Hadoop by splitting the search space into smaller cells based on the target attributes. This approach effectively decomposes the search space into smaller units that can be processed simultaneously in parallel. Moutafis et al. [35] suggested an approach that includes several key improvements over existing approaches, including improved partitioning techniques for the dataset, accelerated local processing using the plane-sweep reducers, and reduced network overhead. Although existing techniques have been proposed to answer various types of spatial queries in distributed environments, they cannot be directly applied to evaluate *ANN* queries. This is because the algorithm used to evaluate the *ANN* queries differs in several ways from those used in previous studies. To the best of our knowledge, this is the first attempt to develop a distributed method for evaluating *ANN* queries to address the unique challenges posed by this type of query.

III. PRELIMINARIES

1) ROAD NETWORK

In this study, a road network graph is formally denoted as G, which is an undirected and weighted graph. The graph comprises nodes, edges, and weights. The edge represents the road segments, and weight is the value assigned to that edge. The formal definition of a road network is as follows:

Definition 1 (Road Network): A road network graph $G = \langle N, E, W \rangle$ *is an undirected and weighted graph, where* N *is a set of nodes,* E *is a set of edges, and* W *denotes the edge weights respectively.*

The distance between two nodes n_i and n_j can be denoted either as $dist(n_i, n_j)$ or $dist(n_j, n_i)$, which is the sum of the weights in a path. The *length* of a path is the sum of all the weights of the involved edges on the path. The *shortest path* distance between the two nodes n_i and n_j, denoted by $SPF(n_i, n_j)$, is the path with the smallest distance between n_i and n_j. The notation used in this study is summarized in Table 1.

2) ALL-NEAREST NEIGHBOR QUERIES

Formally, given two datasets Q and D, each tuple $q \in Q$ and $d \in D$ is interpreted as a query and data object respectively. The shortest distance between q and d is the minimum number of edges that must be traversed in the network

TABLE 1. Notations and their meanings.

Notation	Meaning
$G = \{N, E, W\}$	Graph network with nodes, edges, and weights.
n_i	Node $n \in N$.
e_{ij}	Edge $e_{ij} \in E$.
$dist(n_i, n_j)$	Edge weight or distance.
q	Query object $q \in Q$.
d	Data object $d \in D$.
$dist(q, d)$	Shortest path distance between the data object and the query object.
$len(q, d)$	Distance of the segment connecting q and d such that they lie in the same segment.
D^q	Set of data objects closest to query objects q
$G' = \left\{G'_1, G'_2, ..., G'_p\right\}$	Set of sub-graphs from G.
$B\left(G'_i\right)$	Set of boundary nodes of sub-graphs.
E_b	Set of boundary edges.
n_i^b	boundary node $\in B(G_i)$
$E_b\left(n_i^b, n_j^b\right)$	boundary edge $\in E_b$.
$SPList\left(B_{spf}\right)$	Set of shortest-paths list between all boundary nodes.
$G_{emb} = \{N_e, E_e, W\}$	Embedded Graph Network.
$DA\left[G_{emb}\right]$	Distance array table.

to reach d from q - $dist\,(q, d)$. Subsequently, $ANN(Q, D)$ returns one nearest d for each q. In general, the *ANN* query in a road network returns a data object for each query object within the closest proximity.

Definition 2 (ANN Query): Given two different object sets Q and D, where $Q = \{q_1, q_2, \ldots, q_n\}$ and $D = \{d_1, d_2, \ldots, d_m\}$, the ANN *query returns a data object d for each query object q such that d is in the closest proximity of q.*

$$Q \ltimes D = \{\langle q, d\rangle \mid \forall q \in Q, \forall\, d \in D^q\}$$

IV. DESIGN AND DEVELOPMENT

A. ROAD NETWORK PARTITIONING

Since our distributed algorithm relies on graph partitioning, the first step is to divide the road network graph into p equal parts with minimum edge cuts. The graph partitioning problem can be defined as follows: Given a road network graph *G=(N, E, W)* where N is the set of nodes, E is the set of edges e_{ij} connecting node i and node j, with $|N| = n$ and $|E| = m$, a graph partitioning algorithm divides N into the union of p disjoint subsets, $N_1, N_2, \ldots, N_p : N_i \cap N_j = \phi$ for $i \neq j$ from which a set of sub-graphs $G' = \left\{G'_1, G'_1, \ldots, G'_p\right\}$ is created. Each sub-graph holds $G'_i = \{N_i, E_i\}$.

The boundary edge E_b is the set of edge cuts, that is, the set of all edges for which the source and destination nodes are in two different partitions, and represents the amount of inter-process communication. Node $n_i \in B_i$ is a *boundary node* of B_i if there exists a connecting edge $\left(n_i, n_j\right) \in E_b$. The set of boundary nodes is denoted by $B\left(G'_i\right)$.

Since choosing optimal graph partitioning is NP-hard, and obtaining an approximate solution is computationally uncontrollable, only heuristics can be adopted to solve the problem. However, the problem of road network graph partitioning is beyond the scope of this study; thus, we adopted an open-source graph-partitioning algorithm, *METIS* [36] developed in the Karypis laboratory to partition the graph network. *METIS* provides a set of serial programs for partitioning graphs, and its algorithms are based on multilevel paradigms. It has been adopted in other spatial-network query processing studies [32], [37] and has shown very good results; thus it was adopted in our study.

Given a road network graph, *METIS* requires the number of partitions k as an input parameter, followed by coarsening of the graph and computation of each partition to ensure a minimal number of edge cuts. The output includes n number of lines, and each line of the file represents a vertex in the original graph and specifies the partition index to which that vertex has been assigned.

B. BOUNDARY NODES AND EDGES

Once the result is obtained after graph partitioning, the next step is to identify the boundary nodes and edges. For each partition G'_i, we define the *boundary nodes* and *boundary edges*, which consist of the nodes and edges of G and are usually shared nodes between two or more sub-graphs.

Definition 3 (Boundary Node): Node $n_i^b \in N$ is a boundary node if and only if $\exists G'_i, G'_j$ such that $n_i^b \in N_i \cap N_j$ $(i \neq j \,\forall\, i, j \in [1, p])$.

Any path from a non-boundary node in G'_i to a non-boundary node in G'_j must pass through one or more boundary nodes because these boundary nodes are the only "*bridging point*" between the sub-graphs.

Definition 4 (Boundary Edge): An edge $E_b\left(n_i, n_j\right) \in E$ is a boundary edge if it cuts graph G into a sub-graph and its removal results in a disconnected graph.

Figure 1 depicts a road network partitioned into two parts: $G' = \left\{G'_1, G'_2\right\}$. The boundary nodes are represented by orange-filled circular shapes. For example, $B\left(G'_1\right) = \{1, 6, 10\}$ and $B\left(G'_2\right) = \{2, 12\}$ are the boundary nodes of sub-graphs G'_1 and G'_2, respectively. Four boundary edges exist: $E_b = \{(1, 2), (6, 2), (6, 12), (10, 12)\}$.

C. EMBEDDED GRAPH NETWORK

From each boundary node of a partitioned road network, we interpret the embedded graph network G_{emb} that consists of the union of the boundary nodes and their shortest paths to other boundary nodes that lies on different partitions.

Definition 5 (Embedded Graph): Let G *be an undirected graph of the road network and,* p *be the number of partitions that yields $G' = \left\{G'_1, G'_2, \ldots, G'_p\right\}$ partitions. The embedded graph network $G_{emb} = (N_e, E_e, W)$ is a sub-graph of G, where N_e, E_e denotes nodes and edges of the sub-graph consisting of:*

$$N_e = \bigcup_{G'} \left[B\left(G'\right) \cup \left\{i | (i, j) \in SP\left(n_i^b, n_j^b\right) \wedge n_i^b, n_j^b \in B\left(G'\right)\right\}\right]$$

$$E_e = E_b \cup \bigcup_{G'} \left\{(i, j) \mid (i, j) \in SP\left(n_i^b, n_j^b\right)\right\}$$

Figure 2 shows the steps for creating an embedded network from boundary nodes. The initial step is to partition

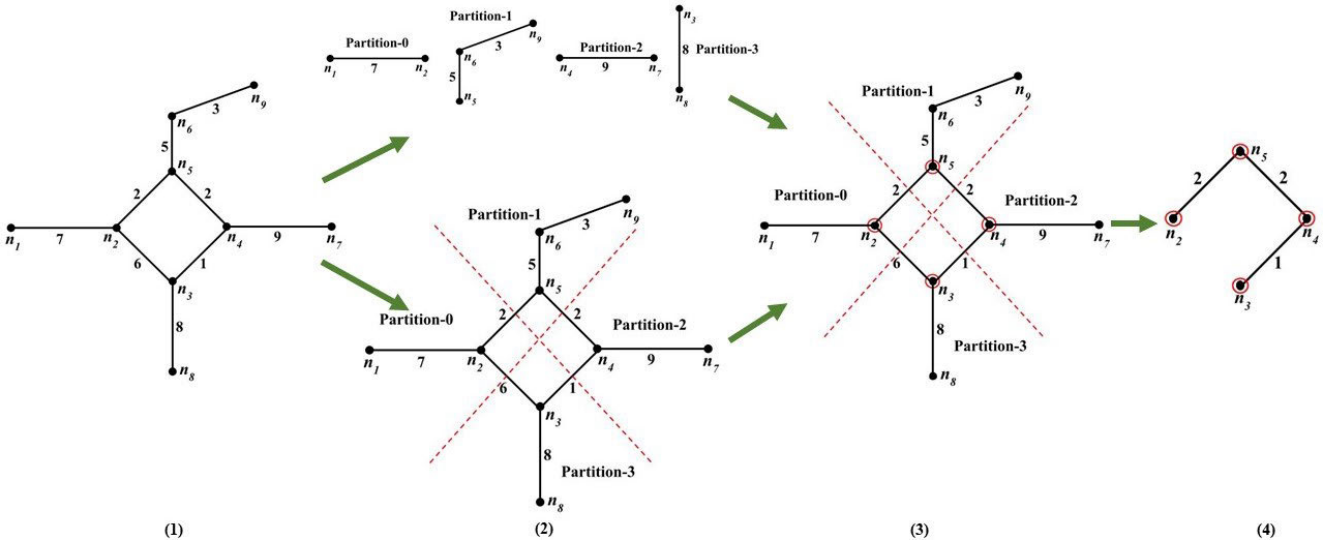


FIGURE 2. Steps showing the construction of an embedded graph from an original graph network.

the graph network into sub-graphs, as represented in Figure 2(1) and (2). To identify boundary nodes, the shortest path algorithm is executed from each boundary node to another boundary node. The final step is to unify all shortest paths and create an embedded network, as shown in Figure 2(4). The embedded network consists solely of boundary nodes. Any non-boundary nodes in the shortest path are omitted while considering only their distances.

Lemma 1: Let G *be a graph of the road network, p be the number of partitions that gives:* $G' = \left\{G'_1, G'_2, \ldots, G'_p\right\}$ *partitions, and an embedded network* G_{emb}*. The shortest path,* $SP\left(n_i^b, n_j^b\right)$*, between any combination of boundary nodes* $n_i^b \in B\left(G'_i\right)$ *and* $n_j^b \in B\left(G'_j\right)$ *with* $G'_i \neq G'_j$ *is in* G_{emb}*.*

Proof: The correctness of Lemma 1 is proven by contradiction. Assume that $e_{ij} = (i, j)$ is an edge in E such that $e_{ij} \in SP\left(n_i^b, n_j^b\right)$, but $e_{ij} \notin G_{emb}$. Since all the edges connecting the sub-graphs are in the embedded network G_{emb}, e_{ij} cannot be a linking edge; but it must be located in some sub-graph $G'_i \in G'$ joined by a shortest path $SP\left(n_i^b, n_j^b\right)$. The only means of travelling through G'_i is to pass through a boundary node $n_i^b \in G'_i$ and enter through another boundary node $n_j^b \in G'_j$. This creates a sub-path $\left(n_i^b \rightarrow n_j^b\right)$, which is a sub-set of the shortest path between n_j^b and n_j^b, i.e., $\left(n_i^b \rightarrow n_j^b\right) \subseteq SP\left(n_i^b, n_j^b\right)$ that contains e_{ij} and is the shortest path between n_j^b and n_j^b. From the definition 5, G_{emb} is composed of the shortest paths between all boundary nodes of all sub-graphs and, consequently including $\left(n_i^b \rightarrow n_j^b\right)$. Since e_{ij} is in $\left(n_i^b \rightarrow n_j^b\right)$, it is also in G_{emb}. ■

D. DISTANCE ARRAY TABLE

Once an embedded graph is created, we create a virtual node n^* that connects every boundary node in G_{emb}. The edge formed after connecting n^* to every n_i^b is a directed edge $\overrightarrow{e_{n^* n_i^b}}$ with a weight of zero. A directed edge ensures that repetitive graph and network traversal overlaps are avoided. Afterward, we run Dijkstra's algorithm starting from the virtual node n^* to find the nearest data object d to every boundary vertex n_i^b. After finding the closest data object, the result is stored in an array table.

Definition 6 (D-ART): For an embedded graph, we define a Distance-Array Table *(D-ART). The D-ART of the embedded graph* G_{emb} *stores the shortest distance from each boundary node* n_i^b *to the nearest data object, d.*

$$DA\left(G_{emb}\right) = \left\langle n_i^b, d, dist\left(n_i^b, d\right)\right\rangle \mid n_i^b \in B\left(G_i\right)$$

Lemma 2: Given an embedded graph G_{emb} *and a virtual node* n^**, for every data object* $d \in D$*, there must be only one boundary node* $n_i^b \in B\left(G'_i\right)$ *on the shortest path from* n^* *to d.*

Proof: The correctness of lemma 2 is proven by non-contradiction. Since G_{emb} is a fully connected graph from the boundary nodes, there exists a path that links n^* to d. Because, n^* is connected only to the n_i^b, any shortest path from n^* to d must pass through at least one n_i^b. ■

For instance, in Figure 3, the shortest path between n^* and d_5 includes only one boundary node n_1^b with path $\left(n^* \rightarrow n_1^b \rightarrow d_5\right)$.

Figure 3 shows the embedded graph network of our running example of Figure 1. It comprises all boundary nodes and their respective shortest paths to the other boundary nodes. We use n^* as a virtual node connecting each boundary node in G_{emb}. The directed edges generated after connecting n^* to each boundary node are $\left\{\left(\overrightarrow{e_{n^*,n_1^b}}\right), \left(\overrightarrow{e_{n^*,n_2^b}}\right), \left(\overrightarrow{e_{n^*,n_6^b}}\right), \left(\overrightarrow{e_{n^*,n_{10}^b}}\right), \left(\overrightarrow{e_{n^*,n_{12}^b}}\right)\right\}$ with an edge weight set to 0. To compute the nearest neighbors, any single-source shortest-path algorithm can be used. In our study, Dijkstra's algorithm was implemented for simplicity. Dijkstra's algorithm starts traversing from the virtual

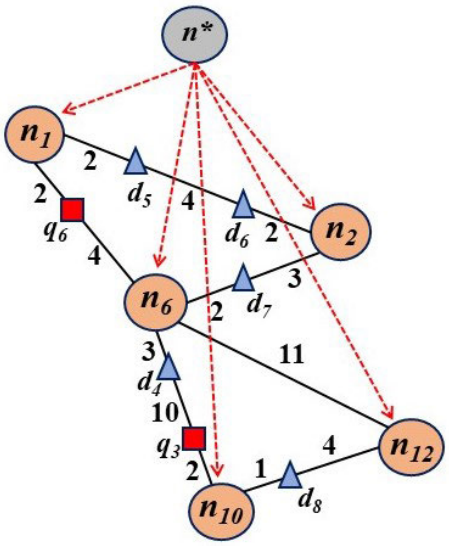


FIGURE 3. An embedded graph network with a virtual node.

TABLE 2. D-ART of the embedded graph G_{emb}.

	n_1^b	n_2^b	n_6^b	n_{10}^b	n_{12}^b
NN	d_5	d_6	d_7	d_8	d_8
distance	2	2	2	1	4

node n^* to each boundary node. Once the data object closest to the boundary node is computed, it is stored in a distance-array table. Table 2 lists the $DA\,[G_{emb}]$ for the example in Figure 1.

Algorithm 1 shows the steps involved in generating G_{emb}. The initial step was to partition the given input road network graph using *METIS*. Afterward, from the p-partitioned output file, the boundary, and non-boundary nodes are located, and the boundary edges are extracted. The shortest-path query from each boundary node in $n_i^b \in B\left(G_i'\right)$ to another boundary node $n_j^b \in B\left(G_j'\right)$ is computed in Step-1. Each shortest path is stored in $SPList\left(B_{spf}\right)$, as shown in Lines 1–4. Step-2 included the union of all shortest paths from the $SPList\left(B_{spf}\right)$. To create new nodes and edges for G_{emb} from each element of $SPList\left(B_{spf}\right)$, the first and the last elements are extracted along with the distances as shown in Step 8–17. A newly created graph G_{emb} with the new n and e is returned in Step-18, and it is stored in the master/driver node.

Algorithm 2 summarizes the D-ART table computation procedure. Once G_{emb} is created, the process of finding the nearest data object d and filling it in the D-ART is performed. The algorithm initially creates a *virtual node* $n*$ that is linked to each boundary node n_i^b with a weight of zero in Lines 1–5. Then, it initializes an array of $DA\,[G_{emb}]$ of size $|N|$ to store the nearest data object d_i and the distance in Lines 6–10. Here, the graph traversal starts. The priority queue PQ facilitates network traversal. Each element in PQ is the boundary node $n_i^b \in B\left(G_i'\right)$ to be visited. Initializing the traversal requires that PQ and, n^* are enqueued

Algorithm 1 Embedded Graph Construction

Input : $G = \langle N, E, W\rangle$: graph network, $B\,(G_i)$: Set of boundary nodes, E_b : Set of boundary edges
Output: G_{emb} : Embedded Graph Network

1 **Step-1**: Compute the shortest path from each boundary node to another boundary node
2 **foreach** *boundary node* $n_i^b \in B\,(G_i)$ **do**
3 $\quad SPList\left(B_{spf}\right) \leftarrow$ *YenAlgorithm.runSP* $(G, srcVertex, destVertex)$
4 **end**
5 **return** SPList $\left(B_{sp}\right)$
6 **Step-2**: Create an Embedded network by unifying the shortest-path list
7 $G_{emb} \leftarrow \emptyset$; `// Create a new empty graph`
8 **foreach** $e_{ij} \in E_b$ **do**
9 $\quad G_{emb} \leftarrow addEdge\left(e_{ij}\right)$
10 **end**
11 **foreach** shortest-path $\left(B_{sp}\right) \in$ SPList $\left(B_{sp}\right)$ **do**
12 $\quad$ **if** $e_{ij} \in E_b$ **then**
13 $\quad\quad$ **if** $n_i^b.partitionId = n_j^b.partitionId$ **then**
14 $\quad\quad\quad G_{emb} \leftarrow addEdge\left(e_{ij}\right)$
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 **end**
18 **return** G_{emb}

in Lines 11 and 12. A loop is kept to dequeue the nodes from PQ in Lines 13–26. When a node SN_i is dequeued and visited for the first time, the adjacent nodes SN_j to SN_i are inserted into PQ in Lines 14–16. For each edge $\overrightarrow{e_{SN_i,SN_{ij}}}$, if a data object d_i is in the shortest path rather than the existing shortest path to a data object d_j, the distance is updated to the nearest data object d_i of SNi in Lines 17-26. Once PQ is empty, all the nodes are visited and their nearest data objects are computed and stored in the D-ART table. The $DA[G_{emb}]$ is returned, and the algorithm terminates Line 28.

E. QUERY PROCEDURE

The query procedure includes two steps: local *ANN* computation and global *ANN* computation. The master node accumulates the partition information, query requests, and location information. It then creates an *RDD* from the gathered information. Subsequently, the *RDD*s are distributed to each executor which is responsible for computing the local *ANN* utilizing the approach presented in [18]. The local computed results are returned to the master node.

Once the D-ART is computed, the global *ANN* is processed, which we call the *merge* process. Each boundary node n_i^b, finds the nearest query object and compares the distance that is, the local *ANN* results, with the D-ART table result. We compare the distance between the query object and its nearest neighbor in a local *ANN* with the distance between the query object and its nearest boundary vertex in

Algorithm 2 Distance-Array Table Construction

Input : G_{emb} : Embedded Graph, D : set of Data objects
Output: $DA[G_{emb}]$

Create a virtual node n^*;
foreach $n_i^b \in B\left(G_i'\right)$ **do**
  Create a virtual edge $\overrightarrow{e_{n^*,n_i^b}}$; // from the virtual node to boundary node
  $dist\left(\overrightarrow{e_{n^*,n_i^b}}\right) = 0$;
end
Initialize an array $DA[G_{emb}]$ with size $|N|$;
foreach $n_i^b \in B\left(G_i'\right)$ **do**
  $G_{emb}[n_i^b].nndistance = 0$;
  $G_{emb}[n_i^b].nnId = d_i.id$;
end
Initialize a Priority Queue PQ as $\emptyset$;
$PQ \leftarrow$ enqueue n^*;
while PQ $\neq \emptyset$ **do**
  $SN_i = PQ$.deque ; // taking the first element from the PQ
  if SN_i *has not been visited before then* **then**
    foreach *adjacent node* SN_j *of* SN_i *that has not been visited before* **do**
      if $\overrightarrow{e_{SN_i,SN_{ij}}}$ *has data object* **then**
        if $G_{emb}[n_i^b].nndistance > G_{emb}[n_i^b].nndistance + d_i.getDistFromStart$ **then**
          $G_{emb}[n_i^b].nndistance = G_{emb}[n_i^b].nndistance + d_i.getDistFromStart$;
          $G_{emb}[n_i^b].nnId = d_i.getObjectId$;
          $PQ \leftarrow SN_j$;
        end
      end
    end
    mark SN_i as visited;
  end
end
return DA$[G_{emb}]$;

the D-ART table. If the distance to a neighbor in the local *ANN* is greater, we update the result for q. The main aim is to find the closest match for query object q.

$$dist\left(q, d_i\right) > len\left(q, n_i^b\right) + dist\left(n_i^b, d_j\right) \tag{1}$$

From Equation 1, we update the result of q.

Continuing with the example shown in Figure 1, there are six query objects $(q_1, q_2, q_3, q_4, q_5, q_6)$ annotated by the red square boxes. In addition, there are eight data objects $(d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8)$. Let us consider that, the size of p is two; hence, the partitioning algorithm partitions the given input graph into two balanced partitions G_1', G_2' with minimum edge cuts. From the obtained sub-graphs,

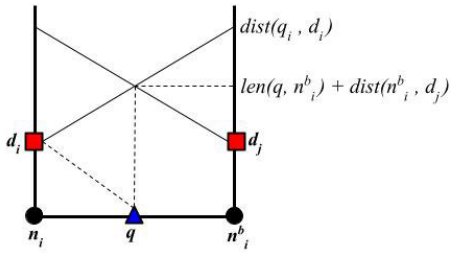


FIGURE 4. Global ANN process for comparing the distance with the DART.

the boundary nodes, $B\left(G_1'\right) = \{1, 6, 10\}$ and $B\left(G_2'\right) = \{2, 12\}$, and boundary edges $E_b = \{(1, 2), (6, 2), (10, 12)\}$ are identified. From Algorithms 1 and 2, an embedded graph G_{emb} and a distance array table are created, as shown in Table 2. Next, the main driver distributes the RDD to each worker for computational purposes. The local ANN result set from both sub-graphs returns $\{(q_6, d_5), (q_4, d_3), (q_3, d_4), (q_1, d_1), (q_2, d_1), (q_5, d_2)\}$. For q_1, $dist\left(q_1, n_2^b\right) + dist\left(n_2^b, d_6\right) < dist\left(q_1, d_1\right)$; hence, the nearest data object for q_1 is d_6 because $dist\left(q_1, n_2^b\right) + dist\left(n_2^b, d_6\right) = 3$. Similarly, the nearest neighbor for q_3 is d_8, respectively.

1) PRUNING HEURISTICS

While the approach outlined above may be successful in identifying the nearest match to a query object, it can also be computationally intensive. To overcome this challenge, we propose the use of a heuristic pruning technique that enhances the effectiveness of the global ANN computation phase and increases its efficiency. To optimize the merging process, we integrate an approximate range query θ for each boundary node, which facilitates the comparison of query objects located only within the specified range. Thus, we reduced the computational burden and significantly enhanced the merging step performance. To calculate the optimal value of θ for each boundary node, we employed Equation 2, which considers the maximum distance from the boundary node to its adjacent nodes $dist_{max}\left(n_{i_{adj}}^b\right)$ and the degree of the boundary node (δ). By incorporating these parameters into the calculation of θ, we determine an appropriate range that encompasses all potential query objects relevant to the specific boundary node, thereby smoothing the merging process and improving its overall efficiency.

$$\theta = \left[\frac{dist_{max}\left(n_{i_{adj}}^b\right)}{\delta}\right] + dist\left(n_i^b, d_j\right) \tag{2}$$

F. FRAMEWORK DESIGN

In the initial phase, *parSCL* partitions the given input road network graph into p sub-graphs and computes the following data structures: *boundary vertices* and *boundary edges*. The shortest path between each boundary node is determined to identify each component. Afterward, an embedded

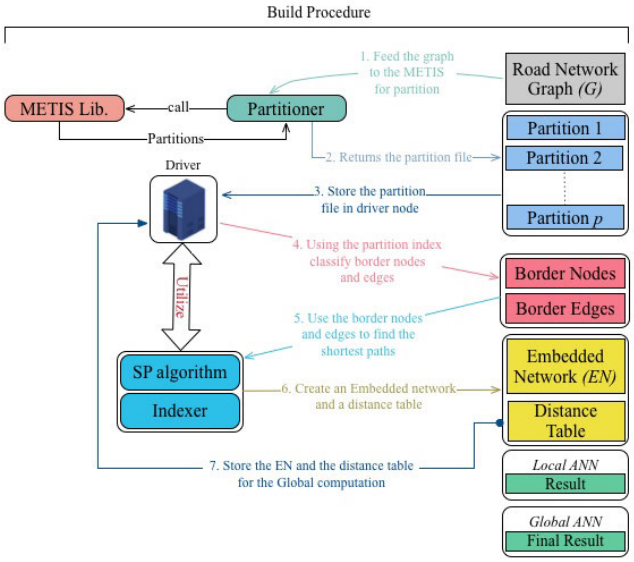


FIGURE 5. ***parSCL*** **build procedure. SP refers to the shortest path.**

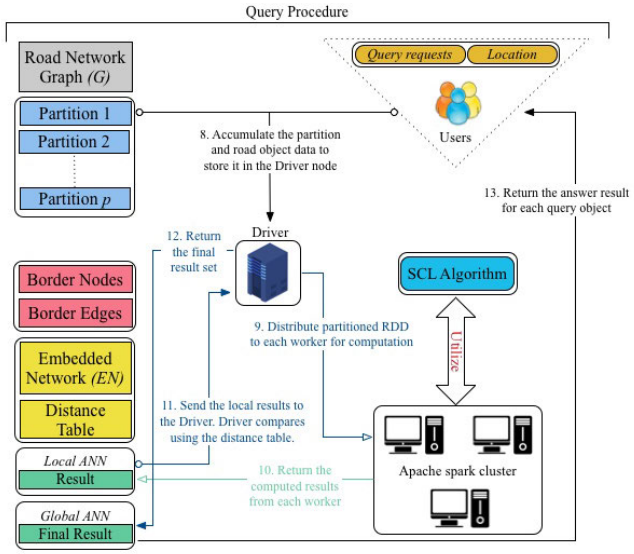


FIGURE 6. ***parSCL*** **Query Procedure. SCL refers to Standard-Clustered Loops.**

network (G_{emb}) is created, that, for every boundary node, stores the distance from it to the nearest data object d. The aforementioned structure is used for the global computation of the ANN once the executors complete the local ANN computation. Figures 5 and 6 depicts the building and querying procedures for the *parSCL* framework.

1) BUILD PROCEDURE

The build procedure of *parSCL* algorithm returns the embedded graph that is created by extracting a subset of the original graph that includes only the boundary nodes and boundary edges. This simplified the problem of graph embedding while preserving the connectivity and distance information of the original graph. By contrast, the distance array table contains the shortest path distances between the boundary nodes and data objects. The components that go into building an embedded graph are as follows:

- Graph partitioning is an important step in the *parSCL* algorithm that, divides the input graph into smaller sub-graphs that can be processed independently. This allows the algorithm to take advantage of parallel processing to speed up computation. The build process of *parSCL* uses the *METIS* algorithm and the partitioner method to partition the graph into balanced p-partitions. Once the graph has been partitioned, the output is further processed to extract the boundary nodes and edges.
- After the *METIS* algorithm partitions the graph, the output file typically contains one line for each vertex in the graph. Each line specifies the partition index to which the corresponding vertex is assigned. The driver program then creates a tuple for each vertex, which includes the vertex information and partition index. These tuples are stored in the driver program and they allow the driver to keep track of which vertices belong to which partitions.
- The driver program creates a resilient distributed dataset (*RDD*) of the tuples, which includes the vertex information, edge information, and the partition index. To identify the boundary nodes and edges, the driver program performs a simple comparison between the partition keys of each vertex for each edge. If the start and end nodes have the same partition key, they belong to the same sub-graph. However, if the start and end nodes have different partition keys, then they belong to different partitions; thus, the edge is a boundary edge.
- After identifying the boundary nodes and edges, the driver then proceeds to execute the shortest path (SP) algorithm, which computes the shortest path between each pair of boundary nodes. The SP algorithm employs an implementation [38] based on Yen's algorithm [39] and, then uses an indexer to sort and select the paths based on their minimum distance. The shortest returned paths are unified to create an embedded graph, that preserves connectivity and distance information of the origin.
- Once the embedded graph is created, the distance array table is computed and filled. The distance table is an array that stores the precomputed distances from each boundary node of the embedded graph to the nearest data object. Using the pre-computed distances stored in the distance table, the query processing stage can run significantly faster.

2) QUERY PROCEDURE

- To simulate the query request submitted by the users, random road objects are generated along with their positions in the graph. Then the query request is mapped with the RDD containing the partition information. Furthermore, the driver distributes tasks among the executors in the cluster.

- An Apache Spark cluster typically consists of a set of worker nodes, which run the executors, and a master node, which manages the overall execution of the tasks. Each executor is a separate process that runs on a worker node and is responsible for executing the tasks assigned by the driver program. When creating a cluster, it is important to specify the configuration of the executor nodes to ensure the availability of the resources necessary to perform the allocated tasks. The *SCL* algorithm is executed on each executor, and the local nearest neighbor search is performed.
- Subsequently, the local nearest neighbor results once computed are then returned to the driver program. The driver program uses the embedded graph and distance array table to perform global nearest-neighbor merging. During the merging step, the driver selects the query objects near each boundary node and compares them with the results from the distance array table to update the results. This process is repeated until all query objects have been processed and their nearest neighbors have been found. The final results are returned to the user.

G. PERFORMANCE ANALYSIS

In a distributed setting, the communication cost incurred during the intercommunication process causes overhead. However, the embedded graph and D-ART table that we created were relatively small. Let, p be the number of partitions, $|d|$ the number of data objects, $|n^b|$ the boundary nodes and $|E|$ be the number of edges. Once the embedded graph is generated after unifying all the shortest path pairs between the boundary nodes, the shortest path edges are represented as $SP[E]$. Creating an embedded graph takes $\mathcal{O}\left(n^b + |SP[E]|\right)$ time. The time required to compute the nearest data object for each boundary node in G_{emb} is determined by the time of the single-source shortest path algorithm. We employed Dijkstra's shortest path algorithm, which has a time-complexity of $\mathcal{O}\left(\left(|SP[E]| + |n^b|\right) \log |n^b|\right)$. The size of the D-ART table is linear with respect to the number of boundary nodes; that is, $\mathcal{O}\left(|n^b|\right)$.

V. EXPERIMENTAL EVALUATION

In this section, we study the performance of the proposed algorithm. Furthermore, the section is divided into parts, in which information about the datasets used and the environment setup followed by the experimental results are presented.

A. EXPERIMENTAL SETUP

1) DATASETS

We used three real-world datasets for the evaluation where CAL (*California*), OLDEN (*Oldenburg*), and SANJ (*SanJoaquin*) were downloaded from [40]. This dataset contains a road network in which, the California dataset consists of 21,048 nodes and 21,693 edges, the San-Joaquin dataset consists of 18,263 nodes and 23,784 edges, and the Oldenburg dataset consists of 6105 nodes and 7035 edges. To minimize communication costs, we partitioned each graph into p node-disjoint sub-graphs using the open-source algorithm *METIS* [36] for balanced partitions.

2) EXPERIMENTAL SETTINGS

The *parSCL* algorithm proposed in this study was utilized to identify the nearest neighbor *NN* queries within the datasets. The experiment involved altering the sizes of the data and query objects while maintaining a static size for the query object when the data object size increased, and vice versa. Table 3 lists the parameters employed in the experiment, with bold values representing the default settings used throughout the study. To generate random objects, ten centroid datasets were produced, adhering to a Gaussian Distribution with the centroid set as the mean and a standard deviation of 1% of the side length. The distribution of the data and query objects followed a centroid distribution unless otherwise specified.

TABLE 3. Parameters.

Parameters	Settings
Partitions	2, 3, 4, 5, 6, 7
Number of data objects	2, 3, **5**, 7, 10 $(\times 10^4)$
Number of query objects	2, 3, **5**, 7, 10 $(\times 10^4)$
Distribution of query objects	(**C**)entroid, (U)niform
Distribution of data objects	(**C**)entroid, (U)niform
Real-world roadmaps	CAL, SANJ, OLDEN

To evaluate the effectiveness of the proposed scheme, we utilized the *INE* [20] algorithm and a centralized algorithm *SCL* [18] to compute the *ANN* queries across all n query objects. It is worth noting that the *SCL* algorithm was originally developed for centralized computing and lacks the necessary framework to support parallel execution. Furthermore, given the lack of existing implementations of the *INE* algorithm in distributed settings, we made several modifications to it to ensure compatibility with the parallel architecture. During the global merging process, each boundary node traverses and finds the query object in each sub-graph and compares the nearest distance to another subgraph. This allows the *INE* algorithm to achieve optimal performance in the parallel environment while maintaining the accuracy of the *ANN* queries. The three algorithms were implemented using the Java programming language and executed on a distributed Google Cloud Dataproc cluster. The cluster utilizes eight nodes, with one acting as the main master and the other seven as executors, each equipped with Ubuntu 18.04 64-bit OS, two vCPUs, and 8GB of RAM. Apache Spark 2.3.4 is pre-installed on the Dataproc cluster. The experiments were repeated five times each, and the average values were recorded for analysis.

B. EXPERIMENTAL RESULTS

1) PRECOMPUTATION TIME

In our study, the pre-computation time refers to the duration required to create an embedded graph G_{emb} and a

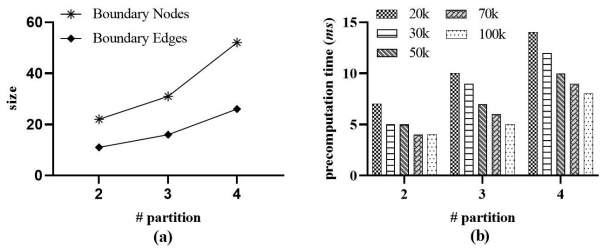


FIGURE 7. Precomputation of *parSCL* algorithm for CAL.

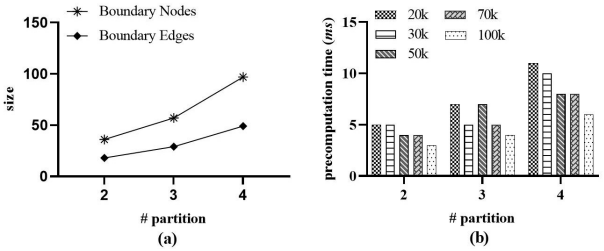


FIGURE 9. Precomputation of *parSCL* algorithm for OLDEN.

DART table. It should be noted that the precomputation time is influenced by both the number of boundary nodes and edges and the size of the data object. Figure 7(a) shows the identified number of boundary nodes and edges for the CAL dataset: 22 boundary nodes and 11 boundary edges were found when the partition size was set to 2; 31 boundary nodes and 16 boundary edges were found when the partition size was set to 3; and 52 boundary nodes and 26 boundary edges were found when the partition size was set to 4. The pre-computation time for the CAL dataset for partition sizes of two, three, and four is shown in Figure 7(b). The results indicate that pre-computation time increases as partition size increases and decreases as the data item size increases for each partition.

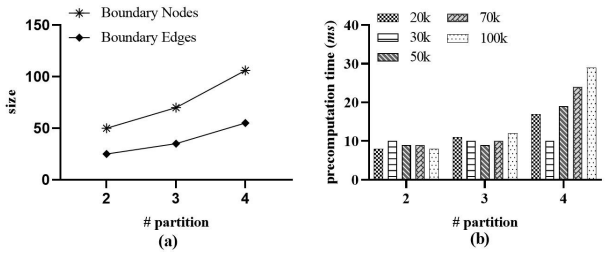


FIGURE 8. Precomputation of *parSCL* algorithm for SANJ.

Figure 8(a) illustrates that there were 50 boundary nodes and 25 boundary edges were identified for the two partitions, whereas three and four partitions resulted in 70 boundary nodes and 35 boundary edges, and 106 boundary nodes and 55 boundary edges, respectively, for the SANJ dataset. Figure 8(b) shows that the precomputation time increased as the size of the data object increased, with partition four taking almost 30 seconds when the data object size reached 100k.

Finally, for the OLDEN dataset, Figure 9(a) shows that increasing the partition from two to four resulted in the identification of 36 boundary nodes, 18 boundary edges, 57 boundary nodes, 29 boundary edges, 97 boundary nodes, and 49 boundary edges. We observe that the pre-computation time trend was similar to that observed for the CAL dataset in Figure 9(b).

2) PERFORMANCE EVALUATION

The Figures 10–12 compare the query processing times for three different datasets: CAL, SANJ, and OLDEN. The comparison was done using three different algorithms: distributed *INE* that is *distINE*, *parSCL*, and centralized *SCL*. Initially, the size of the query object was set to 50k by default, whereas the size of the data object varied from 20k to 100k and vice-versa when the size of the query object varied. The figure also demonstrates the response of the algorithms to different data distribution combinations, including $((U, U), (U, C), (C, U), (C, C))$ for both the query and data objects. The notation used per algorithm in this study is as follows. The hyphenated number following the algorithm name represents the number of partitions used. For example, *distINE-2* indicates that the *distINE* algorithm was executed using two partitions.

Figure 10(a) illustrates the impact of the data object size on the query processing time for CAL. The query processing time also tended to increase as the number of data objects increased. However, this effect was less pronounced for the *parSCL* algorithm. In fact, when the task was divided among four working executors, *parSCL* outperformed the distINE algorithm by a factor of five. This suggests that the performance of *parSCL* improves as the number of executors increases, and the pruning heuristics reduce the inter-communication process during the merging process. By contrast, *distINE* requires additional time during the global merge step.

Figure 10(b) shows a comparison of the processing times of the *distINE* and *parSCL* algorithms as the query object size increases. As expected, the processing time increased as the number of query objects increased because more nearest-neighbor evaluations are required, which incurred a higher computational cost. *parSCL* performs well when the query object size is 20k and 30k, and the processing time increases moderately as the query object size increases. However, *parSCL* outperforms *distINE* by processing queries three times faster. In addition, the processing time of *distINE* with a partition size of two worsens as the query object size increased.

Figure 10(c) illustrates the impact of different data distribution combinations on the query processing of the *parSCL* and *distINE* algorithms. The results demonstrate that *parSCL* consistently outperforms *distINE* under various distribution settings. Specifically, when both the query and data objects followed a (U, U) distribution, *distINE* performance is slower because of the sparse distribution of objects that

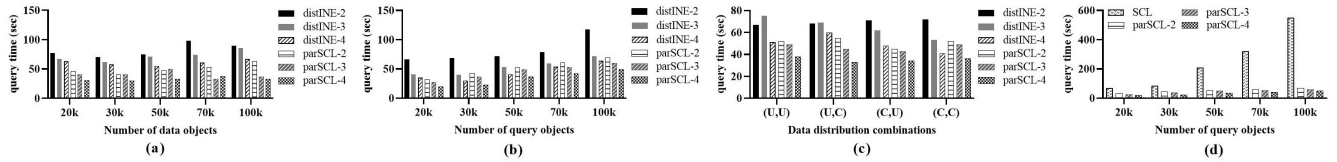


FIGURE 10. Performance evaluation of *distINE* and *parSCL* for CAL: (a) varying D; (b) varying Q; (c) varying the data distribution combinations; (d) *SCL* vs. *parSCL*.

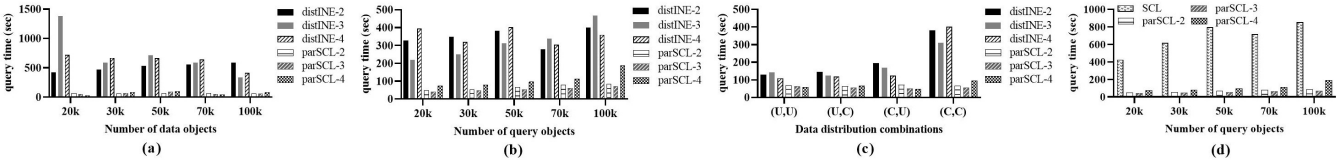


FIGURE 11. Performance evaluation of *distINE* and *parSCL* for SANJ: (a) varying D; (b) varying Q; (c) varying the data distribution combinations; (d) *SCL* vs. *parSCL*.

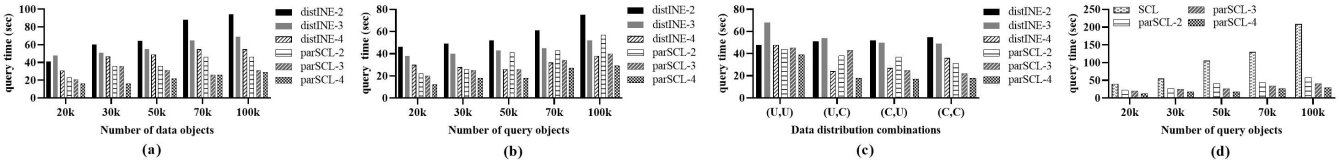


FIGURE 12. Performance evaluation of *distINE* and *parSCL* for OLDEN: (a) varying D; (b) varying Q; (c) varying the data distribution combinations; (d) *SCL* vs. *parSCL*.

are far from each other. In summary, this study suggests that *parSCL* performs better than *distINE* under a wide range of data distribution scenarios.

Figure 10(d) shows a comparison of the centralized *SCL* and *parSCL* algorithms. Both algorithms aim to reduce the computational cost by utilizing the shared-execution techniques; however, the *parSCL* algorithm outperformed centralized *SCL* by an order of magnitude as the query object size increased to 100k. This study suggests that optimizing techniques in a distributed environment can significantly improve algorithm efficiency, making *parSCL* a more efficient algorithm for reducing computational costs.

Figure 11(a) illustrates the comparison of the *distINE* and *parSCL* query processing time concerning the size of the data objects for SANJ. As expected, both algorithms exhibited an increment in the query processing time with an increase in the size of the data objects. However, the *parSCL* performance was less affected by the increase in data object size, owing to its efficient pruning heuristics and reduced intercommunication cost during the merging process. The *parSCL* outperformed *distINE* by a significant factor of seven, on average, with *distINE* suffering poorly at the 20k size of the data object. With fewer data objects, the query object requires redundant traversal across the network, making the global merging step computationally expensive.

Figure 11(b) compares the performances of *distINE* and *parSCL* in terms of the number of *NN* queries that must be evaluated. The *parSCL* algorithm leverages materialized results and shared execution techniques, thereby drastically reducing the number of *NN* queries to be evaluated. Consequently, *parSCL* performed up to seven times faster than *distINE*. With an increase in the number of query objects, both algorithms were required to compute and compare the *NN* results for each query object. At 100k, *distINE* with three partitions required 468 seconds to finish the query computation, including the intercommunication process during the global merging step.

Figure 11(c) shows that the *parSCL* algorithm performs well for all the tested combinations of distributions. While *distINE* performs well when objects followed (U, U), (U, C), (C, U) distributions, its performance degrades significantly when both objects followed (C, C) distributions. This is because *distINE* exerts a redundant traversal within subgraphs, and the additional merging step incurs additional computational overhead. By contrast, the *parSCL* pruning heuristics made them more efficient, enabling them to perform well across all tested distributions. Figure 11(d) depicts a comparison between the centralized *SCL* and *parSCL* algorithms. As depicted in the figure, emparSCL exhibits superior performance, outperforming the centralized *SCL* algorithm. The underlying reason for this considerable difference in performance is associated with the fact that the centralized *SCL* algorithm processes the computation on a single machine, whereas *parSCL* performs distributed computations using multiple executors. Consequently, the parallel processing ability of *parSCL* enables it to process data much faster than

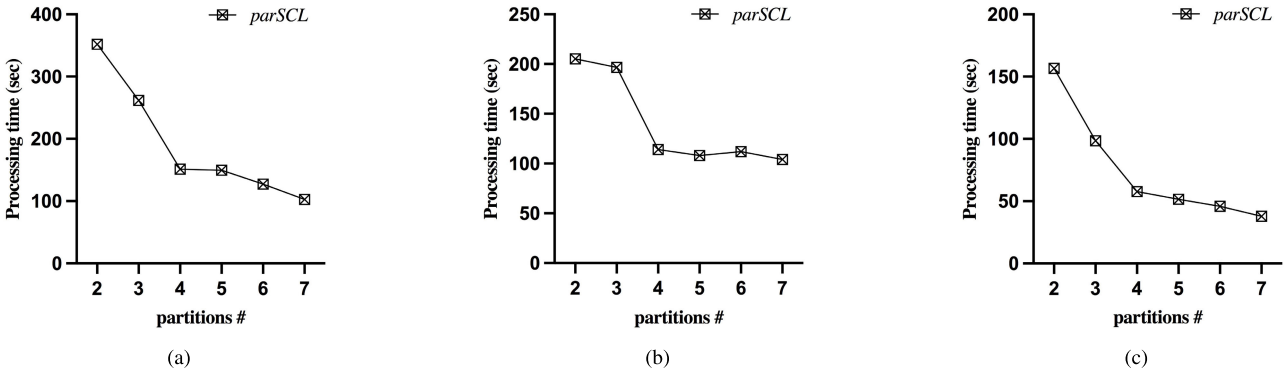


FIGURE 13. Scaling *parSCL*: Impact on Processing Time for (a) CAL, (b) SANJ, and (c) OLDEN.

the centralized *SCL* algorithm. This result further highlights the scalability of the *parSCL* algorithm because increasing the number of partitions reduces the processing time by approximately eight times.

The effect of data object size on the performance of *distINE* and *parSCL* for OLDEN is shown in Figure 12(a). It can be observed that *parSCL* performance is less affected by the size of the data objects. This result is consistent with the trend that is shown in Figure 10(a). On the other hand, *distINE* at partition two had higher computational costs during the *NN* evaluations on average.

Figure 12(b) illustrates the impact of increasing the number of query objects on the performance of *distINE* and *parSCL*. As $|Q|$ increases, the number of *NN* query evaluations increases; however, *parSCL* can reduce the computation required for *NN* queries. It can be inferred that when the number of increases, *distINE* and *parSCL* exhibit similar performances. Nonetheless, *parSCL* is approximately twice as fast as *distINE*.

Figure 12(c) compares the query processing times of the *distINE* and *parSCL* algorithms under different data distribution scenarios. Across all the distribution combinations, *parSCL* outperformed *distINE*. Specifically, for (U, U) data and query object distributions, both algorithms achieved a similar query processing time of around 45 seconds for various partition sizes. However, for the three partitions, both algorithms exhibited a trend similar to that shown in Figure 10(c), with *parSCL* continuing to perform better than *distINE*.

In Figure 12(d), the performance of the centralized *SCL* algorithm is compared with that of *parSCL*. The results clearly reveal that *parSCL* outperforms the centralized *SCL* algorithm in terms of processing time. Further, *parSCL* significantly outperforms the centralized *SCL* algorithm, with up to a five-fold reduction in processing time. It can be inferred that parallel processing is a highly effective approach for reducing the computation time and achieving significant performance improvement, especially when dealing with a large volume of query objects.

Figure 13 demonstrates the performance of *parSCL* algorithm with varying numbers of partitions for all three datasets. The size of the query and data objects was set to 100k following the (C, C) distribution which was then fed to *parSCL* with a different number of executors. The results show a significant reduction in query processing cost as the number of executors increases, indicating the horizontal scaling capability of the algorithm.

VI. DISCUSSIONS

Our study focused on the distributed processing of static *ANN* queries in an undirected road network. We studied two popular big data processing tools, Hadoop and Spark, and found that Spark was better suited to our requirements because of its in-memory processing capabilities. Furthermore, we implemented a pre-computation step using the VIVET [17] technique to optimize the performance of the proposed algorithm. However, we restricted this step to the shortest paths between the boundary nodes to create an efficient embedded graph. We included only the union of the boundary nodes in the embedded graph, and any non-boundary nodes in the shortest path were logically omitted, allowing us to use the distances between the boundary nodes to create a complete path. Our experiments with the SANJ dataset yielded some unexpected results when the *distINE* algorithm was used. As shown in Figure 11(a), even with three partitions, *distINE* was impacted by a relatively small number of data objects. We suspect that this is due to the high number of intersection nodes in the SANJ dataset, which lead to a large number of traversals. In addition, we observed that the *distINE* approach for performing nearest neighbor evaluation for each query object resulted in increased computation time during the global merge process. Although we cannot fully explain these results, they highlight the need for further exploration and optimization. In Figure 11(d), it appears that the *distINE* algorithm performs better than the centralized *SCL* algorithm, especially when the number of query objects is large. Compared to the centralized *SCL* algorithm, the *distINE* algorithm has the advantage of leveraging the parallel processing power of multiple machines, improving its scalability and performance. However, it is still important to note the limitations and challenges of *distINE*, such as its sensitivity to the

structure of the road networks and the potential impact of high intersection density on its performance.

In addition, limitations may be encountered with this approach when increasing the number of partitions, as this can cause extra inter-communication overhead and affect performance. However, our proposed parSCL algorithm overcomes these challenges through its efficient design. Figure 13 illustrates the scalability of *parSCL* algorithm. It can be inferred that the algorithm exhibits robust scalability, as demonstrated by the consistent processing time across the various executor configurations. Concerning scalability issues, our solution effectively avoids bottlenecks, especially when all executors possess similar processing power. Each executor independently computes partial results in parallel, ensuring minimal variations in the time taken to compute and send these partial results back to the master node. Once these partial results are aggregated at the master node, the subsequent merging process remains consistently efficient, regardless of the number of executors involved.

Finally, it is important to note that our current study is limited to static road networks and does not consider changes in edge weights over time. It also assumes that query and data locations remain fixed during the request period. In the future, we plan to explore methods for evaluating *ANN* queries in dynamic road networks and incorporate geo-social keyword queries to enable finding the nearest neighbors based on keywords.

VII. CONCLUSION

The problem of processing *ANN* queries on road networks is experienced in many applications, given the large volume of query workloads that require new scalable solutions. To address this challenge, we proposed *parSCL*, a parallel and distributed algorithm that extends the centralized *SCL* algorithm to adapt to distributed environments. Distributed processing offers significant cost benefits compared with centralized processing. To improve the efficiency and effectiveness of the algorithm, we adopted a pre-computed table that stores the nearest data object for every boundary node, and the global merge step uses pruning heuristic techniques to reduce computation overheads. We evaluated *parSCL* performance using simulation experiments on real-world road network maps based on various parameters. The results demonstrate that *parSCL* outperforms other algorithms in cases involving a large number of query objects in a road network. Our future work will include exploring how to evaluate *ANN* queries in dynamic road networks, where the weight of the road network changes frequently. We also plan to investigate the use of geo-social keyword queries together with *ANN* queries to find the nearest neighbors for each query based on their keywords.

REFERENCES

[1] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee, "Location-based spatial queries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2003, pp. 443–454.

[2] L. Wu, X. Xiao, D. Deng, G. Cong, A. Diwen Zhu, and S. Zhou, "Shortest path and distance queries on road networks: An experimental evaluation," 2012, *arXiv:1201.6564*.

[3] C. Sommer, "Shortest-path queries in static networks," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 1–31, Apr. 2014.

[4] J. Bao, C.-Y. Chow, M. F. Mokbel, and W.-S. Ku, "Efficient evaluation of k-range nearest neighbor queries in road networks," in *Proc. 11th Int. Conf. Mobile Data Manage.*, May 2010, pp. 115–124.

[5] H. Jung, M. Song, H. Youn, and U. Kim, "Evaluation of content-matched range monitoring queries over moving objects in mobile computing environments," *Sensors*, vol. 15, no. 9, pp. 24143–24177, Sep. 2015.

[6] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis, "Continuous nearest neighbor monitoring in road networks," Tech. Rep., 2006.

[7] H.-J. Cho, S. J. Kwon, and T.-S. Chung, "A safe exit algorithm for continuous nearest neighbor monitoring in road networks," *Mobile Inf. Syst.*, vol. 9, no. 1, pp. 37–53, 2013.

[8] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li, "Continuous reverse k nearest neighbors queries in Euclidean space and in spatial networks," *VLDB J.*, vol. 21, no. 1, pp. 69–95, Feb. 2012.

[9] M. Attique, H.-J. Cho, R. Jin, and T.-S. Chung, "Efficient processing of continuous reverse k nearest neighbor on moving objects in road networks," *ISPRS Int. J. Geo-Inf.*, vol. 5, no. 12, p. 247, Dec. 2016.

[10] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-k spatial keyword queries," in *Proc. Int. Symp. Spatial Temporal Databases* Minneapolis, MN, USA: Springer, Aug. 2011, pp. 205–222.

[11] J. B. Rocha-Junior and K. Nørvåg, "Top-k spatial keyword queries on road networks," in *Proc. 15th Int. Conf. Extending Database Technol.*, Mar. 2012, pp. 168–179.

[12] H.-J. Cho, S. J. Kwon, and T.-S. Chung, "ALPS: An efficient algorithm for top-k spatial preference search in road networks," *Knowl. Inf. Syst.*, vol. 42, no. 3, pp. 599–631, Mar. 2015.

[13] M. Attique, M. Afzal, F. Ali, I. Mehmood, M. F. Ijaz, and H.-J. Cho, "Geo-social top-k and skyline keyword queries on road networks," *Sensors*, vol. 20, no. 3, p. 798, Feb. 2020.

[14] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao, "All-nearest-neighbors queries in spatial databases," in *Proc. 16th Int. Conf. Sci. Stat. Database Manage.*, Jun. 2004, pp. 297–306.

[15] Y. Chen and J. M. Patel, "Efficient evaluation of all-nearest-neighbor queries," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 1056–1065.

[16] G. Chatzimilioudis, D. Zeinalipour-Yazti, W.-C. Lee, and M. D. Dikaiakos, "Continuous all k-nearest-neighbor querying in smartphone networks," in *Proc. IEEE 13th Int. Conf. Mobile Data Manage.*, Jul. 2012, pp. 79–88.

[17] Y. Xu, J. Qi, R. Borovica-Gajic, and L. Kulik, "Finding all nearest neighbors with a single graph traversal," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, Gold Coast, QLD, Australia: Springer, May 2018, pp. 221–238.

[18] A. Bhandari, A. Hasanov, M. Attique, H.-J. Cho, and T.-S. Chung, "Efficient processing of all nearest neighbor queries in dynamic road networks," *Mathematics*, vol. 9, no. 10, p. 1137, May 2021.

[19] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016, doi: 10.1145/2934664.

[20] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *Proc. VLDB Conf.* Amsterdam, The Netherlands: Elsevier, 2003, pp. 802–813.

[21] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 43–54.

[22] K. C. K. Lee, W.-C. Lee, B. Zheng, and Y. Tian, "ROAD: A new spatial object search framework for road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 3, pp. 547–560, Mar. 2012.

[23] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 237–246, Jun. 1993.

[24] N. Mamoulis and D. Papadias, "Multiway spatial joins," *ACM Trans. Database Syst.*, vol. 26, no. 4, pp. 424–475, Dec. 2001.

[25] C. Xia, H. Lu, B. C. Ooi, and J. Hu, "GORDER: An efficient method for KNN join processing," in *Proc. 13th Int. Conf. Very Large Data Bases*, vol. 30, 2004, pp. 756–767.

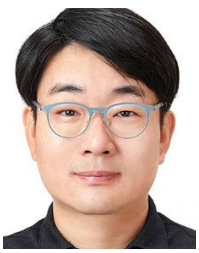
[26] B. Yao, F. Li, and P. Kumar, "K nearest neighbor queries and kNN-joins in large relational databases (almost) for free," in *Proc. IEEE 26th Int. Conf. Data Eng. (ICDE)*, Mar. 2010, pp. 4–15.
[27] K. L. Clarkson, "Fast algorithms for the all nearest neighbors problem," in *Proc. 24th Annu. Symp. Found. Comput. Sci. (SFCS)*, Nov. 1983, pp. 226–232.
[28] P. M. Vaidya, "An optimal algorithm for the all-nearest-neighbors problem," in *Proc. 27th Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1986, pp. 117–122.
[29] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W.-C. Lee, and E. Pitoura, "Distributed in-memory processing of all k nearest neighbor queries," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 4, pp. 925–938, Apr. 2016.
[30] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," in *Proc. 15th Int. Conf. Extending Database Technol.*, Mar. 2012, pp. 38–49.
[31] W. Lu, Y. Shen, S. Chen, and B. Chin Ooi, "Efficient processing of k nearest neighbor joins using MapReduce," 2012, *arXiv:1207.0141*.
[32] S. Luo, Y. Luo, S. Zhou, G. Cong, and J. Guan, "DISKs: A system for distributed spatial group keyword search on road networks," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1966–1969, Aug. 2012.
[33] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce framework for spatial data," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 1352–1363.
[34] T. Yokoyama, Y. Ishikawa, and Y. Suzuki, "Processing all k-nearest neighbor queries in Hadoop," in *Proc. Int. Conf. Web-Age Inf. Manage.* Cham, Switzerland: Springer, 2012, pp. 346–351.
[35] P. Moutafis, G. Mavrommatis, M. Vassilakopoulos, and S. Sioutas, "Efficient processing of all-k-nearest-neighbor queries in the MapReduce programming framework," *Data Knowl. Eng.*, vol. 121, pp. 42–70, May 2019.
[36] G. Karypis and V. Kumar, "METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," Tech. Rep., 1997.
[37] K. Schloegel, G. Karypis, and V. Kumar, "Parallel multilevel algorithms for multi-constraint graph partitioning," in *Proc. Eur. Conf. Parallel Process.* Munich, Germany: Springer, Aug. 2000, pp. 296–310.
[38] B. Smock, "Bsmock/k-shortest-paths: First official major release," Tech. Rep., Apr. 2017, doi: 10.5281/zenodo.439762.
[39] J. Y. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quart. Appl. Math.*, vol. 27, no. 4, pp. 526–530, 1970.
[40] (2005). *Real Datasets for Spatial Databases*. [Online]. Available: https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm

AAVASH BHANDARI received the B.Sc. degree (Hons.) in business computing and information systems from the University of Central Lancashire, in 2017. His research as the Ph.D. student with Ajou University, focuses on spatial database technologies including query processing in road networks. His research interests include location-based services and spatial query processing.

PRINCE HAMANDAWANA received the B.Sc. degree (Hons.) in computer science from the National University of Science and Technology (NUST), Bulawayo, Zimbabwe, in 2010, and the Ph.D. degree in artificial intelligence from Ajou University, Suwon, South Korea, in 2020. He is currently an Assistant Professor with the Department of Software, Ajou University. His research interests include distributed and parallel storage systems and machine learning performance optimizations.

MUHAMMAD ATTIQUE received the bachelor's degree in information and communication systems engineering from the National University of Science and Technology, Pakistan, in 2008, and the Ph.D. degree in computer science and engineering from Ajou University, South Korea, in 2017. He is currently an Assistant Professor with the Department of Software, Sejong University, South Korea. His research interests include spatial queries, big data analysis, social network analysis, information retrieval, and health care monitoring systems.

HYUNG-JU CHO received the B.S. and M.S. degrees in computer engineering from Seoul National University, in February 1997 and February 1999, respectively, and the Ph.D. degree in computer science from KAIST, in August 2005. He is currently a Professor with the Department of Software, Kyungbook National University, South Korea. His research interests include moving object databases and query processing in mobile peer-to-peer networks.



TAE-SUN CHUNG received the B.S. degree from KAIST, Daejeon, South Korea, in 1995, and the M.S. and Ph.D. degrees from Seoul National University, Seoul, South Korea, in 1997 and 2002, respectively. He is currently a Professor with the Department of Artificial Intelligence, Ajou University, Suwon, South Korea. His research interests include flash memory storage, XML databases, and database systems.