Distributed Spatio-Temporal k Nearest Neighbors Join

Ruiyuan Li[1,2], Rubin Wang[2], Junwen Liu[2], Zisheng Yu[4,2], Huajun He[3,2], Tianfu He[5,2], Sijie Ruan[4,2], Jie Bao[2], Chao Chen[1], Fuqiang Gu[1], Liang Hong[6], Yu Zheng[2]
[1] Chongqing University, Chongqing, China
[2] JD Intelligent Cities Research, Beijing, China
[3] Southwest Jiaotong University, Chengdu, China
[4] Xidian University, Xi'an, China
[5] Harbin Institute of Technology, Harbin, China
[6] Wuhan University, Wuhan, China
{liruiyuan,hong}@whu.edu.cn;{wangrubin3,liujunwen8,yuzisheng3,hehuajun3,baojie}@jd.com
{Tianfu.D.He,sijieruan,msyuzheng}@outlook.com;{cschaochen,gufq}@cqu.edu.cn

ABSTRACT

The rapid development of positioning technology produces an extremely large volume of spatio-temporal data with various geometry types such as point, line string, polygon, or a mixed combination of them. As one of the most basic but time-consuming operations, k nearest neighbors join (kNN join) has attracted much attention. However, most existing works for kNN join either ignore temporal information or consider point data only.

This paper proposes a novel and useful problem, i.e., ST-kNN join, which considers both *spatial closeness* and *temporal concurrency*. To support ST-kNN join over a huge amount of spatio-temporal data with any geometry types efficiently, we propose a novel distributed solution based on Apache Spark. Specifically, our method adopts a two-round join framework. In the first round join, we propose a new spatio-temporal partitioning method that achieves spatio-temporal locality and load balance at the same time. We also propose a lightweight index structure, i.e., Time Range Count Index (TRC-index), to enable efficient ST-kNN join. In the second round join, to reduce the data transmission among different machines, we remove duplicates based on spatio-temporal reference points before shuffling local results. Extensive experiments are conducted using three real big datasets, showing that our method is much more scalable and achieves 9X faster than baselines. A demonstration system is deployed and the source code is released.

CCS CONCEPTS

• **Computing methodologies → MapReduce algorithms**; • **Information systems → Geographic information systems**; **Join algorithms**.

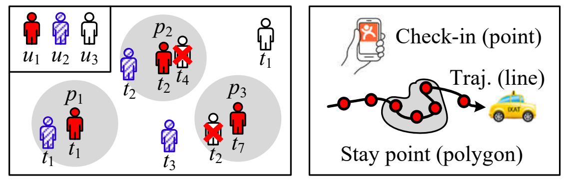
KEYWORDS

Distributed Computing, Spatio-Temporal kNN Join, kNN Join

ACM Reference Format:
Ruiyuan Li, Rubin Wang, et al.. 2021. Distributed Spatio-Temporal k Nearest Neighbors Join. In *29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21), November 2–5, 2021, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3474717.3484209

ACM SIGSPATIAL'21, November 02–05, 2021, Beijing, China

ACM ISBN 978-1-4503-8664-7/21/11...$15.00
https://doi.org/10.1145/3474717.3484209



(a) Epidemic Prevention (b) Various Geometry Types

Figure 1: Motivation of ST-kNN Join.

1 INTRODUCTION

With the rapid development of positioning technology, an extremely large number of spatio-temporal data is generated. Among spatio-temporal data analyses, k nearest neighbors join (kNN join) [29, 30, 34, 37, 42] is one of the most common operations, which is very useful in many applications. As shown in Fig. 1(a), in the case of epidemic prevention [20], given a set of check-ins of the infected patient u_1, kNN join ($k = 1$) finds the nearest user of each check-in point. Most existing solutions consider the *spatial closeness* only, so they find u_2 for p_1, u_3 for p_2, and u_3 for p_3, respectively. As a result, both u_2 and u_3 are of potentially vulnerable population and should be isolated. However, if we consider the *temporal concurrency* as well, u_3 is no longer the nearest to p_2, because they are generated at different times (i.e., t_2 and t_4, respectively). Similarly, u_3 is not the nearest to p_3. At the end, only u_2 is the potentially suspected user, which brings in a more precise epidemic prevention.

This paper proposes a new problem called ST-kNN join that considers both spatial closeness and temporal concurrency. ST-kNN can be applied to many other applications such as ride-sharing [28], companion detection [9] and travel recommendation [5]. However, it is challenging to perform ST-kNN join for three reasons: 1) **big data**. Spatio-temporal data is generated constantly at a very high frequency, leading to a prohibitively large volume of data; 2) **high dimensionality**. In addition to spatial information, we should also consider the temporal information, which is more intractable; and 3) **various geometry types**. Spatio-temporal data comes with various geometry types, e.g., points of check-ins, line strings of trajectories, and polygons of stay points [21], as shown in Fig. 1(b).

Over the last decade, there emerged many distributed frameworks, e.g., Apache Hadoop [11] and Apache Spark [41], which cope with big data efficiently. Many works [29, 30, 34, 37, 42] based on distributed frameworks for kNN join ignore the temporal information, therefore they cannot be applied to ST-kNN join directly. Besides, most of them [29, 30, 37, 42] are designed based on triangle inequality that is only fit for the distance between two points, so they do not support complex geometries such as line strings and polygons, and cannot support sophisticated urban applications.

As a result, this paper proposes a novel distributed solution based on Apache Spark, which supports ST-kNN join with various geometry types efficiently. Specifically, our solution follows a two-round join framework. In the first round join, we first partition the objects according to the spatio-temporal distribution, then find a distance bound for each object, such that its k nearest neighbors considering both spatial closeness and temporal concurrency must locate in a specific region. In the second round join, we first perform a local ST-kNN join to get local results, then merge them into a global one. Overall, the contributions of this paper are four-fold:

(1) This paper proposes a novel and useful ST-kNN join problem, and presents a distributed solution based on Apache Spark that supports ST-kNN join with any geometry type efficiently.

(2) We propose a new spatio-temporal partitioning method that achieves spatio-temporal locality and load balance at the same time. We devise a lightweight but effective index structure called Time Range Count Index (TRC-index), which returns the minimum number of satisfied objects in a partition. To reduce the data transmission among different machines, we remove duplicates based on spatio-temporal reference points before shuffling local results.

(3) Extensive experiments are carried out using three real datasets, which verifies the powerful efficiency and scalability of our method.

(4) An online demonstration system is deployed based on JUST [19, 24–26], and the source code of ST-kNN join is released [2].

Outline. We give some preliminaries in Section 2. In Section 3, we describe the overview of our proposed solution. The details of ST-kNN join are presented in Section 4. We present the evaluation results in Section 5, followed by the related works in Section 6. Finally, we conclude this paper with future works in Section 7.

2 PRELIMINARY

In this section, we give related definitions, and introduce some knowledge about Apache Spark. We list the symbols and their meanings in Appendix A for the purpose of reference.

2.1 Definition

Definition 1. (ST-object) An ST-object (spatio-temporal object) $r = (geom, tr)$ contains a spatial attribute *geom* and a time range *tr*, where *geom* can be any geometry (e.g., a point, a line string, a polygon, etc., or a mixed set of them), and $tr = [t_{min}, t_{max}]$ is a time range. The time span of r is defined as $|tr| = t_{max} - t_{min}$.

Note that $t_{min} = t_{max}$ is a special case of our definition. In the following, for the sake of simplicity, we call an ST-object **object**.

Definition 2. (MBR and EMBR) The MBR (Minimum Bounding Rectangle) of an object r is the smallest axis-aligned rectangle that contains all points of $r.geom$, which can be represented by two points $MBR(r) = \langle (lat_{min}, lng_{min}), (lat_{max}, lng_{max}) \rangle$. Its EMBR (Extended Minimum Bounding Rectangle) with regard to a distance threshold γ is defined as $EMBR(r, \gamma) = \langle (lat_{min} - \gamma, lng_{min} - \gamma), (lat_{max} + \gamma, lng_{max} + \gamma) \rangle$.

Definition 3. (Temporal Domain and Spatial Domain) Given a set of objects R, its temporal domain $TD(R)$ is the minimum time range that contains all time ranges of $r \in R$.

Similarly, the spatial domain of R is the MBR that contains all MBRs of $r \in R$, denoted as $SD(R)$.

Definition 4. (Expanded Time Range) Given a time range $tr = [t_{min}, t_{max}]$ and a time threshold δ, the expanded time range of tr is defined as $ETR(tr, \delta) = [t_{min} - \delta, t_{max} + \delta]$.

Definition 5. (ST-kNN) Given an object r, a set of objects S, an integer k, and a time threshold δ, the ST-kNN (Spatio-Temporal k Nearest Neighbors) of r from S is defined as $S' = \text{ST-}k\text{NN}(r, k, \delta, S)$, where S' contains at most k objects, i.e., $|S'| \leq k$, and $\forall s_i \in S'$, it satisfies the following two constraints at the same time:

(1) *Temporal Concurrency.* The temporal gap between r and s_i is no more than δ, i.e.,

$$ETR(r.tr, \delta) \cap s_i.tr \neq \emptyset \tag{1}$$

(2) *Spatial Closeness.* Suppose $S'' \subseteq S$ is the set of objects that meet the temporal concurrency constraint. Spatial closeness requires that $\forall s_i \in S', \forall s_j \in S'' \setminus S', d(r, s_i) < d(r, s_j)$.

Here, $|S'| < k$ iff $|S''| < k$. In this case, $|S'| = |S''|$. $d(r, s)$ measures the distance between r and s, which is defined as:

$$d(r, s) = \min_{p \in r.geom, q \in s.geom} d(p, q) \tag{2}$$

where $d(p, q)$ is the Euclidean distance between two spatial points.

Discussion. The temporal gap δ is defined because in many real applications such as ride-sharing [28], users would have tolerance for some time deviation (e.g., 15 minutes). In fact, the temporal concurrency with a gap is more general for various applications with different values of δ. Besides, we do not combine spatio-temporal dimensions into a single distance metric using a linear combiner with different weights [13]. Because 1) the temporal dimension has a very different scale from spatial dimension, so we should not put them together simply; and 2) for different applications the weights are different. It is intractable for end users to assign appropriate weights to spatio-temporal dimensions.

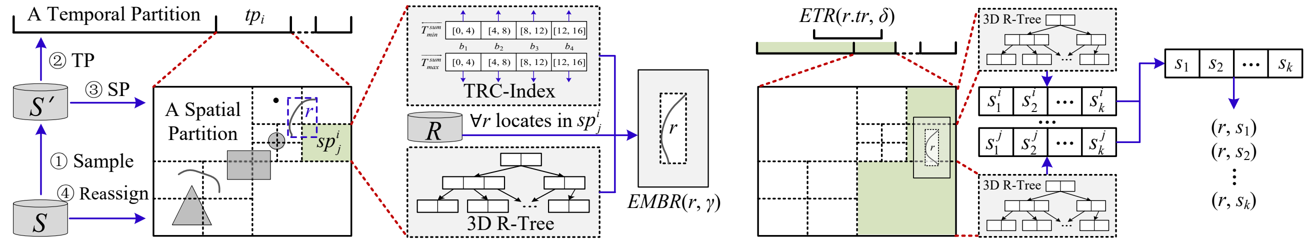
Definition 6. (ST-kNN Join) Given two sets of objects R and S, an integer number k, and a time threshold δ, ST-kNN join of R and S (denoted as $R \ltimes S$) combines each object $r \in R$ with its ST-kNNs from S. Formally,

$$R \ltimes S = \{(r, s) | \forall r \in R, \forall s \in \text{ST-}k\text{NN}(r, k, \delta, S)\} \tag{3}$$

According to the definition of ST-kNN join, those objects outside of the time period $GT = ETR(TD(R), \delta) \cap TD(S)$, i.e., $tr \cap GT = \emptyset$, would not contribute to the final results. So before actually performing ST-kNN join, we first filter out the objects in R and S outside of GT to avoid unnecessary computations. We call GT *global temporal domain*, and $GS = SD(S)$ *global spatial domain*. In the following, R and S represent the filtered set, respectively.

2.2 Apache Spark

Apache Spark [41] is an in-memory distributed framework for large-scale data processing with fault-tolerance. It provides an abstraction called resilient distributed dataset (`RDD`) consisting of several `partitions` across a cluster of machines. Each `RDD` is built using parallelized operations (e.g. `map`, `filter`, `reduce`). `RDDs` can be cached in memory or made persistent on disk to accelerate data reusing and support iteration. In Spark, we can broadcast variables



(a) Data Partition for S (b) First Round Local Join (c) Second Round Local Join (d) Merge Result

Figure 2: Overview of ST-kNN Join.

to all partitions in an RDD. Shuffle is an operation to reorganize data across partitions. Note that shuffle is very expensive as it moves data among partitions or even machines in a cluster, so we should try to avoid it when possible.

Although this paper presents ST-kNN join based on Apache Spark, we can transplant it easily to other distributed frameworks, such as Apache Hadoop [11] and Apache Flink [8].

3 OVERVIEW

Figure 2 presents the framework of our proposed solution for ST-kNN join, which consists of four main steps:

Data Partition for S**.** In this step, as shown in Fig. 2(a), we divide S into several spatio-temporal partitions (ST-partitions), where the numbers of objects in different partitions are almost the same to achieve a good load balance.

First Round Local Join. In this step, as described in Fig. 2(b), for each ST-partition, we build two local indexes, i.e., time range count index (TRC-index) and 3D R-tree index. Using these two indexes, for each object $r \in R$ that locates in this partition, we determine an area, in which the ST-kNNs of r must reside.

Second Round Local Join. As presented in Fig. 2(c), in this step, we examine *all* ST-partitions that overlap with the area of r calculated in the previous step. In each satisfied partition, we perform a kNN search, generating a set of local ST-kNNs of r.

Merge Result. As shown in Fig. 2(d), for each object r, we merge multiple local ST-kNN results into a global one, and produce the final result.

4 ST-kNN JOIN

In this section, we elaborate on the details of each step, and analyze the performance of our method finally.

4.1 Data Partition for S

In distributed environments for ST-kNN join, it is vital to design a good data partition strategy, which requires that: 1) *Spatio-Temporal Proximity.* Objects that are close spatially and temporally should be assigned to the same partition as much as possible, thus we are likely to find all ST-kNNs in one partition, reducing the network communication overhead among different partitions. 2) *Even Distribution.* The numbers of objects in different partitions are as equal as possible, thus we can achieve load balance.

Existing distributed frameworks for spatial data processing either focus on spatial partitioning merely [34, 37], or aim at spatial-temporal join [35, 36], which cannot be used for ST-kNN join directly. To that end, this paper devises a simple but effective spatio-temporal data partition strategy for ST-kNN join. We partition S with four steps: 1) *Sampling*, 2) *Temporal Partitioning*, 3) *Spatial Partitioning*, and 4) *Reassignment*, as shown in Fig. 2(a).

Sampling. In this step, we take a set of random samples S' from S with a sampling rate of η. Because S' is sampled randomly from S, it keeps the spatio-temporal data distribution of S. Then S' is collected to the driver program on the master node, where we would construct spatio-temporal partitions based on these samples. We take the same sampling rate $\eta = 1\%$ as Simba [37] did.

Temporal Partitioning. In this step, we divide the global temporal domain GT into at most α disjoint time ranges (called *temporal partitions*) $TP = \{tp_1, tp_2, ..., tp_m\}$, $m \leq \alpha$, such that $GT = \bigcup_{1 \leq i \leq m} tp_i$, and $\forall i \in [1, m], \forall j \in [1, m], i \neq j, tp_i \cap tp_j = \emptyset$. For any $s \in S'$, if its time range $s.tr$ overlaps with a temporal partition tp_i, i.e., $s.tr \cap tp_i \neq \emptyset$, s will be assigned to tp_i. As a consequence, an object will be copied many times if it intersects multiple temporal partitions. Here α is a system parameter, and we will show its effect on the ST-kNN join performance in Section 5.

The time span of a temporal partition has a significant impact on ST-kNN join. **First of all**, intuitively, to reduce the data replication of S, the time span of a temporal partition should not be too small (at least it should not be smaller than the time span of $s \in S'$). **Secondly**, however, during the join process, as we will see later, we will leverage temporal partitions to filter out irrelevant objects. As a result, to ensure a good filtering ability, the time span of a temporal partition should be small as much as possible. **Thirdly**, to avoid the replication of $r \in R$ during the following join process, the time span of a temporal partition is expected to be bigger than that of $ETR(r, \delta)$.

Based on the observations above, the time span of any temporal partition tp_i, $\forall i \in [1, m]$, should hold:

$$|tp_i| \geq max\{\overline{|s.tr|}, 2\delta + \overline{|r.tr|}\} \tag{4}$$

where $\overline{|s.tr|}$ and $\overline{|r.tr|}$ are the average time spans of $s \in S$ and $r \in R$, respectively. We adopt $2\delta + \overline{|r.tr|}$ because the expanded time span of $r \in R$ is expected to be $2\delta + \overline{|r.tr|}$.

Besides, to achieve load balance, the numbers of objects in different temporal partitions should be as equal as possible, which can be achieved by limiting the minimum number of samples in each temporal partition:

$$samples(tp_i) \geq |S'|/\alpha \tag{5}$$

where $|S'|$ is the object number in S'. $|S'|/\delta$ guarantees the number of temporal partitions is no more than α.

We propose a new temporal partitioning method based on Sweep Line Algorithm [32]. As shown in Algorithm 1, we first sort the objects in S' by the start time in an ascending order (Line 1), then initialize the following variables: tps stores the final temporal partitions, cur is a set of objects in the current temporal partition, $start$ records the start time of the current temporal partition, and sl is the sweep line (Line 2). In Lines 5-9, we scan S' from left to right. If the current temporal partition satisfies both Equ. (4) and Equ. (5), it forms a final temporal partition and is added to tps. Those objects in cur that do not contribute to the next temporal partition are filtered out. Finally, we process the last temporal partition and return the final results (Line 10).

Algorithm 1: TP(S', GT, k, δ, α, β, η)

1 Sort S' by the start time of objects in ascending order;
2 $tps = \emptyset$; $cur = \emptyset$; $start = GT.t_{min}$; $sl = GT.t_{min}$;
3 $minSpan = max\{\overline{|s.tr|}, 2\delta + \overline{|r.tr|}\}$;
4 $minNum = |S'|/\alpha$;
5 **for** $s \in S'$ **do**
6     $sl = s.tr.t_{min}$; $cur = cur \cup \{s\}$; $span = sl - start$;
7   **if** $span \geq minSpan$ *and* $|cur| \geq minNum$ **then**
8         $tps = tps \cup \{[start, sl]\}$; $start = sl$;
9   Filter out $s' \in cur$ that $s'.tr.t_{max} < start$;
10 **return** $tps \cup \{[start, GT.t_{max}]\}$;

Spatial Partitioning. In this step, for each temporal partition tp_i, we divide the global spatial domain GS into at most β spatial partitions $SP_i = \{sp_1^i, sp_2^i, ..., sp_n^i\}$, $n \leq \beta$, using Quad-tree [15] based on the samples S_i' assigned to tp_i. As these spatial partitions belong to a temporal partition, we call them **ST-partitions**. Like α, β is a system parameter as well, and we will test its impact on ST-kNN join performance in Section 5.

This paper adopts Quad-tree [15] to perform spatial partitioning for three reasons. **Firstly**, Quad-tree can mitigate the problem of unbalanced spatial distribution comparing to Grid partition [4, 27], as Quad-tree partitions the areas with denser objects into smaller regions. **Secondly**, comparing to R-tree [17] and its variants [6, 38], Quad-tree considers all parts of spatial domain, but R-tree and its variants ignore those unsampled areas. One optional method is to adjust the MBRs of nodes in R-tree when assigning the entire set S, but this is time-consuming and may produce a poor-performance R-tree, especially for non-point objects (e.g., line strings and polygons). **Thirdly**, for KD-tree [7], it is hard to determine a split line for non-point data, but Quad-tree splits the space more easily.

Quad-tree recursively splits the global spatial domain GS into four equal-sized sub-regions. If the MBR of an object $s \in S_i'$ intersects multiple sub-regions, it will be copied to all intersected sub-regions. Each sub-region is further split if it has more than ζ objects. All leaf sub-regions form a set SP_i of spatial partitions. Note that we check the MBR of an object instead of the object itself here, because it is much faster to check the spatial relation of two MBRs than that of two complex objects themselves.

However, it is not easy to decide a good ζ. It gets more complicated if we limit the maximum number β of spatial partitions. In our ST-kNN join problem, each $r \in R$ needs to find its ST-kNNs. It is efficient if we can find all its ST-kNNs in one partition. Besides, the numbers of objects in different spatial partitions should be as the same as possible for load balance. As a result, ζ is defined as:

$$\zeta = max\{|S_i'|/\beta, 4\eta \times k \times |tp_i| \div (2\delta + \overline{|r.tr|})\} \quad (6)$$

where $|S_i'|/\beta$ is the average samples number in a spatial partition. $4\eta \times k \times |tp_i| \div (2\delta + \overline{|r.tr|})$ ensures that after a split, at least one of its sub-regions is expected to have more than k satisfied objects.

As shown in Algorithm 2, we resort to a priority queue pq to split the Quad-tree nodes. Initially, the global spatial domain GS (i.e., the root of Quad-tree) is inserted into pq. Then we check all nodes in pq in a descending order of sample numbers. If the current node has less than ζ samples, the split process is terminated. Otherwise, we split the node into four sub-nodes, and add them into pq. This process is repeated until the number of spatial partitions is not less than β. Each node in pq represents a spatial partition.

Algorithm 2: SP(S_i', GS, k, β, η)

Initialize a priority queue pq, with keys as the sample numbers in Quad-tree nodes, sorted in a descending order;
$\zeta = max\{|S_i'|/\beta, 4\eta \times k \times |tp_i| \div (2\delta + \overline{|r.tr|})\}$; $pq.push(GS)$;
while $pq.length < \beta$ **do**
    $node = pq.pop()$;
    if $samples(node) < \zeta$ **then**
        $pq.push(node)$; **break**;
    Split *node* into four sub-nodes, and add them into pq;
return the nodes in pq as spatial partitions;

Reassignment. After previous two steps, we get at most $\alpha \times \beta$ ST-partitions. Each ST-partition is bounded to a time range and an MBR, with which we build a global index GI, where the time ranges are organized as a sorted array, and the MBRs are organized as a Quad-tree. The global index GI is broadcast to all partitions of S. For each $s \in S$, if its time range and geometry *both* intersect with that of an ST-partition, the identifier of the ST-partition will be bounded to s. After that, S is re-partitioned according to the bounded identifiers. The objects with the same identifier will be assigned to the same partition. Note that an object s may be assigned to multiple new partitions, as it may intersect several ST-partitions.

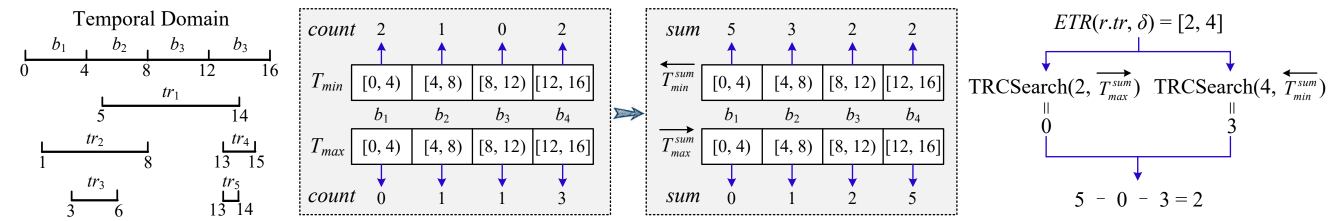
4.2 First Round Local Join

In this step, for each $r \in R$, we aim to find an area $EMBR(r, \gamma)$, such that its ST-kNNs in S must intersect with $EMBR(r, \gamma)$. Existing two-round methods such as LocationSpark [33, 34] mainly focus on point data. Besides, they do not consider the temporal information.

It is much more challenging for ST-kNN join because of two reasons. **First**, for a non-point object $r \in R$ with a time range, it may intersect with more than one ST-partitions at the same time. **Second**, it is hard to figure out whether a partition contains at least k objects that meet the temporal concurrency requirement.

For the first challenge, we check all intersected ST-partitions to find the nearest one. For the second challenge, we propose a new index structure called TRC-index (Time Range Count Index) in each ST-partition to get the minimum number of intersected time ranges of $ETR(r, \delta)$ efficiently. Overall, the first round local join contains three steps: 1) *TRC-index Construction*, 2) *Data Partition for R*, and 3) *Distance Bound Calculation.*

TRC-index Construction. There are two requirements for TRC-index. 1) Given a set of objects S_i in an ST-partition and a time range tr, it returns efficiently the minimum number of objects in S_i whose time ranges intersect with tr. 2) TRC-index should be as small as possible, because it will be broadcast to all partitions of R.



(a) Time Range Database (b) Count for Each Bin (c) TRC-Index (d) TRC Search Example

Figure 3: Illustration of TRC-Index.

To this end, we design a lightweight but effective TRC-index. The intuition of TRC-index is straightforward: if we know the upper bound number N of time ranges that would not intersect with the given time range tr, then we can obtain easily the lower bound number, i.e., $|S_i| - N$, of intersected time ranges. For any object $s \in S_i$, its time range does not intersect with tr iff $s.tr.t_{max} < tr.t_{min}$ or $s.tr.t_{min} > tr.t_{max}$. Therefore, to accelerate the computation of N, TRC-index stores the number of objects whose maximum time is less than $tr.t_{min}$ or minimum time is greater than $tr.t_{max}$.

Algorithm 3: TRCIndex(S_i, $binNum$)

1 Initialize two arrays T_{min} and T_{max} with length of $binNum$;
2 $binLen = \lceil |TD(S_i)|/binNum \rceil$;
3 **for** $s \in S_i$ **do**
4     $j_1 = \lfloor (s.tr.t_{min} - TD(S_i).t_{min})/binLen \rfloor$; $T_{min}[j_1] + +$;
5     $j_2 = \lfloor (s.tr.t_{max} - TD(S_i).t_{min})/binLen \rfloor$; $T_{max}[j_2] + +$;
6 **for** $j = 1; j < binNum; j + +$ **do**
7     $T_{min}[binNum - j - 1]$ += $T_{min}[binNum - j]$;
8     $T_{max}[j]$ += $T_{max}[j - 1]$;
9 **return** $\langle TD(S_i), |S_i|, binNum, T_{min}, T_{max} \rangle$ as TRC-index;

As the time dimension is continuous, we use discrete disjoint bins with equal length to represent the time information approximately. Algorithm 3 presents the pseudo-code of TRC-index construction. We use two arrays T_{min} and T_{max} to record the number of time ranges whose start time and end time locate in each bin, respectively (Line 1). The objects S_i in the ST-partition are scanned linearly. For each object $s \in S_i$, we first calculate its start and end time bin numbers, respectively, then increase their counts by 1 (Lines 3-5). After that, we accumulate the counts by scanning T_{min} and T_{max} for once (Lines 6-8). Note that we accumulate the counts of T_{min} from right to left, but T_{max} from left to right. By doing this, we can get quickly the number of objects whose start time is greater than $tr.t_{max}$ using T_{min}, and the number of objects whose end time is less than $tr.t_{min}$ using T_{max}. Finally, the TRC-index is returned as a quintuple $\langle TD(S_i), |S_i|, binNum, T_{min}, T_{max} \rangle$ (Line 9).

With the help of TRC-index, we can calculate quickly the lower bound number of objects whose time ranges intersect with $tr = [t_{min}, t_{max}]$. We first compute the bin numbers, i.e., b_{min} and b_{max}, of $tr.t_{min}$ and $tr.t_{max}$, respectively, using the similar method in Lines 3-5 of Algorithm 3. The number of objects whose end time is smaller than $tr.t_{min}$ is **at most** $T_{max}[b_{min}]$ (note that in the bin b_{min}, there exist some objects whose end time is not smaller than $tr.t_{min}$). Similarly, the number of objects whose start time is greater than $tr.t_{max}$ is **at most** $T_{min}[b_{max}]$. As a result, the number of objects whose time ranges intersect with tr is **at least** $|S_i| - T_{max}[b_{min}] - T_{min}[b_{max}]$.

The total bin number $binNum$ provides a trade-off between network overhead and result precision. A bigger $binNum$ means a higher lower bound, but requires more network data transmission. We will investigate its effect on ST-kNN join in Section 5.

For example, given a time range database shown in Fig. 3(a), we first count the number of objects in each bin in Fig. 3(b) (here $binNum$ is set as 4), then accumulate the counts in Fig. 3(c). With TRC-index, we find that there are at least 2 time ranges intersecting with "[2, 4]" (i.e., "[1, 8]" and "[3, 6]"), as shown in Fig. 3(d).

Data Partition for R. In the previous step, we build a TRC-index in each ST-partition. Recall that in Section 4.1, we built a global index GI. In this step, we broadcast GI and all TRC-indexes to the partitions of R. Because GI and TRC-indexes are small enough, the broadcast overhead can be ignored. For each $r \in R$, we find a set of temporal partitions $TP' = \{tp'_1, tp'_2, ..., tp'_u\}$ that intersects with $ETR(r.tr, \delta)$ using GI. In each $tp'_i \in TP'$, we find r's nearest spatial partition sp'^i that has at least k satisfied objects (i.e., whose time ranges intersect with $ETP(r.tr, \delta)$) in S using GI and TRC-index. At the end, we get u spatial partitions $\{sp'^1, sp'^2, ..., sp'^u\}$, among which we select the nearest one and assign its identifier to r. Finally, R is re-partitioned according to the bounded identifiers, where the objects $r \in R$ with the same identifier are shuffled to the same ST-partition. Note that for each $r \in R$, it is assigned to at most ONE ST-partition in this step. If $u = 0$, i.e., r cannot find any satisfied ST-partition with TRC-index, we do not re-partition it and let it skip the first round local join directly.

It is efficient to find r's nearest spatial partition that has at least k satisfied objects in S with the help of GI and TRC-index. Note that the spatial partitions $SP'_i = \{sp'^i_1, sp'^i_2, ..., sp'^i_n\}$ in a temporal partition $tp'_i \in TP'$ are organized as a Quad-tree in GI, thus we can easily check each spatial partition in SP'_i from near to far iteratively. For each sp'^i_j, we get the minimum number of objects whose time ranges intersect with $ETR(r.tr, \delta)$ using TRC-index. If the number is not less than k, the check process is terminated, and sp'^i_j is returned.

Distance Bound Calculation. Assigning r to an ST-partition which has at least k satisfied objects in S guarantees that, we can calculate a distance bound γ in this ST-partition, such that the distance between r and any ST-kNN is less than γ. Suppose R_i and S_i are the objects assigned to the ST-partition sp_i in R and S, respectively. In each ST-partition sp_i, we first build a local 3D R-tree index [43] over S_i, where the temporal information is regarded as the 3rd dimension. For each $r \in R_i$, we perform a local ST-kNN search in this partition using the built local 3D R-tree index, generating a local result $\{s^i_1, s^i_2, ..., s^i_k\}$ ordered by their distances to r. Thus, $\gamma = d(r, s^i_k)$. For those objects that cannot find a satisfied ST-partition in the previous step, we set $\gamma = \infty$. Note that the local join results and 3D R-tree index of S_i are cached to avoid redundant computations in the second round local join.

4.3 Second Round Local Join

In this step, for each $r \in R$, we check all possible ST-partitions that may produce its ST-kNNs, and generate local results.

Recall that after performing the first round local join, we get a distance bound γ for each $r \in R$. All ST-partitions that both temporally intersect with $ETR(r.tr, \delta)$ and spatially intersect with $EMBR(r, \gamma)$ are candidates. These candidates can be figured out efficiently using the global index GI. For each candidate ST-partition of r (except for the one we assigned to r in the first round local join, which must be a candidate ST-partition of r but we can omit it here to avoid repeated computations), we bound its identifier to r. After that, we re-partition R according to the bounded identifiers, where the objects in R with the same identifier are shuffled to the same ST-partition. Note that an object $r \in R$ will be copied several times because there may be multiple candidate ST-partitions of r. Finally, in each new ST-partition sp_i, we perform an ST-kNN search for every $r \in R_i$ by leveraging the local 3D R-tree index over S_i built in the first round local join. Different from the first round local join, the search process can be optimized further using the distance bound γ, i.e., if the distance between r and a 3D R-tree node is greater than γ, the ST-kNN search can be terminated immediately.

This step shuffles small *parts* of R, because we observe that the $EMBR(r, \gamma)$ of most objects $r \in R$ intersect with only one ST-partition. These objects can find their ST-kNNs in the first round join, thus do not participate in the second round join.

4.4 Merge Result

After two-round local joins, we obtain an individual local kNN result of r in its every ST-partition (note that we also consider the local results produced in the first round local join here). A straightforward method performs four steps. 1) shuffle local results, where the results of the same r are re-partitioned to the same new partition; 2) combine them into a global result of r using multiway merge algorithm [10]; 3) remove duplicates, because an object could be assigned to multiple ST-partitions, so there may be duplicated combinations of (r, s) in different local results; 4) take the first k combinations as the final result of r.

To reduce the network transmission overhead, this paper removes duplicates before re-partitioning. For example, as shown in Fig. 4, suppose the combination (r, s) emerges in the local results of ST-partition 0, 1 and 2. The start time of $ETR(r.tr, \delta) \cap s.tr$ is called *temporal reference point* (TRP), and the lower-left corner of $EMBR(r, \gamma) \cap MBR(r)$ is called *spatial reference point* (SRP). We only retain (r, s) in the ST-partition 0 that contains the TRP and SRP, and discard them from the local results in other two ST-partitions.

Lemma 1. *The duplicate removal method proposed above is correct.*

Proof. We prove it from two aspects: *integrity* and *uniqueness*.

Integrity: If s is among the ST-kNNs of r, (r, s) will be generated in the ST-partition in which TRP and SRP locate. According to the definitions of TRP and SRP, we have TRP $\in s.tr$, TRP $\in ETR(r.tr, \delta)$, SRP $\in MBR(s)$, and SRP $\in EMBR(r, \gamma)$. s will be re-partitioned to all ST-partitions that temporally intersect with $s.tr$ and spatially intersect with $MBR(s)$, and r will be re-partitioned to all ST-partitions

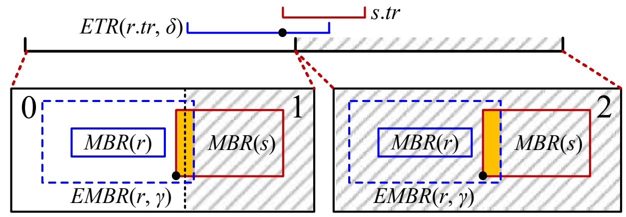


Figure 4: Remove Duplicates.

that temporally intersect with $ETR(r.tr, \delta)$ and spatially intersect with $EMBR(r, \gamma)$. As a result, r and s will emerge simultaneously in the ST-partition sp^i_j that the TRP and SRP locate in, thus (r, s) must be produced in sp^i_j if s is among the ST-kNNs of r.

Uniqueness: only one ST-partition contains TRP and SRP simultaneously. According to the partitioning strategy, we have $tp_i \cap tp_j = \emptyset$ if $i \neq j$, and $sp^i_m \cap sp^i_n = \emptyset$ if $m \neq n$. Suppose there exist two different ST-partitions sp^i_m and sp^j_n contain TRP and SRP simultaneously, i.e. TRP $\in tp_i \cap tp_j$ and SRP $\in sp^i_m \cap sp^j_n$. If $i \neq j$, TRP $\in tp_i \cap tp_j$, which contradicts with the temporal partitioning strategy. If $i = j$ and $m \neq n$, SRP $\in sp^i_m \cap sp^i_n$, which is contradictory to the spatial partitioning strategy. □

4.5 Performance Analysis

One of the most expensive overhead in a distributed environment is the data transmission among different machines, which is triggered when we broadcast data and shuffle RDD. Recall that during ST-kNN join, we broadcast the global index GI and TRC-indexes for once, but we can ignore the broadcast overhead because both GI and TRC-indexes are relatively very small. Figure 5 shows the data shuffle in different steps, where S is shuffled for only once, R is shuffled for twice, and the local join results are shuffled for once. Because most $r \in R$ can find its ST-kNNs in the first round local join (see Section 5), only few objects in R take part in the second round shuffle. We also remove duplicates before shuffling local join results, which reduces data transmission overhead further.

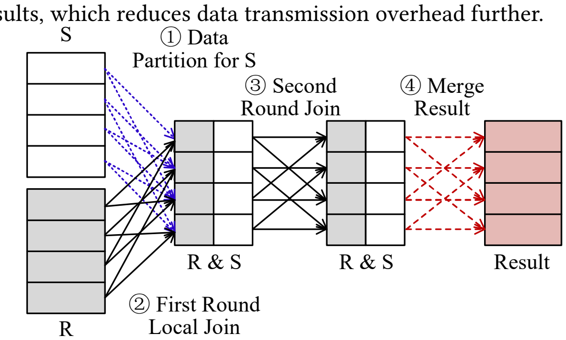


Figure 5: Shuffles of RDD.

As for computation complexity, we build a global index GI (consisting of a sorted array for temporal partitions and multiple Quad-trees for spatial partitions) based on the sample data S', which takes $O(|S'| \times log|S'| + \beta \times |S'| + \alpha \times \beta \times log\beta)$. We build two local indexes (i.e., TRC-index and Quad-tree over S_i) in each ST-partition, which takes $O(|S_i| + |S_i| \times log|S_i|)$. Using global and local indexes, it takes $O((|S| + 2 \times |R|) \times log\alpha \times log\beta)$ to find the ST-partitions of R and S. In each ST-partition, we take $O((|R_i| + |S_i|) \times log|S_i|)$ to perform 2 rounds local join. Finally, it takes $O(|R| \times k \times logv)$ to merge results, where v is the average number of ST-partitions an r locates in. We give more details of complexity analysis in Appendix B.

5 EVALUATION

5.1 Datasets and Settings

Datasets. We use three real big datasets to verify the performance of ST-kNN join method: 1) **NYTrip** [12]. We extract six months of taxi trips in New York City. Each trip has the location and time information of pick-up and drop-off, respectively. The locations of a pick-up or a drop-off can be modeled as *point* data (abbr. **pt**); 2) **DidiTraj** [1], which contains two months of taxi trajectories in Xi'an, China. A trajectory can be modeled as a *line string* (abbr. **ls**); and 3) **DidiSP**, which is a set of stay points extracted from DidiTraj using the method proposed in [23]. A stay point is deemed as a *polygon* (abbr. **py**). Table 1 shows the statistics.

Settings. Table 2 shows the geometry combinations for parameter tuning, which aims at finding out the effects of the introduced parameters. Table 3 summarizes the experimental parameters, where the default values are in bold. All experiments are conducted on a cluster of 5 nodes, with each node equipped with CentOS 7.4, 24-core CPU and 128GB RAM. We deploy Hadoop 2.7.6 and Spark 2.3.3 in our cluster. During the experiment, we assign 5 cores and 5GB RAM to the driver program, and set up 30 executors in the Spark cluster. Each executor is assigned 5 cores and 16GB RAM.

Metrics. We focus on three metrics: 1) **Execution Time (ET)**, which is the time cost for an ST-kNN join; 2) **Copy Amplification (CA)**, which is defined as the ratio of total copy times of objects in R (or S) to $|R|$ (or $|S|$). For example, if an object r intersects with n ST-partitions, it will be copied n times, thus its copy amplification is n; and 3) **Hit Rate (HR)**, which is defined as $|R'|/|R|$. $R' \subseteq R$ is a set of objects that can find their final ST-kNNs in the first round local join, so they do not participate in the second round join.

Baselines. As this paper is the first to address the ST-kNN join problem, we rewrite the source code of two related frameworks, i.e., Simba[37] and LocationSpark [33, 34], to make them support ST-kNN join. We do not compare the works [35, 36] because their source codes are not released. We also compare two variants of our proposed method (our method is termed as **ST-kNNJ**).

- **Simba** [37]. Simba provides efficient kNN join. We first find the $2\times k$ nearest neighbors for each $r \in R$, then filter out the objects $s \in S$ that do not meet the temporal concurrency requirement. It may not produce enough k results, because of the temporal filtering.
- **LocationSpark (LS)** [33, 34]. We rewrite its source code to make it support ST-kNN join, as we did for Simba. Note that both Simba and LocationSpark do not support complex data in the code.
- **ST-kNNJ$_R$**, which adopts R-tree for spatial partitioning. This method makes spatial partitions based on the centroid points of $s \in S'$. Each $s \in S$ is assigned to the nearest spatial partition. Each spatial partition is an MBR containing all $s \in S$ assigned to it. Each $r \in R$ is assigned to all spatial partitions that intersect with it.
- **ST-kNNJ$_{nr}$**, which adopts Quad-tree for spatial partitioning just as ST-kNNJ, but does not remove duplicates based on reference points before shuffling local join results.

5.2 Parameters Tuning

Different Values of α. Figure 6 presents the performance of ST-kNNJ with different values of α. As shown in Fig. 6(a), with an increasing α, the execution time of all dataset combinations first decreases, then increases. There are two reasons for an increasing execution time with a smaller α when $\alpha < 100$. Firstly, for a smaller α, the number of objects from S in an ST-partition tends to be larger, thus the 3D R-tree in the ST-partition gets bigger, and it needs more time to perform a local ST-kNN search with the 3D R-tree. Secondly, a smaller α leads to bigger temporal partitions, which weakens the temporal filtering ability.

Table 1: Statistics of Datasets

Attributes	NYTrip	DidiTraj	DidiSP
Raw Size	11.6GB	8.3GB	1.9GB
# Records	87,110,491	39,224,513	9,108,396
# Coords	174,220,982	348,191,629	73,708,681
Temporal Domain	2013/01/01 - 2013/06/30	2018/10/01 - 2018/11/30	2018/10/01 - 2018/11/30
Spatial Domain	(-74.07 : -73.75), (40.61 : 40.87)	(108.92 : 109.01), (34.20 : 34.28)	(108.92 : 109.01), (34.20 : 34.28)

Table 2: Datasets for Parameters Tuning

Datasets	Geometry	R	S
NYTrip	pt ⋉ pt	10% pick-up points	10% drop-off points
DidiTraj	ls ⋉ ls	10% samples	10% samples
DidiSP	py ⋉ py	50% samples	50% samples
Mixture	py ⋉ ls	50% DidiSP	10% DidiTraj

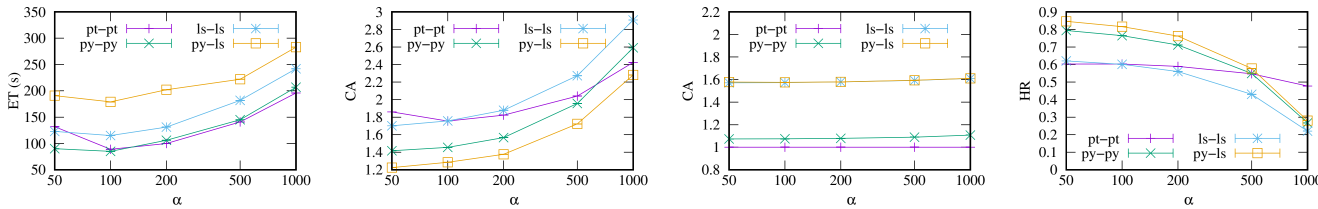
Table 3: Parameters

Parameters	Settings
Max # Temporal Partitions α	50, **100**, 200, 500, 1000
Max # Spatial Partitions β	5, 10, **20**, 50, 100
binNum in a TRC-index	10, 50, **200**, 500, 1000
Query Parameter δ (minutes)	10, 20, **30**, 40, 50
Query Parameter k	1, 5, **10**, 15, 20
Data Size (default values see Table 2)	10%, 20%, 30%, 40%, 50%

However, when $\alpha > 100$, the execution time gets more with a bigger α. The reasons could be 1) the copy rates of R and S gets larger with an increasing α, as shown in Fig. 6(b) and Fig. 6(c); 2) a bigger α results in a lower hit rate, as shown in Fig. 6(d). That is, more objects $r \in R$ cannot find their ST-kNNs in the first local join, thus they need to participate in the second join.

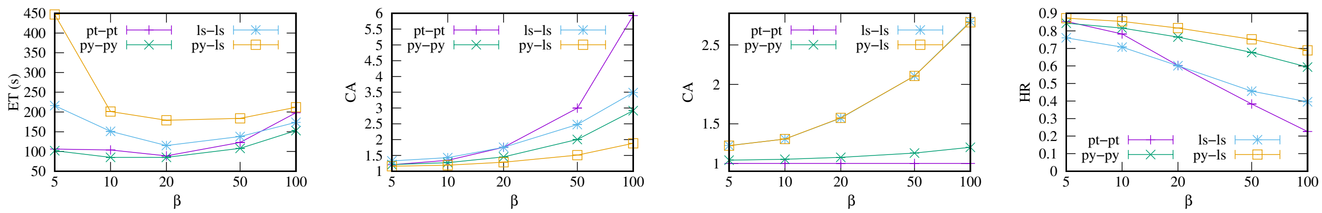
It is also interesting to see that in Fig. 6(a), the execution time of `ls-ls` and `py-ls` is more than that of `pt-pt`, even though the object number of `pt-pt` is much more than that of `ls-ls` and `py-ls` (8.7m ⋉ 8.7m vs 3.9m ⋉ 3.9m vs 4.6m ⋉ 3.9m). This is because 1) it is more time-consuming to calculate the distance between two complex objects; 2) the copy amplification of S for `ls-ls` and `py-ls` is much more than that of `pt-pt`, as shown in Fig. 6(c). Another interesting observation is that the copy amplification of R for `pt-pt` gets larger comparing $\alpha = 50$ to $\alpha = 100$ in Fig. 6(b), resulting in a fierce increasing of execution time in Fig. 6(a) and a slight drop off of hit rate in Fig. 6(d). It is because the global temporal domain of NYTrip is six months, which is much longer than that of other datasets. Given a specified *binNum* = 200, a longer temporal domain gives a coarser lower bound for TRC-index, which causes more $r \in R$ cannot find their ST-kNN in the first round local join.

Different Values of β. Figure 7 demonstrates the performance of ST-kNNJ with different values of β. As shown in Fig. 7(a), when β gets larger from 5 to 100, the execution time first drops, then increases slightly. When β = 20, the performance achieves the best. It is observed that with an increasing β, the copy amplifications of both R and S get larger. This is because with a bigger β, the area of



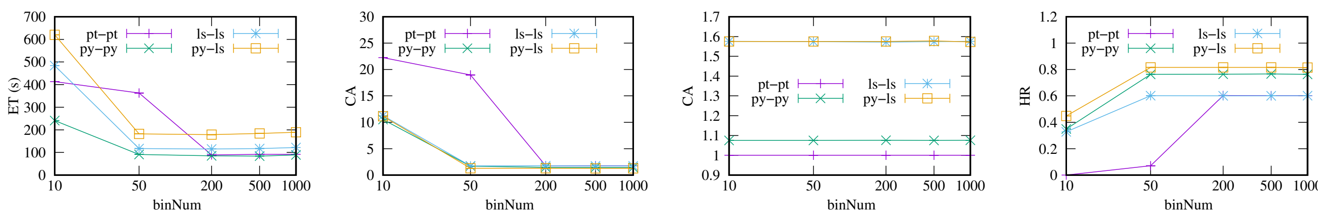
(a) Execution Time. (b) Copy Amplification of R. (c) Copy Amplification of S. (d) Hit Rate.

Figure 6: Performance w.r.t. α



(a) Execution Time. (b) Copy Amplification of R. (c) Copy Amplification of S. (d) Hit Rate.

Figure 7: Performance w.r.t. β



(a) Execution Time. (b) Copy Amplification of R. (c) Copy Amplification of S. (d) Hit Rate.

Figure 8: Performance w.r.t. $binNum$

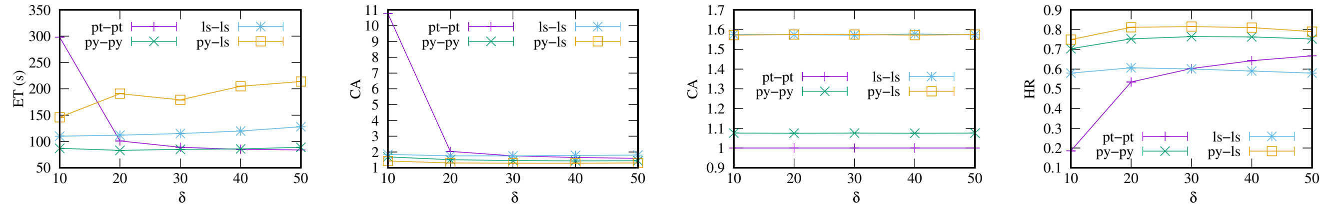
an ST-partition gets smaller, causing objects $s \in S$ more easily to intersect with more ST-partitions, thus the copy amplification of S get larger, especially for the polygon data and line string data, as shown in Fig. 7(c). Smaller ST-partitions also result in less objects in S assigned to the same ST-partition. This further causes objects $r \in R$ harder to find their ST-kNNs in the first round join, leading to a lower hit rate (shown in Fig. 7(d)) and a larger copy amplification of R (shown in Fig. 7(b)).

Different Values of *binNum*. As depicted in Fig. 8(a), with the increase of $binNum$, the execution time first drops significantly, then keeps smooth with a slight increase. This is because with a bigger $binNum$, TRC-index can provide a more precise lower bound, thus helps the objects $r \in R$ more easily to find the ST-partitions that contain their ST-kNNs, reducing the copy amplification of R, as shown in Fig. 8(b). A more precise lower bound improves the hit rate as well, as shown in Fig. 8(d). We can observe that the copy amplification of S has nothing to do with $binNum$, as shown in Fig. 8(c), because we partition S before building TRC-indexes. However, when $binNum$ is big enough, the execution time tends to be stable, as the lower bound of TRC-indexes is precise enough. Increasing $binNum$ only brings in more data transmission among different machines. It is interesting to see that the inflection point of `pt-pt` is larger than that of others (see Fig. 8(a) and Fig. 8(b)). There could be two reasons. Firstly, for point data, it is more easy to use the bins to calculate its real number that satisfies the temporal concurrency requirement, because the point data in our experiments has a time span of 0. Secondly, the NYTrip dataset has a much bigger global temporal domain than others, which needs more bins to capture its temporal distribution.

Different Values of δ. Figure 9 shows the impact of δ on ST-kNNJ performance. As shown in Fig. 9(a), with an increasing δ, the execution time of complex object combinations gets larger smoothly, as their copy amplification of R gets smaller sightly (shown in Fig. 9(b)), and their hit rate gets higher slightly (shown in Fig. 9(d)). However, for `pt-pt` combination, the execution time first drops significantly, then keeps stable. This is because for NYTrip dataset, the time span of objects is 0. If δ is set too small (e.g., $\delta = 10$), its hit rate is very low (shown in Fig. 9(d)), causing a huge copy amplification of R (see Fig. 9(b)). Again, we can see from Fig. 9(c) that the copy amplification of S has little to do with δ.

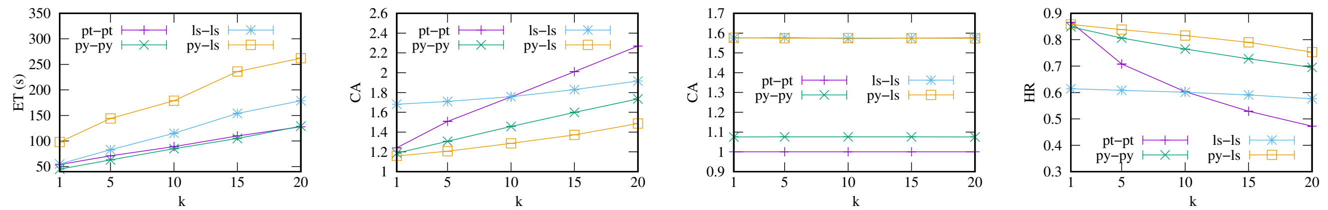
Different Values of k. It is observed from Fig. 10(a) that with a bigger k, the execution time for all combinations get larger linearly, because their hit rate decreases linearly (shown in Fig. 10(d)), making their copy amplification of R increase linearly (shown in Fig. 10(b)). Figure 10 demonstrates that the copy amplification of R is not affected by k.

Execution Time of Different Steps. Figure 11(a) shows the execution time for different steps. It is observed that the first round local join for almost all combinations is the most expensive, because we need to build local indexes in this step. Besides, most objects $r \in R$ can find their ST-kNNs in the first round local join, which reduces the computation of the second round local join. It is interesting to



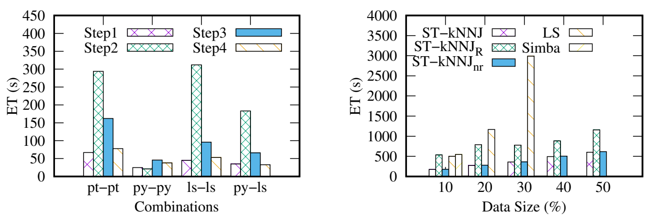
(a) Execution Time. (b) Copy Amplification of R. (c) Copy Amplification of S. (d) Hit Rate.

Figure 9: Performance w.r.t. δ



(a) Execution Time. (b) Copy Amplification of R. (c) Copy Amplification of S. (d) Hit Rate.

Figure 10: Performance w.r.t. k



(a) Execution Time. (b) Execution Time.

Figure 11: Performance w.r.t. Steps and Data Size (for the left picture, we take 50% samples for both R and S; for the right picture, we take `pt-pt` because both Simba and LocationSpark do not support complex geometries)

see that the first round local join for `py-py` is much less expensive than that for other combinations. It could be the spatial distribution of DidiSP is very sparse, and the data size of DidiSP is much smaller than that of other datasets.

5.3 Comparing with Baselines

Figure 11(b) compares the performance of different methods. We only focus on `pt-pt` because both Simba and LocationSpark do not support complex geometries. It is not surprising that with a bigger data size, all methods need more execution time. However, Simba fails when the data size is greater than 10%, because it needs to copy S too much, resulting in memory overflow and redundant computation. LocationSpark takes over 9X more time than ST-kNNJ, because it is not designed for ST-kNN join. Besides, we check its source code, and find that its proposed optimizer does not take effect for ST-kNN join. ST-kNNJ is much faster than ST-kNNJ$_R$, the reasons could be: 1) it is more efficient to build a Quad-tree than an R-tree; 2) R-tree ignores the unsampled areas in the spatial partitioning step, which leads to a poor performance; 3) the ST-partitions acquired by R-tree may intersect with each other, thus we cannot remove duplicated results using spatio-temporal reference points. ST-kNNJ is slightly faster than ST-kNN$_{nr}$, as we can remove duplicates before shuffling, which reduces the data transmission among different machines. However, the improvement is not so significant, because the local join result transmission overhead is relatively much smaller than the overall computation overhead.

6 RELATED WORKS

To the best of our knowledge, none of the existing works are designed for ST-kNN join. We review the related works from three aspects: 1) Spatial Join, 2) kNN Join, and 3) Spatio-Temporal Join.

Spatial Join. Spatial join combines two sets of spatial objects with a given spatial relation, such as containing, overlapping and distance. It has been well studied for a few decades, which can be divided into two categories: *standalone method* and *distributed method*. Most standalone methods [16, 31] adopt a two-phase framework, where in the first phase, they generate candidate pairs according to the MBRs of spatial objects, and in the second phase, they check the spatial relationship of each pair. The work [22] provides a comprehensive summary of the relevant technologies. To support massive spatial objects, many distributed frameworks are proposed for spatial join, such as Hadoop-GIS [3], SpatialHadoop [14], LocationSpark [33, 34], SpatialSpark [39], GeoSpark [40], Stark [18] and Simba [37]. Most of these distributed frameworks first partition the two sets, where the candidate pairs are assigned to the same partition. In each partition, they build a local spatial index [15, 17] and perform a spatial join using the standalone method. Finally, they merge local spatial join results into a global one.

kNN Join. Comparing to spatial join, kNN join is much more intractable, as it is hard to determine whether an object is one of kNNs of the other. There are two types of methods for distributed kNN join. The first one is one-round join method [29, 30, 37, 42]. They first partition R, then copy S to the target partitions based on the pivots of voronoi diagram [30], the partition center points [37], or space filling curves [29, 42], thus the kNNs of $r \in R$ must locate in the same partition with r. This type of method may cause too many copies of S, which hinders the efficiency. The other one is two-round join method [33, 34], which partitions S first, then copies R to the target partitions for twice. As most $r \in R$ can find their kNNs in the first round, it is much more efficient than the former.

Spatio-Temporal Join. The work [35, 36] considers both spatial and temporal information for spatial join, called spatio-temporal join. It employs two primary methods, i.e., broadcast join for the case when at least one of datasets can fit entirely into memory of a Spark executor, and bin join for the case when both datasets are too large to fit into memory. For bin join, it first spatially partitions the dataset using quadtree-based grid, then temporally partitions the dataset with a temporal interval. Stark [18] adds spatio-temporal support to Spark. It includes spatial partitioners, different indexing modes, as well as filter, join, and clustering operators. But Stark does not discuss how to support spatio-temporal join in the paper.

7 CONCLUSION

This paper proposes a novel and useful ST-kNN join problem, which finds the k nearest neighbors considering both spatial closeness and temporal concurrency. To efficiently perform ST-kNN join over big spatio-temporal data with any geometry types, we propose a novel distributed solution based on Apache Spark, which follows a two-round join framework. The extensive experimental results based on three big real datasets show that our method is much more scalable and achieves 9X faster than baselines. A demonstration system and the source code are available at [2].

There are two main directions to polish this work. First, data partitioning and index construction would be performed for each new ST-kNN join request currently. We can cache some intermediate results to avoid rebuilding all partitions and indexes from scratch and further improve the efficiency. Second, there still some system parameters, i.e., α, β and $binNum$, that may be affected by the sizes, geometry types, or spatio-temporal distributions of datasets. It is not easy to fine-tune them for every join manually. As a result, we will design cost models to execute ST-kNN join more intelligently.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (2019YFB2101801) and the National Natural Science Foundation of China (61976168, 72074172, 61872050, 62172066, 42174050). Data sources: Illinois Data Bank and Didi Chuxing GAIA Initiative.

REFERENCES

[1] 2021. Didi Chuxing GAIA Initiative. https://gaia.didichuxing.com

[2] 2021. ST kNN Join. http://stknnjoin.urban-computing.com/.

[3] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. In *PVLDB*, Vol. 6. NIH Public Access.

[4] Jie Bao, Ruiyuan Li, Xiuwen Yi, and Yu Zheng. 2016. Managing massive trajectories on the cloud. In *ACM SIGSPATIAL*. 1–10.

[5] Ramesh Baral, SS Iyengar, Tao Li, and XiaoLong Zhu. 2018. HiCaPS: Hierarchical contextual poi sequence recommender. In *ACM SIGSPATIAL*. 436–439.

[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD*. 322–331.

[7] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *IEEE TCDE* 36, 4 (2015).

[9] Lu Chen, Yunjun Gao, Ziquan Fang, Xiaoye Miao, Christian S Jensen, and Chenjuan Guo. 2019. Real-time distributed co-movement pattern detection on streaming trajectories. *PVLDB* 12, 10 (2019), 1208–1220.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.

[11] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. (2004).

[12] Brian Donovan and Dan Work. 2016. New York City Taxi Trip Data (2010-2013). https://doi.org/10.13012/J8PN93H8

[13] Jean Dubé and Diègo Legros. 2013. A spatio-temporal measure of spatial dependence: An example using real estate data. *Papers in Regional Science* 92, 1 (2013), 19–30.

[14] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*. IEEE, 1352–1363.

[15] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[16] Oliver Gunther. 1993. Efficient computation of spatial joins. In *ICDE*. IEEE, 50–59.

[17] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*. 47–57.

[18] Stefan Hagedorn, Philipp Gotze, and Kai-Uwe Sattler. 2017. The STARK framework for spatio-temporal data analytics on spark. *BTW* (2017).

[19] Huajun He, Ruiyuan Li, Jie Bao, Tianrui Li, and Yu Zheng. 2021. JUST-Traj: A Distributed and Holistic Trajectory Data Management System. In *ACM SIGSPATIAL*.

[20] Huajun He, Ruiyuan Li, Rubin Wang, Jie Bao, Yu Zheng, and Tianrui Li. 2020. Efficient suspected infected crowds detection based on spatio-temporal trajectories. *arXiv preprint arXiv:2004.06653* (2020).

[21] Yue Hu, Sijie Ruan, Yuting Ni, Huajun He, Jie Bao, Ruiyuan Li, and Yu Zheng. 2021. SALON: A Universal Stay Point-Based Location Analysis Platform. In *ACM SIGSPATIAL*.

[22] Edwin H Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM TODS* 32, 1 (2007), 7–es.

[23] Quannan Li, Yu Zheng, Xing Xie, and et al. 2008. Mining user similarity based on location history. In *ACM SIGSPATIAL*. 1–10.

[24] Ruiyuan Li, Huajun He, Rubin Wang, Yuchuan Huang, Junwen Liu, Sijie Ruan, Tianfu He, Jie Bao, and Yu Zheng. 2020. Just: Jd urban spatio-temporal data engine. In *ICDE*. IEEE, 1558–1569.

[25] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, Liang Hong, and Yu Zheng. 2021. TrajMesa: A Distributed NoSQL-Based Trajectory Data Management System. *TKDE* (2021).

[26] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. 2020. Trajmesa: A distributed nosql storage engine for big trajectory data. In *ICDE*. IEEE, 2002–2005.

[27] Ruiyuan Li, Sijie Ruan, Jie Bao, and Yu Zheng. 2017. A cloud-based trajectory data management system. In *ACM SIGSPATIAL*. 1–4.

[28] Chang Liu, Jiahui Sun, Haiming Jin, Meng Ai, Qun Li, Cheng Zhang, Kehua Sheng, Guobin Wu, Xiaohu Qie, and Xinbing Wang. 2020. Spatio-Temporal Hierarchical Adaptive Dispatching for Ridesharing Systems. In *ACM SIGSPATIAL*. 227–238.

[29] Y Liu, N Jing, L Chen, and W Xiong. 2013. Algorithm for processing k-nearest join based on r-tree in mapreduce. *Journal of Software* 24, 8 (2013), 1836–1851.

[30] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. 2012. Efficient processing of k nearest neighbor joins using mapreduce. *arXiv preprint arXiv:1207.0141* (2012).

[31] Jack A Orenstein. 1989. Strategies for optimizing the use of redundancy in spatial databases. In *Symposium on Large Spatial Databases*. Springer, 115–134.

[32] Franco P Preparata and Michael I Shamos. 2012. *Computational geometry: an introduction.* Springer Science & Business Media.

[33] Mingjie Tang, Yongyang Yu, Ahmed R Mahmood, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. 2020. Locationspark: in-memory distributed spatial query processing and optimization. *Frontiers in Big Data* 3 (2020), 30.

[34] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. 2016. Locationspark: A distributed in-memory data management system for big spatial data. *PVLDB* 9, 13 (2016), 1565–1568.

[35] Randall T Whitman, Bryan G Marsh, Michael B Park, and Erik G Hoel. 2019. Distributed spatial and spatio-temporal join on apache spark. *ACM TSAS* 5, 1 (2019), 1–28.

[36] Randall T Whitman, Michael B Park, Bryan G Marsh, and Erik G Hoel. 2017. Spatio-temporal join on apache spark. In *ACM SIGSPATIAL*. 1–10.

[37] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *ACM SIGMOD*. 1071–1085.

[38] Keyu Yang, Xin Ding, Yuanliang Zhang, Lu Chen, Baihua Zheng, and Yunjun Gao. 2019. Distributed similarity queries in metric spaces. *DSE* 4, 2 (2019), 93–108.

[39] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in cloud. In *ICDE workshops*. IEEE, 34–41.

[40] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *ACM SIGSPATIAL*. 1–4.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. 15–28.

[42] Chi Zhang, Feifei Li, and Jeffrey Jestes. 2012. Efficient parallel knn joins for large data in mapreduce. In *EDBT*. 38–49.

[43] Qing Zhu, Jun Gong, and Yeting Zhang. 2007. An efficient 3D R-tree spatial index method for virtual geographic environments. *ISPRS* 62, 3 (2007), 217–224.

A SYMBOLS AND THEIR MEANINGS

For the purpose of reference, Table 4 lists the symbols and their meanings used frequently in this paper.

Table 4: Symbols and Their Meanings

Symbol	Meaning
R (resp. S)	an ST-object set R (resp. S)
r (resp. s)	an ST-object $r \in R$ (resp. $s \in S$), where $r.geom$ is a spatial attribute, $r.tr = [t_{min}, t_{max}]$ is a time range
$MBR(r)$	the minimum bounding box of r
$EMBR(r, \gamma)$	the expanded minimum bounding rectangle of r w.r.t a distance threshold γ
$TD(R)$	the temporal domain of R
$SD(R)$	the spatial domain of R
$ETR(tr, \delta)$	expanded time range of tr with a time threshold δ
ST-kNN (r, k, δ, S)	the spatio-temporal k nearest neighbors of r from S with a time threshold δ
$d(r, s)$, $d(p, q)$	the distance between r and s, and the Euclidean distance between two spatial points p and q
$R \ltimes S$	ST-kNN join of R and S
GT, GS	global temporal domain, global spatial domain
η, α, β	sampling rate, maximum number of temporal partitions, maximum number of spatial partitions
tp, sp, GI	temporal partitions, spatial partitions, global index
TRC-index	temporal range count index
$binNum$	bin number in a TRC-index
TRP, SRP	temporal reference point, spatial reference point

B DETAILS OF COMPLEXITY ANALYSIS

In this section, we give more details of complexity analysis.

Data Partition for S. Recall that this step can be divided into four sub-steps: 1) *Sampling*, 2) *Temporal Partitioning*, 3) *Spatial Partitioning*, and 4) *Reassignment*.

The cost of *Sampling* can be ignored, as the number of samples is rather small and there is no other computation cost.

For *Temporal Partitioning* (i.e., Algorithm 1), it first sorts the samples in S', then scans them only once. Therefore, the overall computation complexity of *Temporal Partitioning* is $O(|S'| \times log|S'|)$.

The most time-consuming part of Algorithm 2 is the `While` loop. As in each iteration, we add 3 more spatial partitions, thus there is at most $\beta/3$ iterations. In each iteration, we need to scan at most $|S_i'|$ samples, and each new sub-node takes at most $log\beta$ to be inserted into pq. As a result, the computation complexity of Algorithm 2 is $O(\beta/3 \times (|S_i'| + 4 \times log\beta))$. Because $S' = S_1' \cup S_2' \cup \ldots \cup S_n'$, $n \leq \alpha$, the overall computation cost of spatial partitioning is $O(\beta/3 \times |S'| + 4/3 \times \alpha \times \beta \times log\beta)$.

For *Reassignment*, it incurs a shuffle of S, thus it can be a bottleneck of ST-kNN join. As the size of global index GI is very small, the overhead of broadcast can be ignored. Using the global index GI, each $s \in S$ can find the targeted partitions in $O(log\alpha \times log\beta)$. Therefore, the time complexity of this step is $O(|S| \times log\alpha \times log\beta)$.

First Round Local Join. Note that this step consists of three sub-steps: 1) *TRC-index Construction*, 2) *Data Partition for R*, and 3) *Distance Bound Calculation*.

TRC-index Construction method (i.e., Algorithm 3) scans linearly objects of S_i and the two arrays (T_{min} and T_{max}) for once. As $binNum$ is relatively much smaller than S_i, the overall computation for all ST-partitions is $O(|S|)$. The search complexity using TRC-index is $O(1)$.

For each $r \in R$, we find u satisfied temporal partitions in $O(log\alpha)$. For each satisfied temporal partition, we find the target ST-partition in $O(log\beta)$. Consequently, the overall computation complexity of this sub-step is $O(|R| \times log\alpha \times u \times log\beta)$. This step triggers a shuffle of R, thus it could be a bottleneck.

For *Distance Bound Calculation*, building a local R-tree index takes $O(|S_i| \times log|S_i|)$. The time complexity of finding the ST-kNNs of R_i using R-tree is highly dependent on the data distribution. In most cases, it can be done with $O(|R_i| \times log|S_i|)$.

Second Round Local Join. There is only a small number of objects $r \in R$ that participate in the second round local join. For each $r \in R$, it takes the same time with that in the first round local join.

Merge Result. This step incurs a shuffle of local results. Suppose each $r \in R$ is bounded to v ST-partitions, thus the multiway merge algorithm takes $O(v \times k)$. The overall time complexity of merge result is $O(|R| \times v \times k)$.

C DEMONSTRATION

We integrate the ST-kNN join method proposed in this paper into JUST [24], a distributed spatio-temporal data engine. As a result, we can perform ST-kNN join with a SQL-like statement:

SELECT * **FROM** R, S **WHERE** st_knnjoin($R.geom$, $S.geom$, $R.t_{min}$, $R.t_{max}$, $S.t_{min}$, $S.t_{max}$, k, δ)

where R and S are the names of two tables, and $R.geom$, $S.geom$, $R.t_{min}$, $R.t_{max}$, $S.t_{min}$, $S.t_{max}$ are the spatio-temporal field names of the two tables, respectively.

Figure 12: User Interface of JUST for ST-kNN Join [2].

Figure 12 shows the user interface of ST-kNN join in JUST. It consists of three panels: *Table Panel*, *JustQL Panel* and *Result Panel*. Table Panel lists the tables in the system. In this demo, we preset six tables of spatio-temporal objects with various geometry types. The objects are sampled from the datasets of DidiTraj and DidiSP. Users can also upload their own datasets to JUST. In the JustQL Panel, we input a SQL-like statement, and click the first left-top button to run ST-kNN join. Here, we perform an ST-kNN join on two point tables $point01$ and $point02$, where $k = 2$ and $\delta = 100$s. The join result is shown in Result Panel. Readers can visit the public website [2] to experience ST-kNN join, or download the source code from the website and run it on their own clusters.