Learning-Based Distributed Spatio-Temporal k Nearest Neighbors Join

Ruiyuan Li, Jiajun Li, Minxin Zhou, Rubin Wang, Huajun He, Chao Chen, Jie Bao, Yu Zheng *Fellow, IEEE*

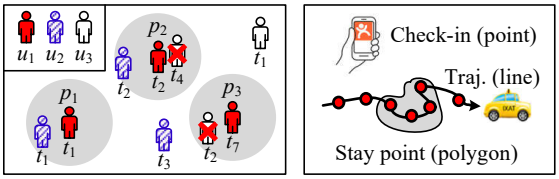
***Abstract*—The rapid development of positioning technology produces an extremely large volume of spatio-temporal data with various geometry types such as point, line string, polygon, or a mixed combination of them. As one of the most fundamental but time-consuming operations, k nearest neighbors join (kNN join) has attracted much attention. However, most existing works for kNN join either ignore temporal information or consider only point data. Besides, most of them do not automatically adapt to the different features of spatio-temporal data.**

This paper proposes to address 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 large 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. Furthermore, we design a set of models based on Bayesian optimization to automatically determine the values for the introduced parameters. Extensive experiments are conducted using three real big datasets, showing that our method is much more scalable and achieves 9X faster than baselines, and that the proposed models can always predict appropriate parameters for different datasets. The source codes are released at https://github.com/Spatio-Temporal-Lab/stknnjoin.

***Index Terms*—Distributed computing, spatio-temporal kNN Join, knob tuning, Bayesian optimization, AI4DB.**

I. Introduction

With the rapid development of positioning technology, an extremely large number of spatio-temporal data are generated. Among spatio-temporal data analyses, k nearest neighbors join (kNN join) [2]–[6] is one of the most fundamental operations, which is crucial in many applications. As shown in Fig. 1(a), in the case of epidemic prevention [7], 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 only the *spatial closeness*, 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 appeared at different times (i.e., t_2 and t_4, respectively). Similarly, u_3 is not the nearest to p_3. In the end, only u_2 is the potentially suspected user, which brings in a more precise epidemic prevention.



(a) Epidemic Prevention (b) Various Geometry Types

Fig. 1. Motivation of ST-kNN Join.

This paper proposes Spatio-Temporal k Nearest Neighbors join (i.e., ST-kNN join) that considers both spatial closeness and temporal concurrency. It can be applied to many other applications such as ride-sharing [8]–[11], companion detection [12]–[14] and travel recommendation [15]–[17], all of which cannot ignore the temporal information.

It is challenging to perform ST-kNN join for four reasons. 1) **Big Data**. Spatio-temporal data are generated constantly at a high frequency, leading to a prohibitively large volume of data. In contrast, ST-kNN join itself is rather time-consuming. Merely extending the standalone spatial-related join solutions [18]–[20] cannot handle such big spatio-temporal data efficiently. 2) **High Dimensionality**. In addition to spatial information, we should also consider the temporal information, which is more intractable. 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). It sometimes requires to perform ST-kNN join on all geometry types or even a mixture of them. But it is difficult to design a unified framework that supports ST-kNN join for all geometry types. 4) **Unbalanced and Various Distribution**. The distribution of spatio-temporal data is highly skewed. For example, downtown areas contain more taxi trajectories than suburb areas, and more taxi trajectories can be discovered during rush hours than other time periods [22]. Different spatio-temporal data have different distributions that affect the efficiency of ST-kNN join. To achieve efficient ST-kNN join, the system parameters

Manuscript received September 30, 2023; revised January 25, 2024; accepted July 23, 2024. This paper is supported by the National Natural Science Foundation of China (62202070, 62322601, 62172066, 62076191), China Postdoctoral Science Foundation (2022M720567) and the Excellent Youth Foundation of Chongqing (CSTB2023NSCQJQX0025). Recommended for acceptance by xxx.

This paper is an extension version of the conference paper [1].

Ruiyuan Li, Jiajun Li, Minxin Zhou and Chao Chen are with Chongqing University, China (e-mail: ruiyuan.li@cqu.edu.cn; jiajunli@stu.cqu.edu.cn; minxinzhou@stu.cqu.edu.cn; cschaochen@cqu.edu.cn).

Rubin Wang, Huajun He, Jie Bao and Yu Zheng are with JD iCity, China and JD Intelligent Cities Research, China. Yu Zheng is also with Xidian University, China. (e-mail: wangrubin3@jd.com; hehuajun@my.swjtu.edu.cn; baojie@jd.com; msyuzheng@outlook.com).

should be fine-tuned for every request, which is intractable.

Over the last decade, there emerged many distributed frameworks, e.g., Apache Hadoop [23] and Apache Spark [24], which cope with big data efficiently. Many distributed works [2]–[6] for kNN join ignore the temporal information, therefore they cannot be applied to ST-kNN join directly. Besides, most of them [2], [4]–[6] 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. Moreover, most works [4]–[6] do not provide adaptive techniques to determine some important system parameters (a.k.a. knobs) for different datasets. It is hard if not impossible for users to fine-tune good knobs manually for every request. Other systems such as Simba [2] and LocationSpark [3], [25] deploy cost models for some spatial or spatio-temporal operations, but not for ST-kNN join.

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, and then merge them into a global one. Moreover, we design a set of models based on Bayesian optimization to automatically determine good values for the introduced system parameters. As a result, our method is adaptive to different features of various spatio-temporal datasets. ST-kNN join is deployed to the product JUST (JD Urban Spatio-Temporal data engine) [26]–[32], where the spatio-temporal data is stored in a NoSQL database with carefully designed indexes. In JUST, users can execute ST-kNN join with a standard SQL statement. Overall, the contributions of this paper are five-fold:

(1) This paper proposes a novel and useful ST-kNN join problem, and presents a distributed solution that supports ST-kNN join with any geometry types 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) We design a set of models based on Bayesian optimization to determine the values of introduced system parameters automatically, which guarantees that our ST-kNN join method performs well for various datasets with different distributions, different sizes and various geometry types.

(4) We deploy ST-kNN join to JUST, with which an ST-kNN join can be performed with a standard SQL statement.

(5) Extensive experiments are carried out using three real datasets, which verifies the efficiency and scalability of our method and the effectiveness of the knob tuning models. All source codes are released [33].

TABLE I
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
$TD(R), SD(R)$	the temporal domain and 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

Outline. We give some preliminaries in Section II, and describe the overview of ST-kNN Join Executor in Section III. The details of ST-kNN Join Executor are presented in Section IV. We propose knob tuning models in Section V. In Section VI, we analyze the performance of ST-kNN join theoretically. We present the evaluation results in Section VII, followed by the implementation details in JUST in Section VIII. We review the related works in Section IX. Finally, we conclude this paper in Section X. Table I lists the symbols and their meanings used frequently throughout this paper.

II. PRELIMINARY

A. Definition

Definition 1. (Clustering) ST-object (spatio-temporal object) $r = (geom, tr)$ contains a spatial attribute $geom$ and a time range $tr = [t_{min}, t_{max}]$, where $geom$ can be any geometry (e.g., a point, a line string, a polygon, etc., or a mixed set of them). 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, we call an ST-object **object** for simplicity.

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 $MBR(r) = \langle (lat_{min}, lng_{min}), (lat_{max}, lng_{max}) \rangle$. 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 $ETR(tr, \delta) = [t_{min} - \delta, t_{max} + \delta]$.

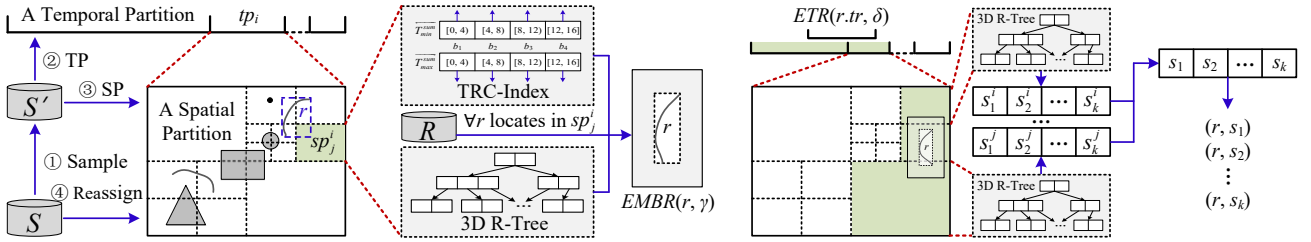


Fig. 2. Overview of ST-kNN Join Executor.

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 $|S'| \leq k$, 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 $\forall s_i \in S', \forall s_j \in S'' \backslash S', d(r, s_i) < d(r, s_j)$.

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

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

where $d_{eu}(p, q)$ is the Euclidean distance between two spatial points p and q.

Discussion. The temporal gap δ is defined because in many real applications such as ride-sharing [8], users would have a tolerance for some time deviation (e.g., 15 minutes). In fact, the temporal concurrency with a gap is more general for various applications. Besides, we do not combine spatio-temporal dimensions into a single distance metric using a linear combiner with different weights [34]. Because 1) the temporal dimension has a rather 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, respectively.

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,

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

We find that 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.

B. Apache Spark

Apache Spark [24] is an in-memory distributed framework for large-scale data processing. 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 persistent on disk to accelerate data reusing and support iteration. In Spark, we can broadcast variables 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, so we should try to avoid it when possible. In a Spark cluster, there are two types of nodes, i.e., master node and slave node. In the master node, there is a driver program to submit Spark jobs or broadcast variables. In a slave node, there are several executors. Each executor processes multiple partitions parallelly, whose degree of parallelism is dependent on the CPU cores assigned to the executor.

III. Overview

Figure 2 presents the framework of our proposed ST-kNN Join Executor, 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 (i.e., 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 based on $s \in S$ that locates in this ST-partition. 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 shown in Fig. 2(c), we check 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.

IV. ST-kNN Join Executor

A. 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 both

spatially and temporally should be assigned to the same partition as many as possible, so 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, therefore we can achieve load balance.

Existing distributed frameworks for spatial data processing either focus on spatial partitioning merely [2], [3], 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. Since S' is sampled randomly from S, it keeps the spatio-temporal distribution of S. Then S' is collected to the driver program on the master node, where we would construct spatio-temporal partitions based on the samples. We take $\eta = 1\%$ as Simba [2] 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 that has a significant impact on the performance of ST-kNN join. In this section, we assume that α is given, and in Section V, we will deploy knob tuning models to determine a good value of α for each request. The effects of different values of α will be shown in Section VII.

The time span of a temporal partition has a significant impact on ST-kNN join. On one hand, 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 $s \in S'$). Moreover, 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 $ETR(r, \delta)$. On the other hand, however, during the join process, 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 as small as possible.

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 number of objects in S'. $|S'|/\alpha$ guarantees the number of temporal partitions is no more than α.

We propose a novel temporal partitioning method based on Sweep Line Algorithm [37], which forms temporal partitions one by one. 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). $minSpan$ and $minNum$ are the minimum time span of a temporal partition and the minimum number of objects in a temporal partition, respectively (Line 3). In Lines 4-8, we scan S' from left to right. Once the current temporal partition satisfies both Equ. (4) and Equ. (5), it forms a final temporal partition and is added to tps. Those objects $s' \in cur$ that will not contribute to the next temporal partition (i.e., $s'.tr.t_{max} < start$) are filtered out. Finally, we process the last temporal partition and return the final results (Line 9).

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

```
1 Sort S' by the start time of objects in ascending order;
2 tps = ∅; cur = ∅; start = GT.t_min; sl = GT.t_min;
3 minSpan = max{|s.tr|, 2δ + |r.tr|}; minNum = |S'|/α;
4 for s ∈ S' do
5     sl = s.tr.t_min; cur = cur ∪ {s}; span = sl − start;
6     if span ≥ minSpan and |cur| ≥ minNum then
7         tps = tps ∪ {[start, sl]}; start = sl;
8         Filter out s' ∈ cur that s'.tr.t_max < start;
9 return tps ∪ {[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 [38] based on the samples S_i' assigned to tp_i. As each spatial partition belongs to a temporal partition, we call each spatial partition an **ST-partition**. Like α, β is a system parameter as well, which imposes significant effects on the performance of ST-kNN join. We will assign an appropriate value to it based on the parameter models proposed in Section V. The impact of β will be tested in Section VII.

This paper adopts Quad-tree [38] to perform spatial partitioning for three reasons. **Firstly**, Quad-tree can mitigate the unbalanced problem of spatial distribution compared with Grid partition [39], [40], as Quad-tree partitions the areas with denser objects into smaller regions. **Secondly**, compared with R-tree [41] and its variants [42], [43], 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, 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 [44], 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:

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

where $|S'_i|/\beta$ is the average samples number in a spatial partition. $4k \times \eta \times |tp_i| \div (2\delta + \overline{|r.tr|})$ ensures that after a split, at least one of its sub-regions is expected to have more than k satisfied objects. Here, k is multiplied by $\eta \times |tp_i| \div (2\delta + \overline{|r.tr|})$, aiming to linearly scale to the satisfied object number in S'_i. Figure 2(a) gives an example of spatial partitioning, where $\zeta = 3$. Note that r itself does not intersect with sp^i_j, but it is copied to sp^i_j because $MBR(r) \cap sp^i_j \neq \emptyset$.

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. ζ is initialized in accordance with Equ. (6) (Lines 1-2). Then we check all nodes in pq in a descending order of sample numbers (Lines 3-7). 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 (Line 8).

Algorithm 2: SP(S'_i, GS, k, β, η)

```
1 Initialize a priority queue pq, with keys as the sample
    numbers in Quad-tree nodes, sorted in a descending
    order; pq.push(GS);
2 ζ = max{|S'_i|/β, 4k × η × |tp_i| ÷ (2δ + |r.tr|)};
3 while pq.length < β do
4     node = pq.pop();
5     if samples(node) < ζ then
6         pq.push(node); break;
7     Split node into 4 sub-nodes, and add them into pq;
8 return the nodes in pq as spatial partitions;
```

Reassignment. After the previous two steps, we get at most $\alpha \times \beta$ ST-partitions. Each ST-partition is bound 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 bound to s. After that, S is re-partitioned according to the bound 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.

B. 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 $EMBR(r, \gamma)$. It is challenging 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 before scanning it. For the first challenge, we check all intersected ST-partitions to find the nearest one to r. For the second challenge, we propose a novel 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 sub-steps: 1) *TRC-index Construction*, 2) *Data Partition for R*, and 3) *Distance Bound Calculation*.

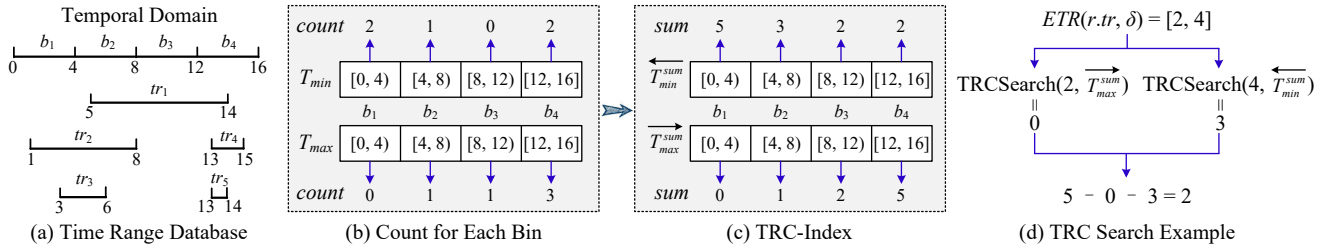
TRC-index Construction. There are two requirements for TRC-index. 1) Given a set of objects S_i and a time range tr, it returns efficiently the minimum number of objects in S_i whose time ranges intersect with tr. 2) TRC-index should be as small as possible, because it will be broadcast to all partitions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

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 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, and each value is set 0;
2 binLen = ⌈|TD(S_i)|/binNum⌉; rt = TD(S_i).t_min;
3 for s ∈ S_i do
4     j_1 = ⌊(s.tr.t_min − rt)/binLen⌋; T_min[j_1] ++;
5     j_2 = ⌊(s.tr.t_max − rt)/binLen⌋; 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 ⟨TD(S_i), |S_i|, binNum, T_min, T_max⟩;
```

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). Then we calculate the length of each bin, and set a reference time as the minimum time of S_i's temporal domain for calculating the bin number



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

Fig. 3. Illustration of TRC-Index.

(Line 2). 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 IDs, 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 $tr.t_{min}$, as we use discrete bins to approximate continuous times). 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. A knob tuning model is deployed to decide $binNum$ for every ST-kNN join in Section V. We will investigate its effect on ST-kNN join in Section VII.

For example, given a time range database shown in Fig. 3(a), we count the objects in each bin in Fig. 3(b) (here $binNum = 4$), and 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 IV-A, 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. In 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 bound 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 having 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 3D R-tree index [45] on 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 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 (i.e., Data Partition for R), 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.

C. 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 get 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 bound 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

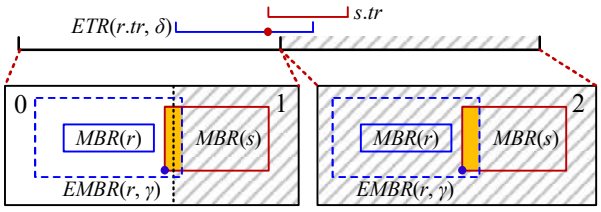


Fig. 4. Remove Duplicates.

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 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 only, because 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.

D. 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 [46]; 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 ST-kNN result of r.

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

Lemma 1. *The proposed duplicate removal method 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 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_j^i that the TRP and SRP locate in, therefore (r, s) must be produced in sp_j^i 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 $sp_m^i \cap sp_n^i = \emptyset$ if $m \neq n$. Suppose there exist two different ST-partitions sp_m^i and sp_n^j containing TRP and SRP simultaneously, i.e. TRP $\in tp_i \cap tp_j$ and SRP $\in sp_m^i \cap sp_n^j$. If $i \neq j$, TRP $\in tp_i \cap tp_j$, which contradicts with the temporal partitioning strategy (i.e., no two temporal partitions intersect with each other). If $i = j$ and $m \neq n$, SRP $\in sp_m^i \cap sp_n^i$, which is contradictory to the spatial partitioning strategy (i.e., no two spatial partitions in one temporal partition intersect with each other). □

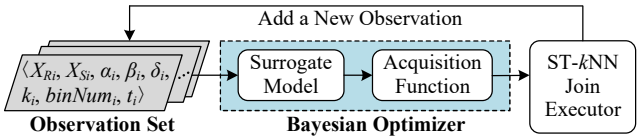


Fig. 5. Traditional Bayesian Optimization.

V. Automatic Knob Tuning

We introduce three system parameters (i.e., α, β and $binNum$) that will impose great effects on the efficiency of ST-kNN join (see Section VII). It is intractable to fine-tune them manually for every request. To this end, we propose an automatic knob tuning framework based on Bayesian optimization to determine appropriate values for the introduced system parameters. In what follows of this section, we start by discussing the knob tuning problem and traditional Bayesian optimization, and then elaborate on the main components of the proposed automatic knob tuning framework.

A. Knob Tuning Based on Traditional Bayesian Optimization

Knob tuning [47] is to identify well-performing configuration settings within a prescribed dataset and under the limitation of system resources. Formally, knob tuning can be defined as an optimization problem:

$$\vec{x}^* = \arg\min_{\vec{x} \in \Omega} f(\vec{x}) \tag{7}$$

where $\vec{x}$ is a vector of knobs, Ω is the search space, and the value of $f(\vec{x})$ is the optimization target under the fixed input data and system resources. In our ST-kNN join problem, $\vec{x} = (\alpha, \beta, binNum)$ and $f(\vec{x})$ is the execution time of a request. Note that it is not straightforward to build a function mapping a given $\vec{x}$ to the execution time unless we actually perform the ST-kNN join operation.

Bayesian optimization [48], [49] is widely-used in automatic database knob tuning when the gradients of $f(\vec{x})$ are unknown. It is particularly suitable for the scenarios where the knob number is small (e.g., 5-20 knobs). Another advantage of Bayesian optimization, compared with Deep Learning-based methods [48], [50], [51], is that it requires fewer samples to achieve high-quality knob settings. Figure 5 exhibits the main steps of knob tuning with traditional Bayesian optimization,

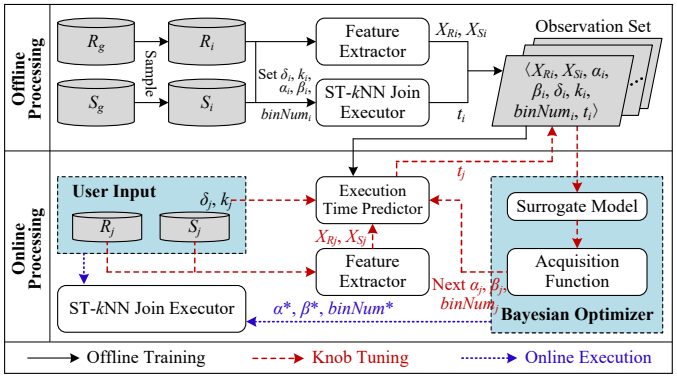


Fig. 6. Proposed Framework of Automatic Knob Tuning.

which is a sequential iterative process. Suppose there is a set of observations (i.e., training samples), where each observation $\langle X_{R_i}, X_{S_i}, \alpha_i, \beta_i, \delta_i, k_i, binNum_i, t_i\rangle$ represents that the execution time of an ST-kNN join request on two datasets R_i and S_i with the query parameters (δ_i, k_i) and knob settings $(\alpha_i, \beta_i, binNum_i)$ is t_i. X_{R_i} and X_{S_i} are the features (see next sub-section) of R_i and S_i, respectively. Bayesian Optimizer consists of two key components: **surrogate model** and **acquisition function**. First, the surrogate model approximates the function $f(\vec{x})$ and sequentially updates itself with new observations. Second, the acquisition function decides where to sample a new parameter setting, i.e., $(\alpha_{i+1}, \beta_{i+1}, binNum_{i+1})$. Third, the new parameter setting is adopted by the ST-kNN join executor on the same datasets and query parameters with the previous iteration (i.e., $R_{i+1} = R_i$, $S_{i+1} = R_{i+1}$, $\delta_{i+1} = \delta_i$ and $k_{i+1} = k_i$), obtaining the execution time t_{i+1} and forming a new observation. Fourth, the new observation is added into the observation set. The four steps repeat until the maximum number N_{BO} of iterations is reached. Finally, the settings corresponding to the minimum execution time are returned.

B. Proposed Framework of Automatic Knob Tuning

The steps shown in Fig. 5 are performed during the online query processing. Since it is relatively time-consuming for ST-kNN join, if we execute an ST-kNN join in every iteration, the inference time would be intolerable. To this end, we replace the ST-kNN join executor in Fig. 5 with a learning-based model, which predicts the execution time instead of actually performing ST-kNN join operations. Ultimately, our proposed framework of automatic knob tuning is shown as Fig. 6, which consists of two main phases (i.e., Offline Processing and Online Processing) with three data streams (i.e., Offline Training, Knob Tuning and Online Execution). In Offline Processing phase, we collect a set of observations, with which we train an Execution Time Predictor. In Online Processing phase, after a user invoke a request, we first detect an appropriate knob setting $(\alpha^*, \beta^*, binNum^*)$ based on Bayesian optimization, and then perform an ST-kNN join with the learned setting.

C. Offline Processing

As shown in the upper part of Fig. 6, we first **sample** two sub-datasets R_i and S_i from the global datasets. R_i and S_i are fed into two modules: 1) **Feature Extractor** that extracts features X_{R_i} and S_{S_i} of R_i and S_i, and 2) **ST-kNN Join Executor** that performs ST-kNN Join operation on R_i and S_i with specified parameters $(\delta_i, k_i, \alpha_i, \beta_i, binNum_i)$, observing the execution time t_i. The previous steps are repeated until we get enough observations, with which we train an **Execution Time Predictor**. The aforementioned process would be repetitively performed whenever the system resources are idle; thus, Execution Time Predictor could be periodically updated.

A_{SD}	I_{TD}	C	Spatial Distribution	Temporal Distribution
1	1	1	h_s	h_t

Fig. 7. Features with Dimensions.

Sample. To diversify the features within the data samples, we employ specific rules and criteria for the sampling process. For example, we sample sub-datasets of varying cardinalities, such as 50,000, 100,000, 150,000, and so on. Furthermore, within each scale of sampling, we further subdivide the process with different temporal and spatial coverages. This approach is to ensure that the knowledge accumulated during Offline Processing can effectively accommodate the diverse ranges of data inputs from users.

Feature Extractor. Feature extraction is to reduce the dimensionality and minimize the computational time on the relevant feature sets. We select features including the area A_{SD} of spatial domain SD, the interval I_{TD} of temporal domain TD, cardinality C, and spatio-temporal distribution of R_i (or S_i). To represent its spatial distribution, we partition SD into equal-sized and disjoint $a \times b$ regions (in our implementation, $a = 10$ and $b = 10$), and record the numbers of objects overlapped with each region. In this manner, we obtain an $(a \times b)$-dimensional spatial distribution vector. We do not perform normalization here since the spatial partitioning process is directly related to the number of objects, rather than the relative value. To reduce the subsequent computational overhead and alleviate the sparsity issue of features, we employ Principal Component Analysis (PCA) [52] to map the high-dimensional features to h_s dimensions. Similarly, we extract temporal distribution features with h_t dimensions. Ultimately, we obtain a vector of features with $h = (3 + h_s + h_t)$ dimensions, as shown in Fig. 7 (in our implementation, $h_s = 25$ and $h_t = 30$).

ST-kNN Join Executor. ST-kNN Join Executor receives R_i and S_i along with specified parameters and executes the ST-kNN join, observing the execution time t_i.

Execution Time Predictor. Traditional Bayesian Optimizer necessitates the evaluation of a set of parameters through actual execution, which could be time-consuming. Therefore, we devise the Execution Time Predictor to predict the execution time and provide feedbacks to the Bayesian Optimizer, which can significantly save time. During Offline Processing, we obtain a set of observations to train a regression model. We opt for the XGBoost regression model [53] due to its efficiency in handling large-scale data and its excellent performance in preventing overfitting. In our system, the Execution Time Predictor will be periodically updated on the dynamic observation set. Additionally, our focus lies in forecasting execution

time rather than directly predicting parameter values, which allows us to include essential parameter values themselves as input, thereby enhancing accuracy. Meanwhile, due to the minimal time overhead associated with model predictions and the search strategy, the time cost incurred is also minimal.

D. Online Processing

As shown in the lower part of Fig. 6, when a user issues an ST-kNN join request, we first search for an appropriate knob setting using Bayesian Optimizer (denoted as **Knob Tuning** stream in dashed red), and then actually perform an ST-kNN join operation with the learned knob setting (denoted as **Online Execution** stream in dotted blue).

Knob Tuning. When a user issues an ST-kNN join request, the input data R_j and S_j are sent to the Feature Extractor (same with the one described in Offline Processing), getting two feature vectors X_{R_j} and X_{S_j}. Simultaneously, the Bayesian Optimizer initializes a Surrogate Model to fit the data from the Observation Set, and then employs the Acquisition Function to determine the next search point $(\alpha_j, \beta_j, binNum_j)$. After that, the new search point $(\alpha_j, \beta_j, binNum_j)$, features X_{R_j} and X_{S_j}, and query parameters δ_j and k_j are fed into the Execution Time Predictor, predicting the execution time t_j and forming a new observation. The previous steps are repeated N_{BO} times, and the setting $(\alpha_j^*, \beta_j^*, binNum_j^*)$ corresponding to the minimum execution time is returned.

(1) Surrogate Model. Since prior knowledge will be updated with new observations, we employ Gaussian Processes (GP) [54], [55] to construct the Surrogate Model, because GP can obtain the distribution of the implicit function based on the updated prior knowledge, resulting in a better fit to the data points. A GP builds a function distribution f_l based on l data points $D_{1:l} = \{(\vec{x}_i, t_i)\}, i = 1, \ldots, l$, where $t_i = f_l(\vec{x}_i) + \epsilon_i$ with $\epsilon_i \sim \mathcal{N}(0, \sigma_{noise}^2)$ as the noisy observation of the objective function at $\vec{x}_i$. Let $\vec{t}_{1:l} = [t_1\ t_2\ \ldots\ t_l]$. Since f_l follows a joint Gaussian distribution, when a new data point $(\vec{x}_{l+1}, t_{l+1})$ is observed, conditional on the previous observed data $D_{1:l}$, we can subsequently derive the predictive distribution of f_{l+1}:

$$P(f_{l+1}|D_{1:l}, \vec{x}_{l+1}) \sim \mathcal{N}(\mu_l(\vec{x}_{l+1}), \sigma_l^2(\vec{x}_{l+1}) + \sigma_{noise}^2) \quad (8)$$

where

$$\mu_l(\vec{x}_{l+1}) = \vec{k}^T[\boldsymbol{K} + \sigma_{noise}^2 \boldsymbol{I}]^{-1}\vec{t}_{1:l} \quad (9)$$

$$\sigma_l^2(\vec{x}_{l+1}) = kernel(\vec{x}_{t+1}, \vec{x}_{t+1}) - \vec{k}^T[\boldsymbol{K} + \sigma_{noise}^2 \boldsymbol{I}]^{-1}\vec{k} \quad (10)$$

$$\vec{k} = [kernel(\vec{x}_{t+1}, \vec{x}_1)\ \ldots\ kernel(\vec{x}_{t+1}, \vec{x}_l)] \quad (11)$$

$$\boldsymbol{K} = \begin{bmatrix} kernel(\vec{x}_1, \vec{x}_1) & \cdots & kernel(\vec{x}_1, \vec{x}_l) \\ \vdots & \ddots & \vdots \\ kernel(\vec{x}_l, \vec{x}_1) & \cdots & kernel(\vec{x}_l, \vec{x}_l) \end{bmatrix} \quad (12)$$

Here, $kernel(\vec{x}_i, \vec{x}_j)$ is the kernel function that returns the covariance matrix of two vectors. We adopt the popular Matérn covariance function [55] as the kernel because of its flexibility.

(2) Acquisition Function. We employ Expected Improvement (EI) strategy [55] to help determining the next search point, because compared with Probability of Improvement (PI) strategy [55], EI strategy considers both the probability of improvement and the magnitude of the improvement a point can potentially yield. In our problem, the improvement refers to the execution time reduced. Suppose f_l is built based on $D_{1:l} = \{(\vec{x}_i, t_i)\}$, and $\vec{x}^* = \arg\min_{\vec{x}_i \in D_{1:l}} f_l(\vec{x}_i)$. EI strategy aims to find $\vec{x}_{l+1}$ that maximizes the following equation:

$$EI(\vec{x}_{l+1}) = \begin{cases} (f_l(\vec{x}^*) - \mu_l(\vec{x}_{l+1}))\Phi(Z) + \sigma_l(\vec{x}_{l+1})\phi(Z), & \text{if } \sigma_l(\vec{x}_{l+1}) > 0 \\ 0, & \text{if } \sigma_l(\vec{x}_{l+1}) = 0 \end{cases} \quad (13)$$

where $Z = (f_l(\vec{x}^*) - \mu_l(\vec{x}_{l+1}))/\sigma_l(\vec{x}_{l+1})$, $\phi(\cdot)$ and $\Phi(\cdot)$ denote the PDF (probability density function) and CDF (cumulative distribution function) of the standard normal distribution respectively, and $\mu_l(\vec{x}_{l+1})$ and $\sigma_l(\vec{x}_{l+1})$ can be calculated by Equ. (9) and Equ. (10) respectively.

Online Execution. After finishing the Knob Tuning process, we obtain an optimized setting $(\alpha^*, \beta^*, binNum^*)$, which is passed into ST-kNN Join Executor along with the user's input. The executor is responsible to finish the join operation. The execution would be recorded as a new observation for updating the predictive model and future Bayesian optimization.

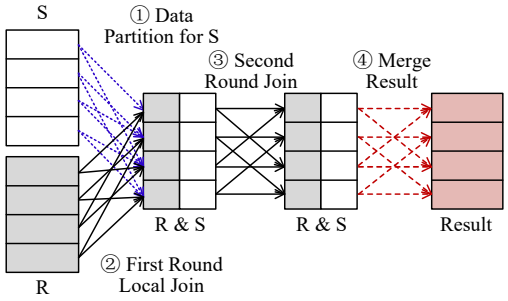


Fig. 8. Shuffles of RDD.

VI. Performance Analysis

We first discuss the computation complexity of ST-kNN Join Executor, and then analyze the performance of knob tuning.

A. Complexity Analysis of ST-kNN Join Executor

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 8 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 VII), 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.

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 two rounds local join. Finally, it takes $O(|R|\times k\times logv)$ to merge the results, where v is the average number of ST-partitions an r locates in. More details are given bellow.

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 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 $S' = S'_1 \cup S'_2 \cup ... \cup S'_n$, $n \leq \alpha$, 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 in $O(|R_i|\times log|S_i|)$.

Second Round Local Join. There is only a small number of objects $r \in R$ that participate in the second round local join, taking 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 bound to v ST-partitions, thus the multiway merge algorithm takes $O(v\times k)$. The overall time complexity of merge result is $O(|R|\times v\times k)$.

B. Performance Analysis of Knob Tuning

The knob tuning framework consists of two main parts: offline processing and online processing. Offline processing is performed in advance, so it does not affect the performance of online ST-kNN join. Online processing is further divided into two streams: Online Execution and Knob Tuning. The performance of Online Execution has been analyzed previously. In Knob Tuning, feature extraction scans R and S once, and most importantly, it is only performed once, so it has little impact on the whole performance. Bayesian optimization performs N_{BO} iterations in total. In each iteration, we replace the most time-consuming ST-kNN Join Executor with the efficient Execution Time Predictor. Since the dimension of parameters is 3 and Bayesian Optimizer can effectively guide the optimization direction, we can achieve an excellent setting with a limited number of N_{BO} (usually less than 50 as shown in Section VII). To this end, the time of knob tuning can be negligible.

TABLE II
STATISTICS OF DATASETS

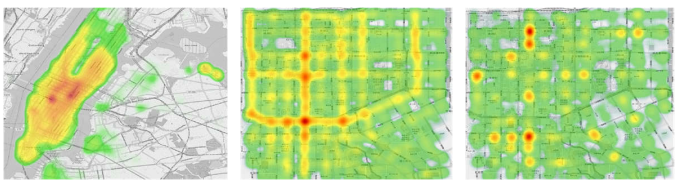
Attributes	NYTrip	DidiTraj	DidiSP
Raw Size	11.6GB	8.3GB	1.9GB
# Records	87,110,491	39,224,513	9,108,396
# Coords	174,220,982	348,191,629	73,708,681
Temporal Domain	2013/01/01 - 2013/06/30	2018/10/01 - 2018/11/30	2018/10/01 - 2018/11/30
Spatial Domain	(-74.07 : -73.75), (40.61 : 40.87)	(108.92 : 109.01),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

VII. EVALUATION

A. Datasets and Settings

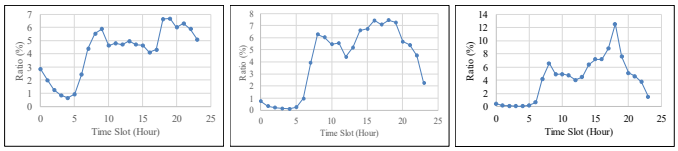
Datasets. We use three real big datasets to verify the performance of ST-kNN join: 1) **NYTrip** [56]. We extract six months of taxi trips in New York City. Each trip has the location and time information of pick-up and drop-off, respectively. The locations of a pick-up or a drop-off can be modeled as *point* data (abbr. **pt**); 2) **DidiTraj** [57], 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 [58]. A stay point is deemed as a *polygon* (abbr. **py**). Table II shows the statistics.

Figure 9 and Figure 10 show the spatial and temporal distributions of the datasets, respectively. We can observe that, 1) for each single dataset, the spatio-temporal distribution is severely unbalanced, i.e., some spatial areas contain much more records than others, and there are obvious temporal peaks and valleys; 2) among different datasets, their spatio-temporal distributions are not the same. Particularly, DidiSP is extracted from DidiTraj, but their spatio-temporal distributions are quite different. Unbalanced and various distributions of different spatio-temporal datasets make it intractable to fine-tune parameters for all ST-kNN join requests manually.



(a) NYTrip (b) DidiTraj (c) DidiSP

Fig. 9. Spatial Distribution.



(a) NYTrip (b) DidiTraj (c) DidiSP

Fig. 10. Temporal Distribution.

Settings. Table III shows the geometry combinations for investigating the effects of the introduced parameters. To train an Execution Predictor, we use 80% data as the train set, and the rest as the test set. For each combination in Table III, we extract 2,000 observations and train a single Execution Predictor for each clustering setting. Table IV summarizes the experimental parameters, where the default values are in bold. By default, the 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, assign 5 cores and 5GB RAM to the driver program, and set up 6 executors in each node in the Spark cluster. Each executor is assigned 5 cores and 16GB RAM.

TABLE III
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 IV
PARAMETERS

Parameters	Settings
# of Temporal Partitions α	50, **100**, 200, 500, 1000
# of 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 III)	10%, 20%, 30%, 40%, 50%, 60%, 70%, 90%, 100%
# of Optimization Iterations N_{BO}	1, 2, ..., **50**, ..., 60
# of Nodes in Spark Cluster	1, 3, **5**, 7

Metrics. In terms of the ST-kNN join, 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. Note that our goal is to reduce the execution time. The copy amplification and hit rate metrics can help us analyze the computation performance.

When assessing Execution Time Predictor, the following metrics are employed : 1) **Mean Absolute Error (MAE)**, which measures the average absolute difference between observed and predicted outcomes. 2) **Root Mean Square Error (RMSE)**, which measures the average difference between predicted values and the actual values. 3) **Mean Absolute Percentage Error (MAPE)**, which defines the accuracy of the forecasting model. It represents the average of the absolute percentage errors of each record in the training data.

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 [2] and LocationSpark [3], [25], 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** [2]. 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)** [3], [25]. We rewrite its source code to support ST-kNN join, as we did for Simba. Note 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.

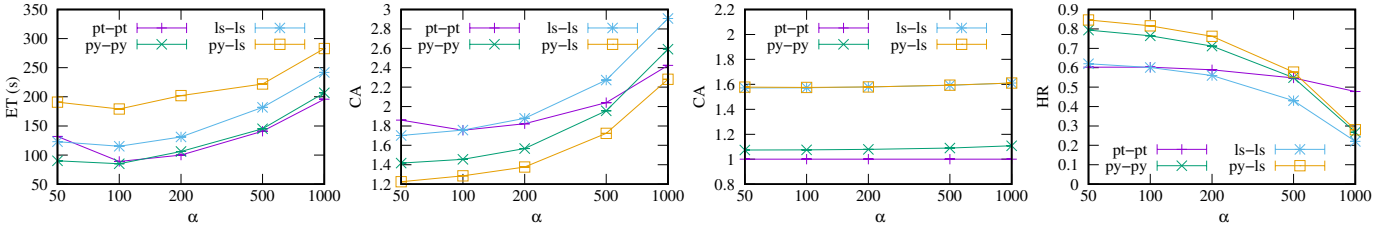
• **ST-kNNJ$_{nr}$**, which adopts Quad-tree for spatial partitioning just as ST-kNNJ does, but does not remove duplicates based on reference points before shuffling local join results.

In terms of execution time prediction models, we compare the adopted XGBoost with nine widely-used models, including Regression Tree (RT) [59], Random Forest (RF) [60], KNearestNeighbors (KNN) [61], Ridge [62], Lasso [63], AdaBoost (AB) [64], GradientBoost (GB) [65], Bagging [66] and LightGBM (LGBM) [67]. We do not compare Bayesian optimization with other knob tuning methods like Deep Learning (DL)-based methods [48], [50] and Reinforcement Learning (RL)-based methods [68]–[71]. This is because DL-based methods require a large number of training observations which are expensive to collect, and RL-based methods have been proved to require much higher tuning overhead than Bayesian optimization methods [47].

B. Effects of Parameters

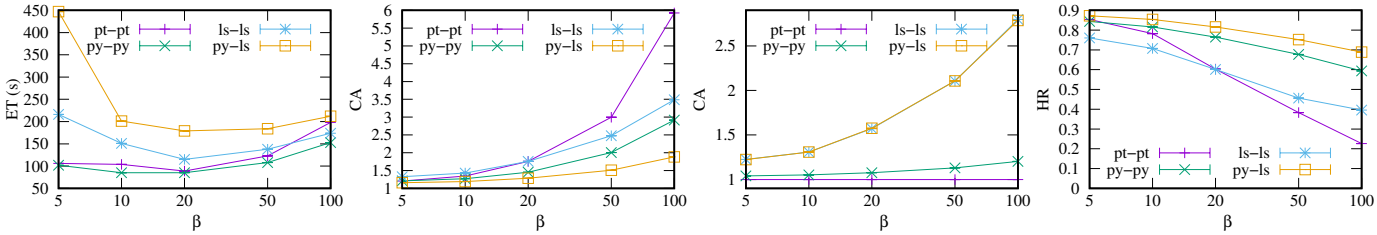
Different Values of α. Figure 11 presents the performance of ST-kNNJ with different values of α. As shown in Fig. 11(a), with an increasing α, the execution time of all dataset combinations first decreases, then increases. There are two reasons for an increasing execution time with a smaller α when $\alpha < 100$. 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.

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



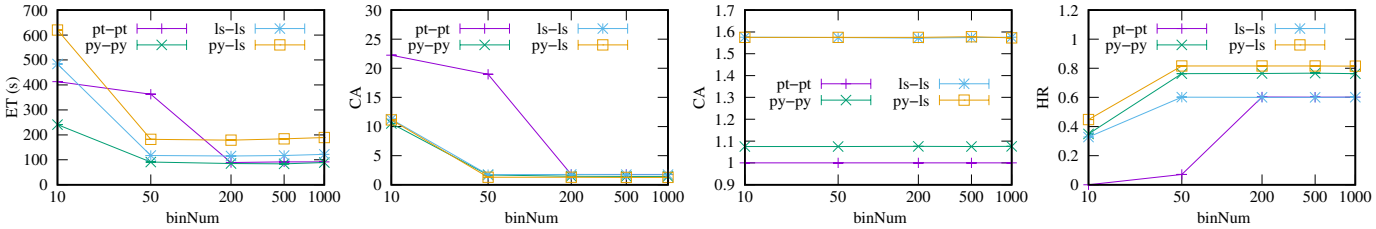
(a) Execution Time. (b) Copy Amplification of R. (c) Copy Amplification of S. (d) Hit Rate.

Fig. 11. Performance w.r.t. α



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

Fig. 12. Performance w.r.t. β



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

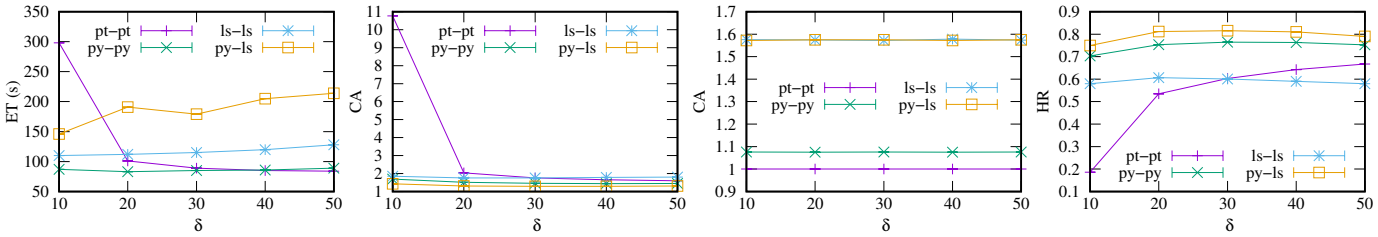
Fig. 13. Performance w.r.t. $binNum$

gets larger with an increasing α, as shown in Fig. 11(b) and Fig. 11(c); 2) a bigger α results in a lower hit rate, as shown in Fig. 11(d). That is, more objects $r \in R$ cannot find their ST-kNNs in the first local join, therefore they need to participate in the second join.

It is also interesting to see that in Fig. 11(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 $\ltimes$ 8.7m vs 3.9m $\ltimes$ 3.9m vs 4.6m $\ltimes$ 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. 11(c). Another interesting observation is that the copy amplification of R for `pt-pt` gets larger comparing $\alpha = 50$ to $\alpha = 100$ in Fig. 11(b), resulting in a fierce increasing of execution time in Fig. 11(a) and a slight drop off of hit rate in Fig. 11(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 that more $r \in R$ cannot find their ST-kNN in the first round local join.

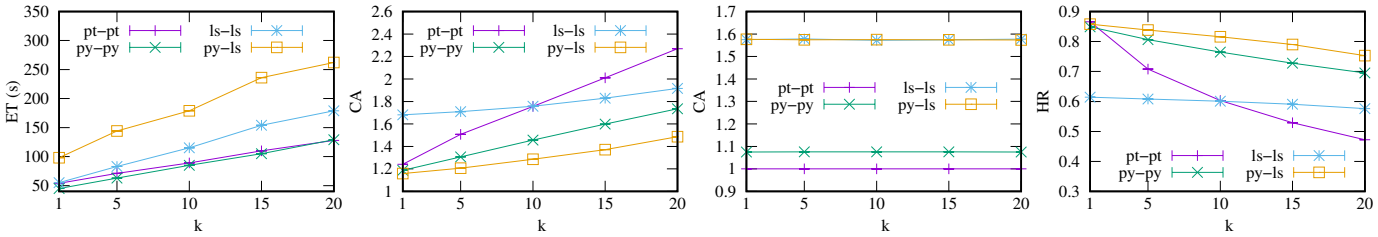
Different Values of β. Figure 12 demonstrates the performance of ST-kNNJ with different values of β. As shown in Fig. 12(a), when β gets larger from 5 to 100, the execution time first drops, then increases slightly. When $\beta = 20$, the performance achieves the best. We also observe that with an increasing β, the copy amplifications of both R and S get larger. This is because with a bigger β, the area of 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. 12(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. 12(d)) and a larger copy amplification of R (shown in Fig. 12(b)).

Different Values of $binNum$. As depicted in Fig. 13(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. 13(b). A more precise lower bound improves the hit rate as well, as shown in Fig. 13(d). We can observe that the



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

Fig. 14. Performance w.r.t. δ



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

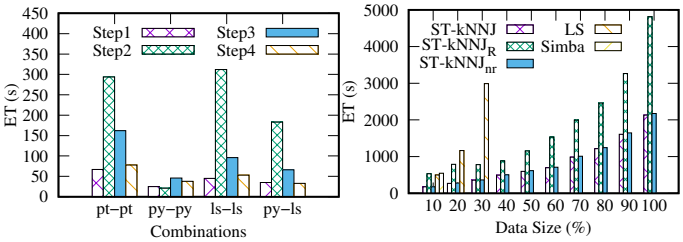
Fig. 15. Performance w.r.t. k

copy amplification of S has nothing to do with *binNum*, as shown in Fig. 13(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. 13(a) and Fig. 13(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 14 shows the impact of δ on ST-kNNJ performance. As shown in Fig. 14(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. 14(b)), and their hit rate gets higher slightly (shown in Fig. 14(d)). However, for `pt-pt` combination, the execution time first drops significantly, and then keeps stable. This is because for NYTrip dataset, the time span of objects is 0. If δ is too small (e.g., $\delta = 10$), its hit rate is very low (see Fig. 14(d)), causing a huge copy amplification of R (see Fig. 14(b)). Again, we can see from Fig. 14(c) that the copy amplification of S has little to do with δ.

Different Values of k. We can observe from Fig. 15(a) that with a bigger k, the execution time for all combinations get larger linearly, because their hit rate decreases linearly (shown in Fig. 15(d)), making their copy amplification of R increase linearly (shown in Fig. 15(b)). Figure 15 demonstrates that the copy amplification of S is not affected by k.

Execution Time of Different Steps. Figure 16(a) shows the execution time for different steps. We can find that the



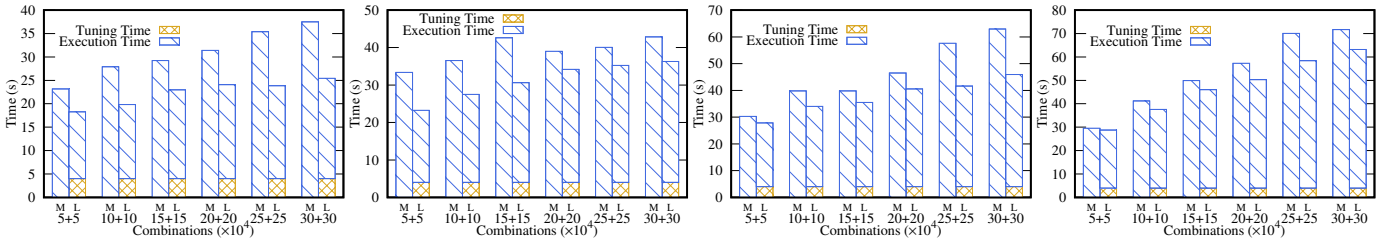
(a) Execution Time. (b) Execution Time.

Fig. 16. 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)

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 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.

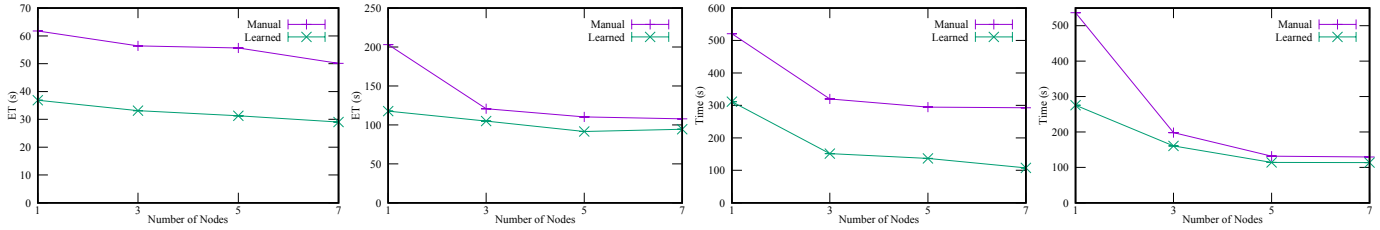
C. Comparing with Baselines

Figure 16(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. We check its source code,



(a) pt-pt. (b) ls-ls. (c) py-py. (d) py-ls.

Fig. 17. Effects of Knob Tuning on Varying Data Sizes.



(a) pt-pt. (b) ls-ls. (c) py-py. (d) py-ls.

Fig. 18. Effects of Knob Tuning on Varying Numbers of Nodes (For the learned settings, the time includes both tuning time and execution time; Data size: $3,000,000 + 3,000,000$).

and find that its proposed optimizer does not take effect for ST-kNN join. Location-Spark also fails when the data size reaches 40%, but our proposed ST-kNNJ methods can easily support much bigger data (even 100% data size), which verifies the scalability of our methods. 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, so 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 marginal, because the local join result transmission overhead is relatively much smaller than the overall computation overhead.

D. Knob Tuning Verification

Evaluation of Predictors. Table V presents the average performance of ten commonly-used machine learning models adopted by Execution Time Predictors, from which we can see that, XGBoost exhibits the overall best performance across all the selected models in terms of three metrics. Therefore, it is more suitable for our prediction task. Moreover, its ability to parallelize tree construction using features and data points allows it to handle larger datasets with faster computational efficiency, making it well-suited for handling the growing records in the Observation Set.

Effects of Varying Data Sizes. Figure 17 compares the online processing time between manual settings (denoted as "**M**") and learned settings (denoted as "**L**") across varying data sizes. For manual settings, we adopt the parameters that perform the best in Figures 11-13 (i.e., $\alpha = 100$, $\beta = 20$ and $binNum = 200$). Note that for manual settings, their tuning time is considered to be 0. We have the following observations. (1) For all combinations and data sizes, the online processing time with learned settings is always less than that with manual settings, even if we consider the tuning time for the learning-based method, which proves the effectiveness of the proposed knob tuning framework. (2) The tuning time of all combinations and data sizes is stable (about 4 seconds). This is because we always perform the same iterations (i.e., 50) to search the optimized parameter settings. It also verifies that the feature extraction overhead can be ignored, since larger datasets have the same tuning time. (3) Compared with the execution time, the tuning time can be negligible, as we replace the time-consuming ST-kNN join operation with an efficient model during the Bayesian optimization. (4) With larger datasets, the difference between the manual settings and learned settings becomes more significant. This is because the execution time often increases with the growth of the data size. When the data size becomes sufficiently large, even small variations in knob settings can have a significant impact on the execution time. Therefore, the advantages of parameters tuning become more pronounced in such scenarios.

TABLE V
PERFORMANCE OF PREDICTORS

Models	RT	RF	KNN	Ridge	Lasso	AB	GB	Bagging	LGBM	XGB
MAE	11.43	8.35	14.61	12.72	12.01	15.86	9.13	9.42	8.95	**8.34**
RMSE	27.64	23.32	28.07	26.26	27.61	22.74	24.72	21.27	20.48	**20.36**
MAPE	0.16	0.17	0.23	0.22	0.22	0.27	0.16	**0.13**	0.15	**0.13**

Effects of Varying # of Nodes. To test the scalability of the proposed methods on different numbers of nodes in the cluster, we investigate the performance of the learned settings and manual settings on a cluster of 1, 3, 5, 7 nodes, respectively.

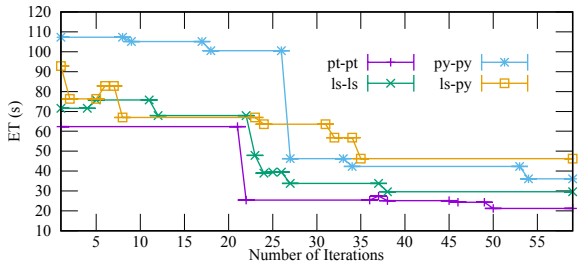


Fig. 19. Effects of the Number of Iterations (Data Size: 200,000 + 200,000).

As shown in Fig. 18, for all combinations of geometries, with an increasing number of cluster nodes, the time of both manual settings and learned settings first drops and then keeps stable. Since with more cluster nodes, there are more computer resources for ST-kNNJ, accelerating the executing process. However, when the number of nodes changes from 5 to 7, the improvement of both methods becomes marginal, as the system bottleneck turns into the network transmission among different nodes. We can also observe that for all numbers of nodes, the learned settings are better than manual settings, which verifies the effectiveness of knob tuning. It is interesting to see in Fig. 18(a) that the time of learned settings for point data does not fluctuate significantly with different data nodes. The reason could be that for point data, our learning models can find the optimal settings more easily. Even in a single node, the execution time can be very low.

Effects of Iterations. Figure 19 shows the effects of the number N_{BO} of iterations. For Bayesian Optimizer, the number of iterations can directly affect its results. When the number of iterations is sufficiently large, Bayesian Optimizer can conduct a very detailed exploration in the parameter space, identifying the most likely parameter combinations. However, conducting an excessive number of search iterations is time-consuming and may not always yield significant improvements. As shown in Fig. 19, Bayesian Optimizer can identify the optimal parameters within 50 iterations. Beyond this point, further iterations often do not lead to substantial improvements.

VIII. System Deployment

We integrate the ST-kNN join method into JUST [26], a distributed spatio-temporal data engine. As shown in Fig. 20, the spatio-temporal data with various geometry types is stored in HBase and indexed using Z2T index [26] (for point data) or XZ2T index [27] (for non-point data). JUST system would pre-compute the metadata (e.g., spatio-temporal distributions) for each dataset and store the metadata in a MySQL database. When answering an ST-kNN Join request, JUST would load the necessary data, and leverage the knob tuning models to determine the introduced parameters. JUST also implements a SQL engine, such that the ST-kNN join can be performed 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)
```

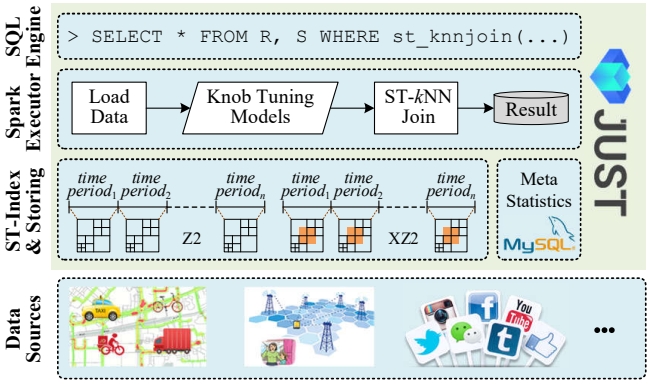


Fig. 20. Implementation in JUST System [26].

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.

Fig. 21. User Interface of ST-kNN Join in JUST.

Figure 21 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. Users can also upload their own datasets. In JustQL Panel, we input a SQL-like statement, and click the first button to run ST-kNN join. Here, we perform an ST-kNN join on two point tables *point*01 and *point*02, where $k = 2$ and $\delta = 100$s. The join result is shown in Result Panel.

IX. Related Works

A. Spatial-Related Join

To the best of our knowledge, none of the existing works are designed specially for ST-kNN join. We classify spatial-related join into three categories: 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 [18], [19] 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 [20] 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 [72], SpatialHadoop [73], LocationSpark [3], [25], SpatialSpark [74], GeoSpark [75], Stark [76] and Simba [2]. 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 [38], [41] and perform a spatial join using the standalone method. Finally, they merge local spatial join results into a global one.

***k*NN Join.** Compared with 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 is one-round join method [2], [4]–[6]. They first partition R, and then copy S to the target partitions based on the pivots of voronoi diagram [4], the partition center points [2], or space filling curves [5], [6], so 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 is two-round join method [3], [25], which partitions S first, and 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 works [35], [36], [77] consider both spatial and temporal information, 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, and then temporally partitions the dataset with a temporal interval. Stark [76] 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. The work [77] proposes a block-join method for spatio-temporal joins. It first partitions the entire spatio-temporal data space into equal-sized blocks, and then evaluates block trajectories in the same and adjacent blocks to get pairs satisfying the spatio-temporal join conditions. However, this work does not support ST-kNN join. Besides, it may face scalability problem since it is a standalone implementation.

B. Knob Tuning

Knob tuning [47] is a technology of intelligent database. It has become one of the hot directions with the development of AI4DB [78] in recent years. Methods for knob tuning include heuristic methods, deep learning (DL)-based methods, reinforcement learning (RL)-based methods, and Bayesian optimization (BO)-based methods. Heuristic methods based on rules [79] usually achieve sub-optimal settings, while heuristic methods based on searches [80] cannot utilize historical tuning data and prior knowledge. DL-based methods [48], [50] require a large number of prepared training samples to well train the models. RL-based methods [68]–[71] are suitable for high-dimensional configuration space exploration, but they require high overhead. BO-based methods [48], [49] can efficiently find high-quality knob settings with the exploration-exploitation strategy. It first establishes a probability distribution model according to the tuning history. Then, the model provides the optional hyper-parameters. Finally, the new observations with the optional hyper-parameters are added [81]. The three steps repeat until the maximum number of iterations is reached. With the progress of AI4DB, various advanced models based on Bayesian optimization have been proposed [49], [82]. In our ST-kNN join problem, there are only three parameters to be predicted. Besides, it requires much time to collect many training samples since it is relatively time-consuming to execute ST-kNN join. To this end, we adopt the BO-based method.

X. 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. We further propose a knob tuning framework based on Bayesian optimization to determine the values of introduced system parameters. The extensive experimental results based on three big real datasets show that our method is much more scalable and achieves 9X faster than baselines. Moreover, our knob tuning framework can always find an appropriate setting for each ST-kNN join request.

Acknowledgments

We would like to thank Junwen Liu, Zisheng Yu, Tianfu He, Sijie Ruan, Fuqiang Gu and Liang Hong for their great efforts in the prior work [1].

References

[1] R. Li, R. Wang, J. Liu, Z. Yu, T. He, S. Ruan, J. Bao, C. Chen, F. Gu, L. Hong, and Y. Zheng, "Distributed spatio-temporal k nearest neighbors join," in *ACM SIGSPATIAL*. ACM, 2021.

[2] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *ACM SIGMOD*, 2016, pp. 1071–1085.

[3] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *PVLDB*, vol. 9, no. 13, pp. 1565–1568, 2016.

[4] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *arXiv preprint arXiv:1207.0141*, 2012.

[5] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *EDBT*, 2012, pp. 38–49.

[6] Y. Liu, N. Jing, L. Chen, and W. Xiong, "Algorithm for processing k-nearest join based on r-tree in mapreduce," *Journal of Software*, vol. 24, no. 8, pp. 1836–1851, 2013.

[7] H. He, R. Li, R. Wang, J. Bao, Y. Zheng, and T. Li, "Efficient suspected infected crowds detection based on spatio-temporal trajectories," *arXiv preprint arXiv:2004.06653*, 2020.

[8] C. Liu, J. Sun, H. Jin, M. Ai, Q. Li, C. Zhang, K. Sheng, G. Wu, X. Qie, and X. Wang, "Spatio-temporal hierarchical adaptive dispatching for ridesharing systems," in *ACM SIGSPATIAL*, 2020, pp. 227–238.

[9] M. Haliem, G. Mani, V. Aggarwal, and B. Bhargava, "A distributed model-free ride-sharing approach for joint matching, pricing, and dispatching using deep reinforcement learning," *TITS*, vol. 22, no. 12, pp. 7931–7942, 2021.

[10] D. Li, Q. Yang, D. An, and Y. Zhang, "A location privacy aware taxi-hailing system: Adaptive differential privacy-based dynamic incentive method," *IEEE Internet of Things Journal*, 2023.

[11] J. Zhao, C. Chen, W. Zhang, R. Li, F. Gu, S. Guo, J. Luo, and Y. Zheng, "Coupling makes better: An intertwined neural network for taxi and ridesourcing demand co-prediction," *TITS*, 2023.

[12] L. Chen, Y. Gao, Z. Fang, X. Miao, C. S. Jensen, and C. Guo, "Real-time distributed co-movement pattern detection on streaming trajectories," *PVLDB*, vol. 12, no. 10, pp. 1208–1220, 2019.

[13] Z. Fang, Y. Gao, L. Pan, L. Chen, X. Miao, and C. S. Jensen, "Coming: A real-time co-movement mining system for streaming trajectories," in *SIGMOD*, 2020, pp. 2777–2780.

[14] D. Zhang, T. Ma, J. Hu, Y. Bei, K.-L. Tan, and G. Chen, "Co-movement pattern mining from videos," *PVLDB*, vol. 17, 2024.

[15] R. Baral, S. Iyengar, T. Li, and X. Zhu, "Hicaps: Hierarchical contextual poi sequence recommender," in *ACM SIGSPATIAL*, 2018, pp. 436–439.

[16] F. Zhou, P. Wang, X. Xu, W. Tai, and G. Trajcevski, "Contrastive trajectory learning for tour recommendation," *TIST*, vol. 13, no. 1, pp. 1–25, 2021.

[17] S. M. M. Rashid, M. E. Ali, and M. A. Cheema, "Deepalttrip: Top-k alternative itineraries for trip recommendation," *TKDE*, 2023.

[18] O. Gunther, "Efficient computation of spatial joins," in *ICDE*. IEEE, 1993, pp. 50–59.

[19] J. A. Orenstein, "Strategies for optimizing the use of redundancy in spatial databases," in *Symposium on Large Spatial Databases*. Springer, 1989, pp. 115–134.

[20] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM TODS*, vol. 32, no. 1, pp. 7–es, 2007.

[21] Y. Hu, S. Ruan, Y. Ni, H. He, J. Bao, R. Li, and Y. Zheng, "Salon: A universal stay point-based location analysis platform," in *ACM SIGSPATIAL*. ACM, 2021.

[22] R. Li, J. Bao, H. He, S. Ruan, T. He, L. Hong, Z. Jiang, and Y. Zheng, "Discovering real-time reachable area using trajectory connections," in *DASFAA*. Springer, 2020, pp. 36–53.

[23] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.

[25] M. Tang, Y. Yu, A. R. Mahmood, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: in-memory distributed spatial query processing and optimization," *Frontiers in Big Data*, vol. 3, p. 30, 2020.

[26] R. Li, H. He, R. Wang, Y. Huang, J. Liu, S. Ruan, T. He, J. Bao, and Y. Zheng, "Just: Jd urban spatio-temporal data engine," in *ICDE*. IEEE, 2020, pp. 1558–1569.

[27] R. Li, H. He, R. Wang, S. Ruan, T. He, J. Bao, J. Zhang, L. Hong, and Y. Zheng, "Trajmesa: A distributed nosql-based trajectory data management system," *TKDE*, 2021.

[28] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng, "Trajmesa: A distributed nosql storage engine for big trajectory data," in *ICDE*. IEEE, 2020, pp. 2002–2005.

[29] H. He, R. Li, J. Bao, T. Li, and Y. Zheng, "Just-traj: A distributed and holistic trajectory data management system," in *ACM SIGSPATIAL*. ACM, 2021.

[30] Y. Sui, R. Li, X. Wang, J. Liu, and J. Tang, "Just-studio: A platform for spatio-temporal data map designing and application building," in *WISE 2021*. Springer, 2021, pp. 536–546.

[31] H. He, R. Li, S. Ruan, T. He, J. Bao, T. Li, and Y. Zheng, "Trass: Efficient trajectory similarity search based on key-value data stores," in *ICDE*. IEEE, 2022, pp. 2306–2318.

[32] H. He, Z. Xu, R. Li, J. Bao, T. Li, and Y. Zheng, "Tman: A high-performance trajectory data management system based on key-value stores," in *ICDE*. IEEE, 2024.

[33] "St knn join," http://stknnjoin.urban-computing.com/, 2023.

[34] J. Dubé and D. Legros, "A spatio-temporal measure of spatial dependence: An example using real estate data," *Papers in Regional Science*, vol. 92, no. 1, pp. 19–30, 2013.

[35] R. T. Whitman, M. B. Park, B. G. Marsh, and E. G. Hoel, "Spatio-temporal join on apache spark," in *ACM SIGSPATIAL*, 2017, pp. 1–10.

[36] R. T. Whitman, B. G. Marsh, M. B. Park, and E. G. Hoel, "Distributed spatial and spatio-temporal join on apache spark," *ACM TSAS*, vol. 5, no. 1, pp. 1–28, 2019.

[37] F. P. Preparata and M. I. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[38] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[39] J. Bao, R. Li, X. Yi, and Y. Zheng, "Managing massive trajectories on the cloud," in *ACM SIGSPATIAL*, 2016, pp. 1–10.

[40] R. Li, S. Ruan, J. Bao, and Y. Zheng, "A cloud-based trajectory data management system," in *ACM SIGSPATIAL*, 2017, pp. 1–4.

[41] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *ACM SIGMOD*, 1984, pp. 47–57.

[42] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *ACM SIGMOD*, 1990, pp. 322–331.

[43] K. Yang, X. Ding, Y. Zhang, L. Chen, B. Zheng, and Y. Gao, "Distributed similarity queries in metric spaces," *DSE*, vol. 4, no. 2, pp. 93–108, 2019.

[44] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[45] Q. Zhu, J. Gong, and Y. Zhang, "An efficient 3d r-tree spatial index method for virtual geographic environments," *ISPRS*, vol. 62, no. 3, pp. 217–224, 2007.

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[47] X. Zhao, X. Zhou, and G. Li, "Automatic database knob tuning: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[48] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang, "ibtune: Individualized buffer tuning for large-scale cloud databases," *PVLDB*, vol. 12, no. 10, pp. 1221–1234, 2019.

[49] X. Zhang, H. Wu, Y. Li, J. Tan, F. Li, and B. Cui, "Towards dynamic and safe configuration tuning for cloud databases," in *SIGMOD*, 2022, pp. 631–645.

[50] D. Van Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Bilien, and A. Pavlo, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *PVLDB*, vol. 14, no. 7, pp. 1241–1253, 2021.

[51] S. Wu, Y. Li, H. Zhu, J. Zhao, and G. Chen, "Dynamic index construction with deep reinforcement learning," *Data Science and Engineering*, vol. 7, no. 2, pp. 87–101, 2022.

[52] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[53] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *ACM SIGKDD*, 2016, pp. 785–794.

[54] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[55] M. Seeger, "Gaussian processes for machine learning," *International journal of neural systems*, vol. 14, no. 02, pp. 69–106, 2004.

[56] B. Donovan and D. Work, "New york city taxi trip data (2010-2013)," 2016. [Online]. Available: https://doi.org/10.13012/J8PN93H8

[57] "Didi chuxing gaia initiative," 2021. [Online]. Available: https://gaia.didichuxing.com

[58] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-Y. Ma, "Mining user similarity based on location history," in *ACM SIGSPATIAL*, 2008, pp. 1–10.

[59] W.-Y. Loh, "Classification and regression trees," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[60] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[61] J. Goldberger, G. E. Hinton, S. Roweis, and R. R. Salakhutdinov, "Neighbourhood components analysis," *Advances in neural information processing systems*, vol. 17, 2004.

[62] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[63] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 58, no. 1, pp. 267–288, 1996.

[64] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

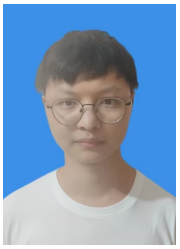
[65] J. H. Friedman, "Stochastic gradient boosting," *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[66] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, pp. 123–140, 1996.

[67] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

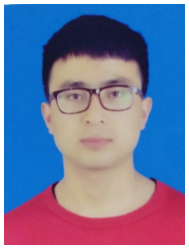
[68] M. Li, Y. Wang, S. Ma, C. Liu, D. Huo, Y. Wang, and Z. Xu, "Auto-tuning with reinforcement learning for permissioned blockchain systems," *PVLDB*, vol. 16, no. 5, pp. 1000–1012, 2023.
[69] L. Zhang, C. Chai, X. Zhou, and G. Li, "Learnedsqlgen: Constraint-aware sql generation using reinforcement learning," in *SIGMOD*, 2022, pp. 945–958.
[70] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li, "Adaptive code learning for spark configuration tuning," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 1995–2007.
[71] X. Zhang, Z. Chang, H. Wu, Y. Li, J. Chen, J. Tan, F. Li, and B. Cui, "A unified and efficient coordinating framework for autonomous dbms tuning," *SIGMOD*, vol. 1, no. 2, pp. 1–26, 2023.
[72] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce," in *PVLDB*, vol. 6, no. 11. NIH Public Access, 2013.
[73] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *ICDE*. IEEE, 2015, pp. 1352–1363.
[74] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *ICDE workshops*. IEEE, 2015, pp. 34–41.
[75] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *ACM SIGSPATIAL*, 2015, pp. 1–4.
[76] S. Hagedorn, P. Gotze, and K.-U. Sattler, "The stark framework for spatio-temporal data analytics on spark," *BTW*, 2017.
[77] T. Li and J. Xu, "Block-join: A partition-based method for processing spatio-temporal joins," in *APWeb-WAIM*. Springer, 2022, pp. 397–411.
[78] X. Zhou, C. Chai, G. Li, and J. Sun, "Database meets artificial intelligence: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 3, pp. 1096–1116, 2020.
[79] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, "Using probabilistic reasoning to automate software tuning," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 404–405, 2004.
[80] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 338–350.
[81] S. Greenhill, S. Rana, S. Gupta, P. Vellanki, and S. Venkatesh, "Bayesian optimization for adaptive experimental design: A review," *IEEE access*, vol. 8, pp. 13 937–13 948, 2020.
[82] X. Wang, Y. Jin, S. Schmitt, and M. Olhofer, "Recent advances in bayesian optimization," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–36, 2023.

Ruiyuan Li is an associate professor with Chongqing University, China. He is the director of Start Lab (Spatio-Temporal Art Lab). He received the B.E. and M.S. degrees from Wuhan University, China in 2013 and 2016, respectively, and the Ph.D. degree from Xidian University, China in 2020. He was the Head of Spatio-Temporal Data Group in JD Intelligent Cities Research, leading the research and development of JUST (JD Urban Spatio-Temporal data engine). Before joining JD, he had interned in Microsoft Research Asia from 2014 to 2017. His research focuses on Spatio-temporal Data Management and Urban Computing.



Jiajun Li received the B.S. degree from Wuhan University of Technology, China in 2023. He is currently pursuing the master's degree in the College of Computer Science, Chongqing University, China. He is now a member of Chongqing University Start Lab, under the supervision of Prof. Ruiyuan Li. His research interests include spatio-temporal data mining and AI4DB.

Minxin Zhou is a undergraduate student at the College of Computer Science, Chongqing University, China. Her research interest is spatio-temporal data mining. She is now a member of Chongqing University Start Lab, under the supervision of Prof. Ruiyuan Li.



Runbin Wang is a research assistant in JD Intelligent Cities Research and JD iCity. He received his B.E. degree and master degree from SWJTU University in 2018 and 2021, respectively. His research interests include urban computing, spatio-temporal data mining, and distributed systems.

Huajun He is a Ph.D. student at the School of Computer Science and Technology, SWJTU University, China. He received his B.E. degree from SWJTU University in 2018. His research interests include urban computing, database, spatio-temporal data mining, and distributed systems. He is now a research intern in JD Intelligent Cities Research and JD iCity, under the supervision of Prof. Yu Zheng and Dr. Jie Bao.

Chao Chen received the B.E. and M.S. degrees from Northwestern Polytechnical University, Xi'an, China, in 2007 and 2010, respectively, and the Ph.D. degree from Pierre and Marie Curie University and the Institut Mines-Télécom/Télécom SudParis, France, in 2014. In 2009, he was a research assistant with the Hong Kong Polytechnic University. He is currently a full professor with the College of Computer Science, Chongqing University, China. His research interests include pervasive computing, mobile computing, urban logistics, data mining from large-scale GPS trajectory data, and big data analytics for smart cities.

Jie Bao got his Ph.D degree in Computer Science from University of Minnesota at Twin Cities in 2014. He worked as a researcher in Urban Computing Group at MSR Asia from 2014 to 2017. He currently leads the Data Platform Division in JD iCity. His research interests include: Spatio-temporal Data Management/Mining, Urban Computing, and Location-based Services.

Yu Zheng is a Vice President and Chief Data Scientist at JD.COM, passionate about using big data and AI technology to tackle urban challenges. His research interests include big data analytics, spatio-temporal data mining, machine learning, and artificial intelligence. He also leads the JD iCity as the president and serves as the director of the JD Intelligent Cities Research. Before joining JD, he was a senior research manager at Microsoft Research. Zheng is also a Chair Professor at Shanghai Jiao Tong University, an Adjunct Professor at Hong Kong University of Science and Technology. He is a fellow of IEEE.