Joinable Search over Multi-source Spatial Datasets: Overlap, Coverage, and Efficiency

Wenzhe Yang[†], Sheng Wang[†*], Zhiyu Chen[‡], Yuan Sun[§], and Zhiyong Peng[†¶*]
[†]School of Computer Science, Wuhan University [‡]Amazon.com, Inc.
[§]La Trobe Business School, La Trobe University [¶]Big Data Institute,Wuhan University
[wenzheyang, swangcs, peng]@whu.edu.cn, zhiyuche@amazon.com, yuan.sun@latrobe.edu.au

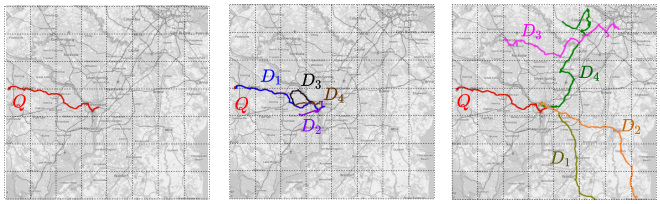
***Abstract*—The search for joinable data is pivotal for numerous applications, such as data integration, data augmentation, and data analysis. Although there have been many successful joinable search studies for table discovery, the study of finding joinable spatial datasets for a given query from multiple spatial data sources has not been well considered. This paper studies two cases of joinable search problems from multiple spatial data sources. In addition to the overlap joinable search problem (OJSP), we also propose a novel coverage joinable search problem (CJSP) that has not been considered before, motivated by many real-world applications in the field of spatial search. To support two cases of joinable search over multiple spatial data sources seamlessly, we propose a multi-source spatial dataset search framework. Firstly, we design a DIstributed Tree-based Spatial index structure called DITS, which is used not only to design acceleration strategies to speed up joinable searches, but also to support efficient communication between multiple data sources. Additionally, we prove that the CJSP is NP-hard and design a greedy approximate algorithm to solve the problem. We evaluate the efficiency of our search framework on five real-world data sources, and the experimental results show that our framework can significantly reduce running time and communication costs compared with baselines.**

***Index Terms*—Spatial dataset search, Overlap joinable search, Coverage joinable search, Indexing, NP-hard.**

I. Introduction

Spatial data is a critical component of real-world data [16], [24], [36], [58]. Efficient spatial queries have become increasingly important in spatial data studies [32], [49], [54], [64]. For example, the spatial join focuses on finding all matched pairs of records that satisfy some specified join predicate for two given spatial datasets [14], [42], [64]. Unfortunately, the traditional join operations require users to specify the datasets to be joined, posing challenges for junior users who are unfamiliar with the data source's data. Thus, the joinable table search [18], [21], [70], [73] has been proposed to identify the tables that can be joined with the query table, which is a key procedure in subsequent data analysis [20], [22]. However, the joinable search over spatial datasets has not been well-studied.

Especially nowadays, massive spatial data often come from different data sources and are located in different regions. Many open-source data platforms, such as OpenGeoMetadata [3], GeoBlacklight [1], OpenGeoHub [2], etc., integrate spatial data from multiple open data sources. Also, individual companies or organizations are independent in managing datasets and have a wealth of spatial data stored in their own data sources. It is crucial to break the barriers between different data sources to facilitate joinable search over different spatial data sources. Motivated by this, we focus on the joinable search problem over multiple spatial data sources. In what follows, we will use a municipal planning example to introduce two types of joinable searches.



(a) Query dataset (b) Overlap joinable search (c) Coverage joinable search

Fig. 1. An example of performing overlap and coverage joinable search on real multi-source transit datasets from Washington, D.C. and Maryland.

Example 1. *In municipal planning, planners need to find datasets for analyzing traffic patterns and building a sound transportation system. There are two main tasks: (1) as shown in Fig. 1(a), a user inputs a query dataset Q consisting of a set of spatial points in Washington, D.C. The user specifies the longitude and latitude as the join columns and wants to find 4 datasets that have the maximum spatial overlap with Q. The results are shown in Fig. 1(b). On this basis, we can utilize the found joinable datasets and Q for subsequent data analysis, such as trajectory near-duplicate detection [56] and traffic congestion prediction [71], [72]. We define this type of search as an **overlap joinable search**; (2) the user also wants to find 4 datasets that have maximum spatial coverage after connecting to with Q, which can help construct transfer routes and meet more travel needs [6], [28]. It is inappropriate for users to transfer for too long when switching transportation. Thus, a natural constraint is to ensure connectivity between different routes. As shown in Fig. 1(c), the user can find 4 datasets of routes in Maryland that are connected to the query and cover larger regions. We define this type of search operation as a **coverage joinable search**.*

What is Overlap Joinable Search? The overlap joinable search is similar to the overlap set similarity search problem [70], [73], [74], whose goal is to find k sets that have the maximum intersection size with the query set [11], [42],

* Sheng Wang and Zhiyong Peng are the corresponding authors.
‡ Work does not relate to the position at Amazon.

[66]. Unfortunately, the spatial dataset is a set of points with longitude and latitude. The mere comparison of the numerical values of two coordinates is insufficient in determining whether the two points overlap or match, as geospatial data typically exhibits a certain degree of precision error in the decimal places.

To facilitate the calculation of the intersection number of the spatial datasets, we partition the space into uniform cells (as shown in the dashed cells of Fig. 1(a)). The size of the set intersection between two spatial datasets is the number of overlapping cells. Fig. 1(b) shows 4 routes in Washington, D.C. with the maximum overlapping with Q. We can see that the query Q and the route D_1 are highly overlapping.

What is Coverage Joinable Search? The goal of coverage joinable search is to find k datasets that have the maximum coverage area with the query dataset. When performing the coverage joinable search, as points have no physical dimensions (length and width), it is difficult to measure the coverage area of the spatial dataset. Thus, we can define the number of cells containing these spatial points as the coverage of the spatial dataset. Additionally, spatial connectivity [13], [52], [53] is a key constraint in the coverage joinable search, ensuring the relevance between the results with the query. Motivated by the above example, a general definition of spatial connectivity is necessary.

The spatial connectivity we define requires that any result dataset be directly or indirectly connected with the query, where a dataset is directly connected to a query if two spatial datasets satisfy the given criteria (e.g., there exists at least one overlapped cell), and a dataset is indirectly connected to a query if one or more intermediate datasets are directly connected to the query as a medium. For example, in Fig. 1(c), for the query Q in Washington, D.C., we can find 4 routes in Maryland with the maximum coverage and satisfy the spatial connectivity, where Q and D_4 are directly connected since they have an overlapped cell, while Q and D_3 are indirectly connected since they can establish a connection through the intermediary datasets D_4.

Two Problems to Solve. In this paper, we take the first attempt to define two types of joinable search problems over spatial datasets to enrich a given query dataset in both depth (overlapping) and width (coverage) directions. The former one can be formulated as the *overlap joinable search problem* (**OJSP**), and the latter one as the *coverage joinable search problem* (**CJSP**). In addition, considering that the data is widely distributed on multiple data sources, it is not enough to only study the local joinable search. Thus, we study the multi-source dataset search that supports overlap and coverage joinable search operations.

Challenges. To solve the above problems, we face multiple key challenges: (1) a naïve way to support overlap and coverage joinable searches is to build individual indexes for each type of joinable search. However, this can be costly in terms of both storage space and index construction time. Hence, it is preferable to have an efficient index that supports both types of joinable searches; (2) data is often stored in multiple independent and autonomous data sources. Breaking the barriers between different spatial data sources to facilitate the joinable search is also a core challenge; (3) the scale of datasets in each data source is very large, so simply designing the index is not enough. Moreover, we also prove that **CJSP** is NP-hard. Thus, it is urgent to design efficient pruning strategies combined with the index to accelerate the joinable search process over multiple data sources.

Contributions. To address the challenges above, we mainly make the following contributions:

- To support two types of joinable searches over multiple spatial data sources, we first formulate them into two problems called **OJSP** and **CJSP**, and prove the NP-hardness of **CJSP** (see Section III). Then, we develop an efficient search framework to support **OJSP** and **CJSP** (see Section IV).
- We design a **DI**stributed **T**ree-based **S**patial index structure DITS, composed of local indices and a global index, which not only accelerates **OJSP** and **CJSP** locally, but also supports efficient communication between multiple data sources (see Section V).
- Based on DITS, we first design query distribution strategies to reduce communication costs. Moreover, we propose effective lower and upper bounds for **OJSP** and **CJSP**, and design an efficient exact algorithm and a greedy approximation algorithm to solve them, respectively (see Section VI).
- We conduct extensive experiments on five real-world spatial data sources to evaluate the index construction, search performance, and communication cost of the multi-source joinable search framework. The experimental results show that our proposed framework is highly effective and efficient in solving **OJSP** and **CJSP** compared with the baselines (see Section VII).

II. Related Work

There are three kinds of work closely related to the joinable search over spatial datasets, including spatial join operation [8], [29], [31], [37], [40], overlap set similarity search [39], [59], [73], [74], and maximum coverage problem (MCP) [17], [30], [47], [48], wherein our coverage joinable search can be viewed as a variant of MCP. In what follows, we will conduct a literature review in these three parts.

Spatial Join. The spatial join operation is similar to the join operation in relational databases [8], [29], [41], [42], [75], which is defined on two given sets of spatial objects (e.g., point, line, polygon), and finds all matched pairs of objects by their spatial relationships (most often intersection). For example, for two given spatial point sets, the spatial intersection join [8], [9] aims to find all pairs of intersection points between the two datasets. In contrast, spatial joinable search identifies k datasets from the data source that can be joined with the query dataset.

As the amount of data increased, the spatial join processing for distributed systems has attracted much attention [46]. For

example, in the Hadoop-GIS system [4], spatial joins are computed by dividing the space into a grid and the objects in each cell are stored locally at nodes in the HDFS. In the SpatialHadoop system [23], a global index is stored at a master node, then the master node partitions all data into chunks and distributes the data partition to slave nodes. Moreover, implementations of the spatial join using Spark [57], [60], [61], where the data is shared in the memories of all nodes. However, the existing frameworks only support the traditional spatial join operation, which cannot be used to accelerate and solve the overlap and coverage joinable search proposed in this paper simultaneously.

Overlap Set Similarity Search. The goal of overlapping set similarity search is to find k sets that have the maximum set intersection size with the query [44], [55], [70], [73], [74]. Currently, the existing set similarity search studies are mainly divided into exact search [7], [39], [55], [73] and approximate search [22], [25], [74].

For example, fuzzy join [12], [15], [19], [50], [65] is a powerful operator used in set matching that allows records to match approximately [62]. An early solution is FastJoin [50], which generates the signature for each set and finds similar pairs of records. SilkMoth [19] optimizes this signature scheme, reducing the number of used tokens to generate fewer candidates. Zeakis et al. [65] proposed a token-based novel filtering technique to improve efficiency. For spatial datasets, the coordinates can form a multi-column composite key theoretically [43], [68] to perform the approximate matching. Nevertheless, the existing fuzzy join studies [12], [15], [19], [50], [65] mainly focus on finding approximate matching pairs in two given sets. There is still a lack of efficient indexes to accelerate the computation of fuzzy matching between a query and plenty of candidate datasets.

In this paper, we focus on the exact solution. For example, Peng et al. [39] transformed time series datasets into a set of cells and measured the similarity based on the Jaccard index. However, it requires scanning all datasets and estimating the number of set intersections, where pairwise comparisons are time-consuming. Zhu et al. [73] proposed an exact algorithm called Josie to accelerate the set intersection computation. However, Josie inherits the limitation of the prefix filter, whose performance highly depends on the data distribution and yields a worst-case time complexity. In addition, unlike table data, spatial datasets are widely distributed in space. Thus, we need to design an effective index and search algorithm for the characteristics of spatial datasets.

Maximum Coverage Problem. The maximum coverage problem (MCP) is a classical problem in combinatorics [30]. Given a universe of elements $\mathcal{U} = \{e_1, e_2, \ldots, e_n\}$ and a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, where each S_i is a subset of $\mathcal{U}$, the objective of MCP is to select a fixed number (k) of sets, such that the total number of elements covered by the selected sets is maximized. Over the past two decades, MCP and its variants have been extensively studied, such as the budgeted MCP [33], the weighted MCP [48], the connected MCP [47], and the generalized MCP [17]. These variants share the common objective of selecting sets to maximize the total weight of elements in the union set.

However, it is worth noting that our **CJSP** has a unique challenge: finding the datasets that are directly or indirectly connected to the query and maximizing the union of elements contained by the found datasets and query. The existing solutions are designed for different constraints and cannot be used directly to solve our problem. Secondly, among the existing solutions, greedy algorithms are one of the most efficient ways. However, they need to traverse all candidate datasets and perform the exact computation, which is very time-consuming. Therefore, efficient indexes are urgently needed to speed up the search process.

III. Definitions

In this section, we introduce the data modelling and formalize problems of overlap and coverage joinable search. Frequently used notations are summarized in Appendix IX-A.

A. *Data Modelling*

Definition 1. ***(Spatial Point)*** *A spatial point $p = (x, y)$ is a 2-dimension vector, which has a longitude x and a latitude y (e.g., p=(116.36422°, 39.88781°)).*

Definition 2. ***(Spatial Dataset)*** *A spatial dataset D contains a set of 2-dimension points, i.e., $D = \{p_1, p_2, \cdots, p_{|D|}\}$, where $|D|$ is the size of the dataset D.*

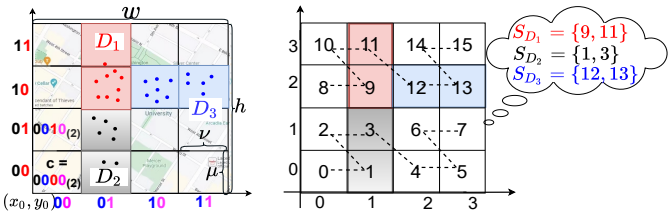
Definition 3. ***(Spatial Data Source)*** *A spatial data source $\mathcal{D}$ contains a set of datasets, i.e., $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, where n denotes the number of datasets in the data source.*

As we introduced in Section I, for ease of quantifying the degree of overlap between the two datasets and the coverage of the datasets, we used a grid partitioning method to represent the spatial dataset, which is widely used in large spatial data processing [4], [23], [35], [69].

Definition 4. ***(Grid and Cell)*** *A 2-dimensional space can be divided into a grid $\mathcal{C}_\theta$ consisting of $2^\theta \times 2^\theta$ cells, where θ called resolution. Each cell c denotes a unit space in the grid $\mathcal{C}_\theta$, and its coordinates (X, Y) can be converted into a non-negative integer to uniquely indicate c using the z-order curve space-filling method [39], [45], [58], denoted as $z(X, Y) = c$. The coordinates are transformed into cell (integer) IDs which are consecutive and form the range $[0, 2^\theta \times 2^\theta - 1]$.*

Based on the grid partition, each spatial dataset can be represented as a finite set consisting of a set of cell IDs.

Definition 5. ***(Cell-based Dataset)*** *Given a 2-dimensional space and a grid $\mathcal{C}_\theta$, a cell-based dataset $S_{D,\mathcal{C}_\theta}$ of the dataset D is a set of cell IDs where each cell ID denotes that there exists at least one point $(x, y) \in D$ falling into the cell c of the grid $\mathcal{C}_\theta$. The coordinates (X, Y) of cell c are $(\frac{x-x_0}{\nu}, \frac{y-y_0}{\mu})$, where (x_0, y_0) represents the coordinate of the bottom-left point in the whole 2-dimensional space, and ν and μ denote the width and height of the cell c. Thus, the cell-based dataset*



(a) Grid Partition (b) Cell-based Dataset

Fig. 2. Illustration of the grid partition and cell-based dataset.

is represented as $S_{D,\mathcal{C}_\theta} = \{z(\frac{x-x_0}{\nu}, \frac{y-y_0}{\mu}) | (x, y) \in D\}$. *For ease of presentation, we omit the subscript* $\mathcal{C}_\theta$ *of* S *when the context is clear.*

Example 2. *Figs. 2(a) and (b) show the process of generating the cell-based dataset by using the grid partition method. As shown in Fig. 2(a), the size of data space containing all datasets is* $h \times w$. *When performing the grid partitioning, we divide the whole space into a* $2^\theta \times 2^\theta$ *grid* ($\theta = 2$). *Each cell in the grid can be represented by an integer by interleaving the binary representations of its coordinate values, e.g., for the bottom left cell, its ID is* 0, *which is transformed from its coordinates (0, 0). Thus, each spatial dataset* D *can be represented as a finite set* S_D *consisting of a sequence of cell IDs. As shown in Fig. 2(b), three cell-based datasets of* D_1, D_2 *and* D_3 *is* $S_{D_1} = \{9, 11\}$, $S_{D_2} = \{1, 3\}$ *and* $S_{D_3} = \{12, 13\}$. *The spatial coverage of* S_D *is the number of IDs in the set.*

Spatial connectivity is a crucial factor in finding relevant datasets, which has also been considered before in trajectory studies [10], [52], [53], such as the coincidence of the endpoints of trajectories [53]. Similarly, to find a set of datasets that is relevant to the query, we first define a general distance measure to quantify the relationship between two spatial datasets, and then establish criteria defining the connectivity relationships among the datasets.

Definition 6. ***(Cell-based Dataset Distance)*** *Given two cell-based datasets* S_D *and* $S_{D'}$, *the distance between* S_D *and* $S_{D'}$ *is defined as the distance between two nearest cells, i.e.,*

$$dist(S_D, S_{D'}) = \min_{c_i \in S_D, c_j \in S_{D'}} \{||c_i, c_j||_2\}. \quad (1)$$

where c_i *and* c_j *denote cell IDs, which can be decomposed into coordinates of the cells in the grid,* $||\,||_2$ *is the Euclidean distance between the coordinates of two cells.*

Definition 7. ***(Directly Connected Relation)*** *Given a distance threshold* δ, *we say two cell-based datasets* S_D *and* $S_{D'}$ *are directly connected if* $dist(S_D, S_{D'}) \leq \delta$.

Definition 8. ***(Indirectly Connected Relation)*** *Two cell-based datasets* S_D *and* $S_{D'}$ *are indirectly connected if exists one or multiple ordered datasets* $\{S_{D_1}, \ldots, S_{D_i}\}(i \geq 1)$ *such that each pair of adjacent datasets (e.g.,* (S_D, S_{D_1}), $\ldots$, $(S_{D_i}, S_{D'})$*) are directly connected.*

Definition 9. ***(Spatial Connectivity)*** *Given a collection of cell-based datasets* $\mathcal{S}_\mathcal{D} = \{S_{D_1}, \ldots, S_{D_{|\mathcal{D}|}}\}$, *the collection* $\mathcal{S}_\mathcal{D}$ *satisfies the spatial connectivity if and only if any pair of datasets* S_{D_i} *and* S_{D_j} *in* $\mathcal{S}_\mathcal{D}(i \neq j)$ *are either directly or indirectly connected relation.*

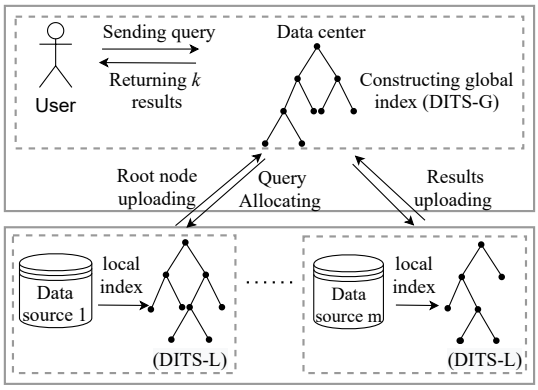


Fig. 3. Multi-source joinable spatial dataset search framework.

Example 3. *Following Example 2, we can compute the distance* $dist(S_{D_1}, S_{D_2}) = 1$, $dist(S_{D_1}, S_{D_3}) = 1$, *and* $dist(S_{D_2}, S_{D_3}) = \sqrt{2}$. *For a given connectivity threshold* $\delta = 1$, *the dataset* S_{D_1} *is directly connected to* S_{D_2} *and* S_{D_3}, *the dataset* S_{D_2} *is indirectly connected to* S_{D_3}. *Thus,* S_{D_1}, S_{D_2} *and* S_{D_3} *satisfy the spatial connectivity.*

B. Problem Definitions

We define two types of problems to find joinable datasets for the given query. The specific joinable search problems are as follows:

Definition 10. ***(Overlap Joinable Search Problem (OJSP))*** *Given a cell-based query dataset* S_Q, *a collection of cell-based datasets* $\mathcal{S}_\mathcal{D} = \{S_{D_1}, \ldots, S_{D_{|\mathcal{D}|}}\}$, *and a positive integer* k, ***OJSP*** *aims to find a subset* $\mathcal{S}^* \subseteq \mathcal{S}_\mathcal{D}$ *such that* $|\mathcal{S}^*| \leq k$ *and* $\forall S_{D_i} \in \mathcal{S}^*$ *and* $\forall S_{D_j} \in \mathcal{S}_\mathcal{D} \backslash \mathcal{S}^*$, *it always satisfies* $|S_Q \cap S_{D_i}| \geq |S_Q \cap S_{D_j}|$.

Definition 11. ***(Coverage Joinable Search Problem (CJSP))*** *Given a cell-based query dataset* S_Q, *a collection of cell-based datasets* $\mathcal{S}_\mathcal{D} = \{S_{D_1}, \ldots, S_{D_{|\mathcal{D}|}}\}$, *and a positive integer* k, ***CJSP*** *aims to find a subset* $\mathcal{S}^* \subseteq \mathcal{S}_\mathcal{D}$ *such that:*

$$\mathcal{S}^* = \arg\max_{\mathcal{S}^* \subseteq \mathcal{S}_\mathcal{D}} |S_Q \cup (\cup_{S_{D_i} \in \mathcal{S}^*} S_{D_i})|, \quad (2)$$
$$s.t. \; |\mathcal{S}^*| \leq k,$$
$$\mathcal{S}^* \cup \{S_Q\} \text{ satisfy spatial connectivity.}$$

C. NP-Hardness of CJSP

Lemma 1. ***CJSP*** *is NP-hard.*

We prove that **CJSP** is NP-hard in Appendix IX-B by performing a reduction from the NP-hard maximum coverage problem, owing to page limitations.

IV. Overview of Our Search Framework

In this paper, we design a joinable search framework over multiple spatial data sources, which not only effectively accelerates overlap and coverage joinable search simultaneously, but also provides efficient search over multiple independent data sources. As shown in Fig. 3, the search framework contains a data center and multiple independent data sources. Each data source organizes its data and constructs the local index. The data center receives the distribution information from data sources and maintains a global index. When the

data center receives a query request, it first performs a global search and sends the query to the candidate data sources for local search. Then, each candidate data source sends the results back to the data center after completing the local search, and the data center finally performs the aggregation and returns the k datasets to the user. The following provides a high-level description of index construction and search acceleration.

Index Construction. We propose a **DI**stributed **T**ree-based **S**patial index structure (DITS), to support joinable search over multiple data sources, where the global index, denoted as DITS-G, constructed by the data center is used to find relevant data sources that contain possible results, and the local index, denoted as DITS-L, is used to find results in each data source. Specifically, instead of indexing each dataset, our global index DITS-G is created by recording the distribution information of data sources, which is mainly used to filter out the data sources irrelevant to the query. Moreover, we propose a novel local index DITS-L, which effectively combines the features of both the balltree and the inverted index. Compared with the inverted and tree indexes, our DITS-L can prune those irrelevant spatial datasets and quickly retrieve the results of two types of joinable search (see Section V).

Accelerating Joinable Spatial Dataset Search. We propose effective query distribution strategies and search algorithms to support two types of joinable searches. Firstly, to finish the multi-source joinable search, the process involves communication between the data source and the data center. Thus, we design two query distribution strategies to reduce communication costs by reducing the frequency of communication and the number of bytes transferred. Moreover, for the overlap joinable search, we design an efficient filter-verification algorithm to solve **OJSP**, where the filter step combines lower and upper bounds and branch-and-bound strategies to prune plenty of dissimilar pairs and get a set of candidates, and the verification step utilizes effective techniques to verify the candidates. Furthermore, for the coverage joinable search, we propose a heuristic greedy algorithm to solve **CJSP** (see Section VI).

V. Index Construction

In this section, we focus on constructing an efficient index to support two types of joinable searches over multiple spatial data sources. In our scenario, each data source is independent and autonomous. Thus, we propose a novel distributed tree-based spatial index structure (DITS) for spatial joinable search, which is composed of one centralized global index and distributed local indices–one per data source. In what follows, we will introduce the detailed construction process of local and global indexes in Sections V-A and V-B, respectively.

A. Local Index Construction

In this subsection, we describe the construction process of the local index. For each data source containing a collection of spatial datasets $\mathcal{D} = \{D_1, \ldots, D_n\}$, we devise a local index to accelerate **OJSP** and **CJSP** (see Fig. 4). For ease of understanding, we take a single data source as an example to illustrate the local index construction. Before constructing the local index, we first transform each spatial dataset D into a dataset node N_D containing specific information. The formal definition of a dataset node is presented below.

Algorithm 1: `LocalIndex`$(\mathcal{N}_\mathcal{D}, pa, f, d)$

Input: $\mathcal{N}_\mathcal{D}$: a list of dataset nodes, pa: parent node, f: capacity, d: dimension
Output: N_{root}: the root node of local index

$N_{root} \leftarrow$ generate the root node of $\mathcal{N}_\mathcal{D}$;
$N_{root}.setParentNode(pa)$;
$max\_width \leftarrow -\infty$, $d_{split} \leftarrow 0$; ▷ split dimension
if $|\mathcal{N}_\mathcal{D}| \leq f$ **then**
  foreach $N_D \in \mathcal{N}_\mathcal{D}$ **do**
    $N_D.setParentNode(N_{root})$;
    $N_{root}.ch.add(N_D)$;
  creating an inverted index for N_{root};
else
  $leftList, rightList \leftarrow \emptyset$; ▷ represent left and right subsets of N_{root}
  foreach $i \leftarrow 1 : d$ **do**
    if *the width of* $N_{root}.rect[i] > max\_width$ **then**
      $max\_width \leftarrow$ the width of $N_{root}.rect[i]$;
      $d_{split} \leftarrow i$;
  foreach $N_D \in \mathcal{N}_\mathcal{D}$ **do**
    if $N_D.o[d_{split}] \leq N_{root}.o[d_{split}]$ **then**
      $leftList.add(N_D)$;
    else
      $rightList.add(N_D)$;
  $N_{root}.N_l \leftarrow$ `LocalIndex`$(leftList, N_{root}, f, d)$;
  $N_{root}.N_r \leftarrow$ `LocalIndex`$(rightList, N_{root}, f, d)$;
return N_{root};

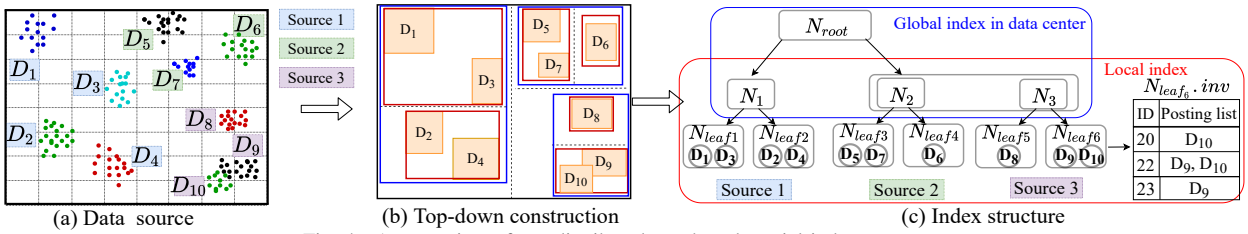
Definition 12. ***(Dataset Node)*** *A dataset node* $N_D = (id, rect, o, r, S_D, pa)$*, where* id *is an dataset identifier,* $rect$ *is its minimum bounding rectangle (MBR),* o *is the pivot of the MBR,* r *is the radius of node,* S_D *is its cell-based dataset, and* pa *is the pointer address of the parent node.*

The $rect$ is a rectangle whose sides are parallel to the x and y axes and minimally enclose all points in the dataset D. We compute the pivot o by averaging the coordinates of the bottom left and upper right corners of the $rect$, and choose half of the farthest diagonal distance as the radius r. Next, we will give the formal definitions of the internal node and leaf node used in the local index.

Definition 13. ***(Internal Node)*** *An internal node* $N_{in} = (rect, o, r, N_l, N_r, pa)$*, where* $rect$ *is the MBR enclosing all child nodes,* N_l *and* N_r *are its left and right child nodes. The meaning of the other symbols in the tuple is the same as in Definition 12. The root node* N_{root} *is an internal node, which has the same structure with* N_{in} *except without* pa*.*

Definition 14. ***(Leaf Node)*** *A leaf node* $N_{leaf} = (rect, o, r, ch, inv, f, pa)$*, where* ch *contains its all child dataset nodes,* inv *is an inverted index of the contained dataset nodes, containing the posting lists that map the cell ID to a list of dataset IDs that contain it, and* f *is the leaf node's capacity. The other symbols' meaning is the same as in Definition 13.*

Firstly, we use n to represent the number of datasets in the data source. When constructing the local index, the bottom-up



(a) Data source (b) Top-down construction (c) Index structure

Fig. 4. An overview of our distributed tree-based spatial index structure.

construction needs to repeatedly find the two balls that make the parent node's MBR volume smallest, and insert the parent node into the tree. The total time complexity is $O(n^3)$ [38]. In contrast, for the top-down construction approach, the algorithm only needs to split the balls into two sets according to the coordinate of the dataset node in the selected dimension, which has a time complexity of $O(n \log n)$. Thus, we take a top-down approach to construct the local index.

In Algorithm 1, we show the whole construction process of the local index. Specifically, for a list of dataset nodes $\mathcal{N}_{\mathcal{D}}$ constructed from $\mathcal{D}$, we construct a node N_{root} that contains all dataset nodes as the root node. Then, we compute the range of coordinate values of the root node in each dimension. We choose the axis with the maximum width as the split dimension and calculate the median of pivot points of all dataset nodes on that dimension, dividing them into left and right subsets based on their position concerning the median (Lines 15 to 19). Next, we continue to build the sub-tree for each of the left and right sets of dataset nodes (Lines 20 to 21). The construction process continues until the size of the contained dataset nodes does not exceed the leaf node capacity f. At this point, we create a leaf node N_{leaf} containing an inverted index.

Example 4. *As shown in Fig. 4(a), there are ten datasets $\{D_1, \ldots, D_{10}\}$ in three data sources, and the leaf node capacity f is set to 2. Then we create the local index for each data source. For example, for the 3-th data source, we first transform datasets $\{D_8, D_9, D_{10}\}$ into three dataset nodes in Fig. 4(b). Then, we construct the root node N_3 that contains all dataset nodes. We can find the widest dimension and split the dataset nodes into two groups along the x-axis. Since the number of dataset nodes per group does not exceed f, the partitioning process is terminated. The final index structure is shown in the right figure of Fig. 4(c), where the leaf node contains an inverted index that stores a mapping from the cell ID to all dataset IDs.*

B. Global Index Construction

When the search scenario includes multiple data sources, one popular solution is to apply a traditional distributed framework with a master-slave structure [4], [23], [63], [67]. It requires all datasets of slave nodes to be uploaded to the master, then the master node partitions all datasets into equal chunks and stores them separately in the slave node to build indexes of the same setting, such as node capacity, resolution, etc. The partition information held by the master node serves as the global index. However, in our search scenario, each data source is independent. Thus, we propose a global index based on the local indexes, which not only plays the role of an "overview" index but is also suitable for situations where each data source builds its index in different settings.

The global index contains root nodes from local indexes and is used when sending a query to candidate data sources. Thus, after the local index is built, each source sends its root node N_{root} to the data center. Then, the data center converts the pivot and MBR coordinates of N_{root} into latitude and longitude to resolve the situation where the resolution of the data sources differs. Next, the data center generates the global root node containing the region of all local root nodes and chooses a dimension to split it into two child sets until it does not exceed the leaf node capacity f. We use an example to illustrate the construction process of the global index.

Example 5. *As shown in Fig. 4(c), each data source sends its root node to the data center. The data center then builds the global index from top to bottom based on the received node information, the construction process is similar to building the local index. The construction process stops until each leaf node meets the leaf node's capacity, but there is no need to generate the inverted index for leaf nodes.*

To deal with the constant update of spatial datasets in each data source, we also design index updating strategies to avoid reconstructing the index. Due to the page limitation, the detailed index update strategies can be found in Appendix IX-C. Additionally, we theoretically analyze the time and space complexity of constructing DITS, which is $O(n \log n)$ and $O(n)$, respectively. The detailed analysis is provided in Appendix IX-D.

VI. Accelerating Joinable Search

In this paper, our goal is to find k datasets from multiple data sources that have the maximum overlap or coverage with the given query dataset. While the existing distributed queries mainly focus on the range query [4], [23], k-nearest neighbor (kNN) [5], [23], and approximate nearest neighbor (ANN) [34], they cannot solve our problem. Thus, we first propose the query distribution strategy to support the efficient communication between the data center and each data source in Section VI-A. Secondly, considering the large scale of datasets in each data source and the high computational complexity of the problem, we design efficient dataset search algorithms and acceleration strategies to solve **OJSP** and **CJSP** in Sections VI-B and VI-C.

A. Query Distribution

Since the overall search process involves communication between the data source and the data center, as well as local

search within the data source, communication cost, and search efficiency are two main bottlenecks in our search framework. Thus, we design two query distribution strategies to solve the above problems. The first strategy is to reduce the frequency of communication. Based on the global index, we can quickly find candidate data sources that may contain query results and transfer the query request only to them.

Specifically, when a query comes in, we recursively search the global index starting from the root node. For a tree node N, we can prune the tree node N directly if it does not overlap with the MBR of the query node ($N.rect \cap N_Q.rect = \emptyset$), or the shortest possible distance between it and the nearest neighbor cell of the query node is greater than the connectivity threshold ($dist(N.o, N_Q.o) - N.r - N_Q.r \geq \delta$); otherwise, we perform a depth-first traversal on it until it is a leaf node. All leaf nodes that intersect with $N_Q.rect$ or possibly connected to N_Q form the candidate data sources. The data center transmits the query only to candidate sources, which reduces the communication cost of query distribution.

The second strategy is to reduce the number of bytes transferred during each communication between the data center and data sources. Because intersections only exist in the area where the MBR intersects, it is unnecessary to transmit the entire S_Q to each candidate source when performing the static search. The data center only transfers the portion of the query that has an MBR intersect to the candidate source to implement the local search to further reduce the communication cost.

B. Accelerating Overlap Joinable Search

The overlap joinable search mainly focuses on finding k datasets with the greatest number of intersections with the query. In this section, we propose an algorithm called OverlapSearch based on the local index to accelerate **OJSP**. Since the intersection number of two datasets whose MBR does not intersect is 0, we can directly filter out those tree nodes that do not intersect with the query's MBR. For those nodes intersecting with the query node, we propose lower and upper bounds based on the leaf node's posting list.

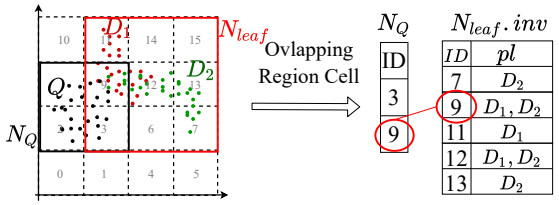
Lemma 2. ***(Upper Bound)*** *Let $S_Q = \{c_1, c_2, \ldots, c_n\}$ represent the cell-based set representation of a query node, the upper bound of intersection between leaf node N_{leaf} and query node N_Q is $\sum_{i=1}^{n} \varphi(c_i)$.*

$$\varphi(c_i) = \begin{cases} 1, & if\ c_i \in N_{leaf}.inv, \\ 0, & otherwise \end{cases}$$

Proof. As shown in Fig. 5, for a cell ID c_i of N_Q, if it is also stored by the key set of the leaf node, which indicates that at least one dataset intersects with N_Q. Therefore, the total number of cells that intersect with the key set of the inverted file is the upper bound between N_Q and N_{leaf}. □

Lemma 3. ***(Lower Bound)*** *The lower bound of intersection between N_{leaf} and N_Q is $\sum_{i=1}^{n} \varphi(c_i)$.*

$$\varphi(c_i) = \begin{cases} 1, & if\ c_i \in N_{leaf}.inv \& |c_i.pl| = |N_{leaf}.ch|, \\ 0, & otherwise \end{cases}$$



(a) Original datasets (b) UB = 1, LB = 1

Fig. 5. Upper and lower bounds of the set intersection.

Proof. From Fig. 5 we can observe that the posting list of $N_{leaf}.inv$ stores the mapping from the cell ID to the spatial dataset. If the size of the $c_i.pl$ in $N_{leaf}.inv$ is equal to the current leaf node's capacity $|N_{leaf}.ch|$, then all dataset nodes contained by the N_{leaf} must contain cell ID c_i. Therefore, the lower bound between the leaf node and the query node is the total number of cell IDs whose corresponding list size is equal to the current leaf node's capacity $|N_{leaf}.ch|$. □

The pseudocode of OverlapSearch can be shown in Algorithm 2. Firstly, for a given query node N_Q, we implement a recursive search down DITS-L in which we prune away recursive contain internal nodes whose MBR does not overlap the query node (Lines 24 to 26). Then, for each intersected leaf node N, we compute the upper and lower bounds between N and N_Q. If the upper bound is less than the current lower bound in the priority queue, all dataset nodes contained by the leaf node can be safely pruned in batch; otherwise, it will become a candidate leaf node (Lines 16 to 22).

After obtaining the first-round candidate nodes, we traverse each candidate leaf node N_{leaf} and compute the set intersection by scanning the posting lists of N_{leaf}. When the size of results queue $\mathcal{R}.size < k$, the candidate dataset is directly inserted into $\mathcal{R}$; otherwise, we compare the set intersection of each candidate dataset with the k-th largest value in $\mathcal{R}$ and judge whether to insert it into $\mathcal{R}$. Finally, we can obtain the set of results $\mathcal{R}$ (Lines 3 to 12). Additionally, OverlapSearch incurs $O(n)$ time complexity. The detailed time complexity analysis of the Algorithm 2 can be found in Appendix IX-D.

C. Accelerating Coverage Joinable Search

For the coverage joinable search, we formulate it into an NP-hard **CJSP** problem, aiming to find k spatial datasets with the maximum coverage and satisfy the spatial connectivity. It is evident that achieving the optimal solution for **CJSP** is computationally prohibitive. In addition, unlike the classic MCP [30], our **CJSP** is a query-driven search problem with the spatial connectivity constraint. The basic greedy solution is that we iteratively traverse all datasets in the data source to find datasets directly connected to each dataset in the result set $\mathcal{R}$, and then select the dataset with the maximum marginal gain to add to the result set. Thus, the time complexity of each round search is $O(|\mathcal{R}|n)$, where $|\mathcal{R}|$ denotes the number of result datasets and n denotes the number of datasets in the data source. As the number of result datasets $|\mathcal{R}|$ gradually increases (from 1 to k), the total time complexity is $O(n + 2n + \cdots + kn) = O(nk(k-1))$.

To accelerate the search process, we design a greedy algorithm with spatial merge based on the local index, called

Algorithm 2: `OverlapSearch`(N_{root}, N_Q, k)

Input: N_{root}: root node of the local index, N_Q: query node, k: number of results
Output: $\mathcal{R}$: result queue

1 $PQ, \mathcal{R} \leftarrow$ Initialize two priority queues;
2 `BranchAndBound`(N_{root}, N_Q, PQ);
3 **foreach** $N_{leaf} \in PQ$ **do**
4 Compute the exact intersection of all dataset nodes contained by N_{leaf} with N_Q according to $N_{leaf}.inv$;
5 **foreach** $N_D \in N_{leaf}$ **do**
6 **if** $\mathcal{R}.size \leq k$ **then**
7 $\mathcal{R}$.Insert(N_D);
8 **else**
9 **if** $|N_D \cap N_Q| > \mathcal{R}.peek()$ **then**
10 $\mathcal{R}$.Dequeue();
11 $\mathcal{R}$.Insert(N_D);
12 **return** $\mathcal{R}$;
13
Function `BranchAndBound` (N, N_Q, PQ) **:**
Input: N: tree node, N_Q: query node, PQ: priority queue;
14 **if** N *is leaf node* **then**
15 **if** $N.rect \cap N_Q.rect \neq \emptyset$ **then**
16 $N.lb, N.ub \leftarrow$ Compute the lower and upper bounds based on the Lemmas 2 and 3;
17 **if** $PQ.isEmpty()$ **then**
18 PQ.Insert(N);
19 **if** $N.ub > PQ.head().lb$ **then**
20 **while** $N.lb \geq PQ.head().ub$ *and* $PQ.size \geq k$ **do**
21 PQ.Dequeue();
22 PQ.Insert(N);
23 **else**
24 **if** $N.rect \cap N_Q.rect \neq \emptyset$ **then**
25 `BranchAndBound`$(N.N_l, N_Q, PQ)$;
26 `BranchAndBound`$(N.N_r, N_Q, PQ)$;

CoverageSearch. Specifically, we first design efficient upper and lower bounds based on the local index, which greatly accelerates the time of finding the connected datasets. Secondly, we design a merge strategy, which merges the found results into a large dataset for searching, reducing the number of searches and effectively solving **CJSP**.

In solving **CJSP**, we define the marginal gain $g(S_D, \mathcal{R})$ as the increased number of cell IDs in the result set $\mathcal{R}$ after adding a new cell-based dataset S_D:

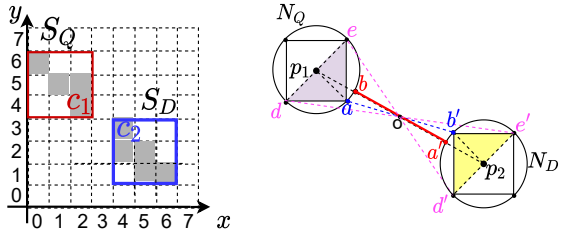
$$g(S_D, \mathcal{R}) = |S_D \cup (\cup_{S_{D_i} \in \mathcal{R}} S_{D_i})| - |\cup_{S_{D_i} \in \mathcal{R}} S_{D_i}|. \quad (3)$$

The strategy of our proposed greedy algorithm is to iteratively pick the dataset that satisfies the spatial connectivity and has the maximum marginal gain. However, it is expensive to compute the spatial connectivity for each pair of datasets. Thus, we derive lower and upper bounds for distances based on the triangle inequality to accelerate the connectivity verification.

Lemma 4. *Let N_Q be a node of the query set S_Q, N_D be a node of the cell-based dataset S_D. The distance $dist(S_Q, S_D)$ between N_Q and N_D can be bounded in the range:*

$$[\max\{||N_Q.p, N_D.p||_2 - N_Q.r - N_D.r, 0\}, \\ ||N_Q.p, N_D.p||_2 + N_Q.r + N_D.r\,]. \quad (4)$$

Proof. We first prove the correctness of the lower bound. As shown in Fig. 6, N_Q and N_D are two nodes transformed by datasets S_Q and S_D. Let a and b denote the two closest points in two MBRs, and a' and b' denote two intersect points in



(a) Cell-based dataset distance (b) Lower and upper bounds

Fig. 6. Lower and Upper bounds of the cell-based dataset distance.

the line segment connected by two centers p_1 and p_2. The triangle composed of points p_1, a and o is denoted by $\triangle p_1 ao$. According to the triangle inequality, we have:

$$||p_1, a||_2 + ||a, o||_2 \geq ||p_1, o||_2. \quad (5)$$

Next, based on Inequation (5) and $||p_1, a||_2 = ||p_1, b||_2$, we can deduce that $||a, o||_2 \geq ||b, o||_2$. Similarly, in $\triangle p_2 b' o$, we can deduce $||b', o||_2 \geq ||a', o||_2$. Hence, $||a, b'||_2 \geq ||a', b||_2$. As the dataset distance $dist(S_Q, S_D)$ is to find the minimum of the distance from an element in S_D to its nearest neighbor element in S_Q, $dist(S_Q, S_D) = ||a, b'||_2 \geq ||b, a'||_2 = \max\{||N_Q.p, N_D.p||_2 - N_Q.r - N_D.r, 0\}$.

In the following, we prove the correctness of the upper bound. As shown in Fig. 6, each ball is divided into two half-balls by the diagonal de and $d'e'$, each part should have at least one cell. Thus, the maximum distance between two nodes' nearest neighbor cells is $\max\{||d, e'||_2, ||d', e||\}$. Similarly, according to the triangle inequality, we can find that $||d, e'||_2 \leq ||p_1, p_2||_2 + ||p_1, d||_2 + ||p_2, e'||$ and $||d', e||_2 \leq ||p_1, p_2||_2 + ||p_1, e||_2 + ||p_2, d'||$. Thus, the upper bound of the dataset distance $dist(S_Q, S_D) \leq ||N_Q.p, N_D.p||_2 + N_Q.r + N_D.r$. □

Example 6. *As shown in Fig. 6, we can compute the dataset distance between the query dataset S_Q and the cell-based dataset S_D is $dist(S_Q, S_D) = ||q_1, d_1||_2 = \sqrt{5} \approx 2.236$. While based on Lemma 4, we can compute the lower bound of the dataset distance is $\max\{5 - \sqrt{2} - \sqrt{2}, 0\} \approx 2.172 \leq 2.236$, and the upper bound is $5 + \sqrt{2} + \sqrt{2} \approx 7.828 \geq 2.236$, which proves the validity of the lower and upper bounds.*

Algorithm 3 shows the pseudocode of CoverageSearch algorithm based on the lower and upper bounds. Specifically, we first initialize a result set $\mathcal{R}$ and put query node N_Q into it. Then, we define a merged node N_M, which is constructed from the merged set $\cup_{N_i \in \mathcal{R}}$. Next, we perform k depth-first search in the local index and find the dataset $S_D \in (\mathcal{S}_\mathcal{D} \backslash \mathcal{R})$ that is connected to the S_i and has the maximum marginal gain $g(S_D, \mathcal{R})$ in each iteration (Lines 14 to 26).

For each connected dataset node, we can decide whether to filter based on the maximum marginal gain it can bring. In each round of searching, we use a symbol τ to represent the maximum marginal gain found so far. Then, for the dataset node that satisfies $|N_D.S_D| > \tau$, we compute the exact number of coverage of each dataset node and judge whether it is the local optimal set (Lines 5 to 9). By repeating this step until the whole candidate set is traversed, we can add the set that satisfies the connectivity and has the maximum increment to $\mathcal{R}$. After k iterations, the algorithm terminates.

Algorithm 3: `CoverageSearch`$(N_{root}, N_Q, \delta, k)$

Input: N_{root}: root node of local index, N_Q: query node, δ: connectivity threshold, k: number of results
Output: $\mathcal{R}$: result set

$List \leftarrow \emptyset$, $\mathcal{R} \leftarrow \{N_Q\}$, $N_M \leftarrow N_Q$;
while $\mathcal{R}.size() \leq k$ **do**
  $\tau \leftarrow -\infty$, $N_{best} \leftarrow null$;
  `FindConnectSet`$(N_{root}, N_M, \delta, List)$;
  foreach $N_D \in List$ **do**
    if $|N_D.S_D| > \tau$ **then**
      if $g(N_D, \mathcal{R}) > \tau$ **then**
        $N_{best} \leftarrow N_D$;
        $\tau \leftarrow g(N_D, \mathcal{R})$;
  $\mathcal{R}.add(N_{best})$;
  $N_M \leftarrow$ Merge N_M and N_{best};
return $\mathcal{R}$;

Function `FindConnectSet` $(N, N_Q, \delta, List)$ **:**
  Input: N: tree node, N_Q: query node, δ: connectivity threshold, List: list of dataset nodes that directly connected to N_Q;
  $lb \leftarrow \max\{||N.p, N_Q.p||_2 - N.r - N_Q.r, 0\}$;
  $ub \leftarrow ||N.p, N_Q.p||_2 + N.r + N_Q.r$;
  if $ub \leq \delta$ **then**
    List.add(all dataset nodes contained by N);
  else
    if $lb \leq \delta$ **then**
      if N *is leaf node* **then**
        foreach $N_D \in N.ch$ **do**
          if $dist(N, N_D) \leq \delta$ **then**
            List.add(N_D);
      else
        `FindConnectSet`$(N.N_l, N_Q, \delta, List)$;
        `FindConnectSet`$(N.N_r, N_Q, \delta, List)$;

The CoverageSearch algorithm has a time complexity of $O(kn)$ and provides an approximation ratio of $1 - 1/e$ under certain assumptions. Please refer to Appendixes IX-D and IX-E for details due to the page limitation.

VII. Experiments

In Section VII-A, we first introduce the experimental settings. Then, we evaluate the efficiency of the index construction in Section VII-B and the search performance and communication cost of **OJSP** in Section VII-C. Afterwards, we present the search performance and communication cost of **CJSP** in Section VII-D.

A. Experimental Setups

Datasets. We conduct experiments on the following five data sources, each of which is downloaded from an open-source spatial data portal. Table I shows the statistics of five data sources. Fig. 7 presents each data source's dataset distribution heatmap, showing the spatial datasets' distribution density in the space. In addition, further details about these five data sources are provided in Appendix IX-F.

Environment. To implement the multi-source joinable search on collected five data sources, we conducted all experiments on devices equipped with a 2.90 GHz Intel(R) Core(TM) i5-12400 processor and 16GB of memory. The implementation code can be obtained from the GitHub repository[1].

[1]https://github.com/whu-totemdb/DITS_code

Query and Parameter Generation. We randomly select 50 datasets from all downloaded datasets as the query datasets. We alter several key parameters including k, q, θ, δ, f to observe the experimental results. Firstly, the parameter θ determines the grid's size; thus, we set the resolution according to the distance sampling [51]. For example, one degree of longitude or latitude is about 111km. If we divide the globe into a $2^{12} \times 2^{12}$ grid, then each cell's area is about $10km \times 5km$. Similarly, the δ can be set according to the closest distance between the point pairs of two spatial datasets that the user requires. For other parameters, we can also set them according to the requirements of users and data sources. We summarize the parameter settings in Table II, and the default value for each parameter is underlined.

B. Efficiency of Index Constructionn

To accelerate the search performance of **OJSP** and **CJSP** in each data source, we design the local index DITS-L and show its index construction performance compared with four sota indexes in Fig. 8. Since DITS-L combines the characteristics of the tree index and inverted index, we choose two classical tree indexes and two inverted indexes for comparison.

- QuadTree [26]: QuadTree divides the whole space containing all datasets into four equal-sized quadrants, and then recursively subdivides each quadrant if it contains a sufficient number of data points.
- Rtree [27]: Rtree groups nearby objects and represent them with their minimum bounding rectangle (MBR) in the inner nodes of the tree.
- STS3 [39]: STS3 divides the plane into cells and constructs an inverted index to accelerate the intersection computation.
- Josie [73]: Josie designs a sorted inverted index where each posting list contains the id, position, and size of datasets.

Time and Space Comparison of Multiple Indexes. We theoretically compare the index construction time and memory space of DITS-L and four indexes as the θ increases, as shown in Fig. 8. We use n to denote the number of datasets, and N to denote the total number of cell IDs contained by all datasets. The time complexity of STS3 [39], Rtree and DITS-L is $O(n \log n)$; the time complexity of Josie [73] is $O(n^2)$, while QuadTree has a time complexity of $O(N \log N)$. Thus, from the left of Fig. 8, we observe that Josie is slower than the other indexes in most cases, because it takes more time to sort datasets. Additionally, we can observe that STS3 is the fastest at lower θ. This is because when θ is small, the size of each cell-based dataset is also small. Compared to STS3, DITS-L takes extra time to construct the tree index. Furthermore, we can observe that DITS-L is always slightly faster than Rtree. This is because Rtree is a balanced search tree, which needs more time to organize the spatial datasets' MBR.

We also compare the memory space of the five indexes theoretically. Firstly, from the right of Fig. 8, we can observe that the memory space of five indexes gradually increases as θ increases. This is because a higher resolution results in an increase in the number of cell IDs in each cell-based dataset. In addition, QuadTree occupies the largest memory space,

TABLE I
DETAILS OF FIVE SPATIAL DATA SOURCES.

Data source	Storage (GB)	Number of datasets	Number of points	Coordinates range
Baidu-dataset	0.18	6,581	[3,710,526]	$[(87°31', 19°59'), (127°09', 46°21')]$
BTAA-dataset	3.39	3,204	[96,788,280]	$[(-179°46', -87°70'), (179°99', 71°40')]$
NYU-dataset	0.89	1,093	[15,303,410]	$[(-138°00', -74°01'), (56°39', 83°09')]$
Transit-dataset	0.21	1,967	[522,461]	$[(-77°73', 36°81'), (-74°53', 39°78')]$
UMN-dataset	1.34	5,453	[54,417,609]	$[(-179°14', -14°55'), (179°77', 71°35')]$

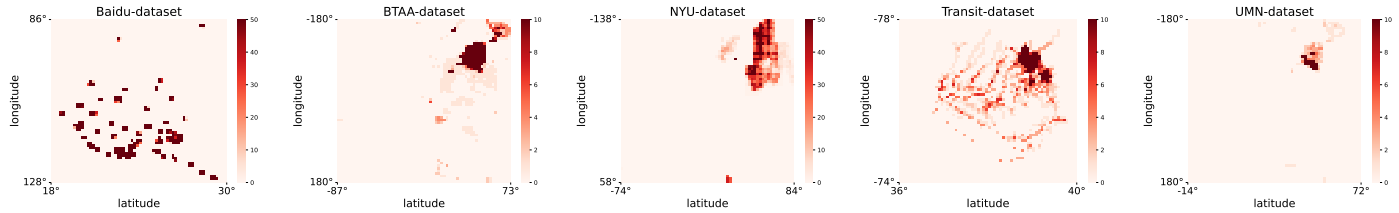


Fig. 7. The heatmaps of five data sources.

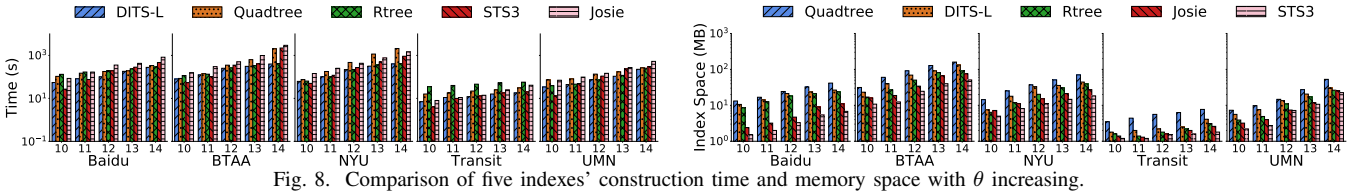


Fig. 8. Comparison of five indexes' construction time and memory space with θ increasing.

TABLE II
PARAMETER SETTINGS.

Parameter	Settings
k: number of results	{10, 20, 30, 40, 50}
q: number of queries	{ 10, 20, 30, 40, 50 }
θ: resolution	{10, 11, 12, 13, 14 }
δ: connectivity threshold	{0, 5, 10, 15, 20 }
f: leaf node capacity	{10, 20, 30, 40, 50 }

followed by DITS-L, while STS3 requires the least memory. This is because QuadTree is built based on the cell IDs of all datasets, and the height of the tree is $O(\log N)$. In contrast, DITS-L and Rtree are built based on the dataset, and the height of the tree is $O(\log n)(N >> n)$. Thus, QuadTree stores more index nodes compared to DITS-L and Rtree.

C. Efficiency of Overlap Search and Communication

Methods for Comparisons. We compare our proposed exact OverlapSearch with four exact algorithms based on the four indexes introduced in Section VII-B. When performing **OJSP** based on QuadTree [26], we find all leaf nodes that intersect the query and count the dataset IDs of the points that overlap with the query dataset. Moreover, when performing **OJSP** based on Rtree [27], we find all intersecting datasets based on their MBR, and compute the size of the set intersection. For STS3 [39] and Josie [73], we directly apply them to cell-based datasets to compute set intersection.

1) Efficiency of Overlap Joinable Search

In this section, we present the experimental results of OverlapSearch algorithm and the four comparison algorithms that vary with several key parameters on five data sources.

Efficiency Comparison with Different k. As shown in Fig. 9, we can see that OverlapSearch achieves 1.7 to 4.8 times speedup compared with the other four algorithms, indicating that our distributed index and pruning strategies are effective. Although Rtree and QuadTree also apply the tree index, Rtree shows the second-best performance compared with QuadTree, as QuadTree is constructed based on the cell IDs of all spatial datasets, we need to find all set elements that intersect with the query and record the sorted results, which is similar to the inverted index. Moreover, we can see that Josie runs faster than STS3 because Josie can terminate the search early by the prefix filter. We can see that for the search algorithms based on OverlapSearch, Rtree and Josie, the runtime shows a slight increase. In contrast, the running time of QuadTree and STS3 remains relatively unchanged with increasing k. The reason is that they need to sort all datasets that intersect with the query according to the number of overlapping cell IDs. Thus, the change in k has minimal impact on runtime.

Efficiency Comparison with Different θ. We increase θ to investigate the search performance of OverlapSearch compared with the other four algorithms. We can observe that the search time of the five algorithms increases gradually as the θ increases in Fig. 10. The reason is that each spatial dataset is divided into finer granularity as the θ increases. However, OverlapSearch is always 1.8 to 6.7 times faster than the other four algorithms under different θ. The reason is that our algorithm can quickly filter out unpromising leaf nodes using DITS-L and bounds. In addition, we can also see that the two algorithms based on the tree index maintain a better search performance than the other two based on the inverted index, indicating that filtering based on the geographic characteristics of the spatial dataset can provide better search performance.

Efficiency Comparison with Different q. Fig. 11 shows the search performance as the q increases. We can see that the search time of OverlapSearch and Rtree increases slightly. The running time at $q = 50$ increases only by about $3\times$

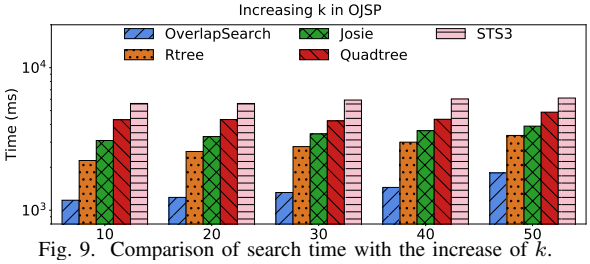


Fig. 9. Comparison of search time with the increase of k.

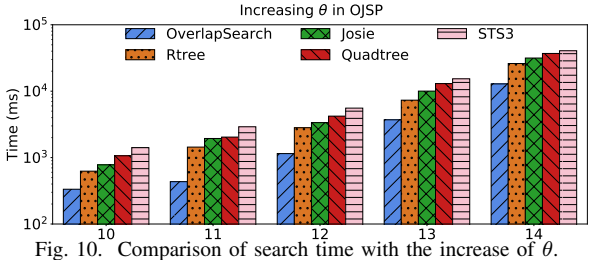


Fig. 10. Comparison of search time with the increase of θ.

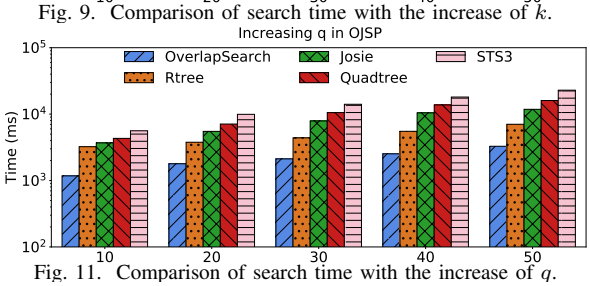


Fig. 11. Comparison of search time with the increase of q.

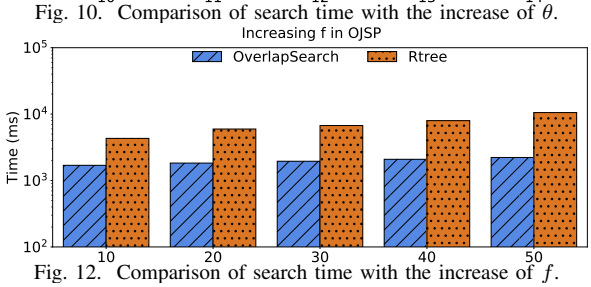


Fig. 12. Comparison of search time with the increase of f.

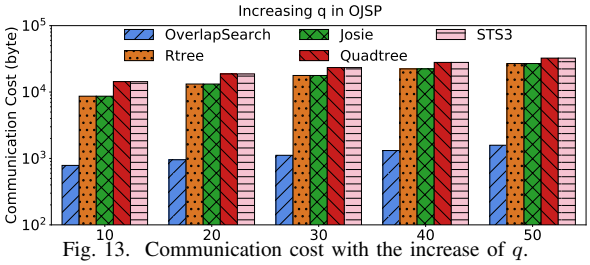


Fig. 13. Communication cost with the increase of q.

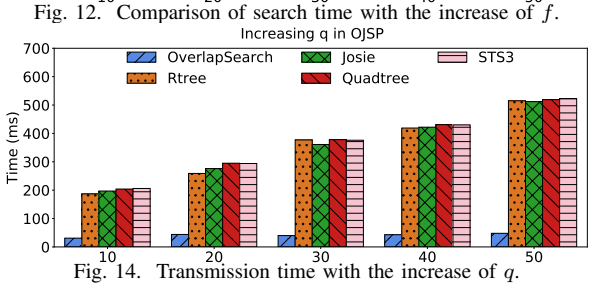


Fig. 14. Transmission time with the increase of q.

compared to the running time at $q = 10$. In contrast, the other four algorithms have an increase of $4\times$ or more. The reason is that the search based on the tree index has a faster filtering power than the other three algorithms, so it can show more stable search performance as the number of queries increases. In addition, the search based on QuadTree is to find the intersection point, which is similar to the inverted index, thus it has a slower time than the other two tree algorithms.

Efficiency Comparison with Different f. We also investigate the search performance as f increases, and the results are shown in Fig. 12. Here, we only show the experiment results of OverlapSearch and Rtree, as the leaf node capacity in QuadTree is 4, STS3 and Josie are based on the inverted index. We can observe that the running time of OverlapSearch increases slightly as the f increases. This is because bigger f cannot be pruned easily, which degrades the performance. However, our algorithm still has a better performance than Rtree, as our algorithm has a stronger filtering power and faster computation time based on the lower and upper bounds.

2) Efficiency of Communication Cost

We compare the communication cost of OverlapSearch with query distribution strategy compared to other baselines as the q increases. Fig. 13 shows the number of bytes transferred as the q increases. We can see that the number of bytes transferred by five algorithms gradually increases as the q increases, wherein OverlapSearch transmits the fewest bytes. This is because OverlapSearch only transmits query region that overlaps with the candidate dataset source. Fig. 14 shows the transmission time. Since the transmission time is inversely proportional to the network bandwidth when the network bandwidth is constant, the more bytes that are transmitted, the longer the transmission time will be.

D. Efficiency of Coverage Search and Communication

Methods for Comparisons. We compare our proposed CoverageSearch algorithm in Section VI-C with two baselines.

- SG [30]: It is a standard greedy algorithm for solving MCP. Here, we extend it to solve our **CJSP**, that is, we traverse all datasets in the data source and find the dataset that is directly or indirectly connected with the query and has the maximum marginal gain at each iteration.
- SG+DITS: It adopts DITS-L to speed up the search process of finding connected candidate datasets each round, and DITS-G to find the candidate data sources.

1) Efficiency of Coverage Joinable Search

We vary key parameters k, θ, q, and δ to compare the efficiency of CoverageSearch with two comparison algorithms.

Efficiency Comparison with Different k. Fig. 15 shows the search time of three algorithms as k increases. We observe that CoverageSearch achieves at most 1.7 and 26.5 times speed up against SG+DITS and SG, respectively. Because CoverageSearch only needs to search in the index tree once to find the node connected with the query in each iteration, significantly reducing the search time. In addition, we can see that SG+DITS is significantly faster than SG, which indicates our local index is very efficient in solving **CJSP**.

Efficiency Comparison with Different θ. Fig. 16 presents the search time as the θ increases. We can see that the search time of the three algorithms gradually increases, but CoverageSearch always maintains the best search performance. Moreover, the search time of SG algorithm increases more

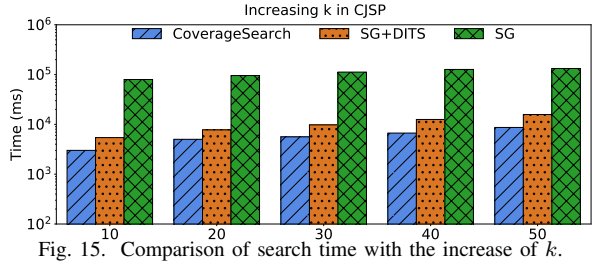


Fig. 15. Comparison of search time with the increase of k.

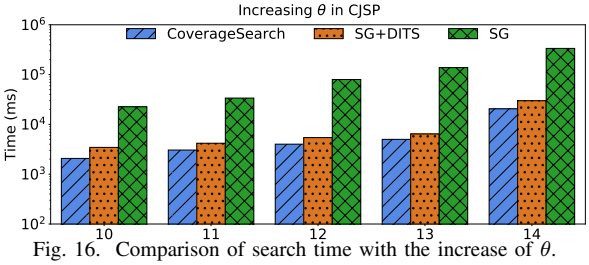


Fig. 16. Comparison of search time with the increase of θ.

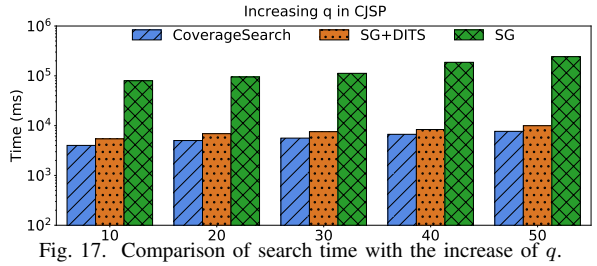


Fig. 17. Comparison of search time with the increase of q.

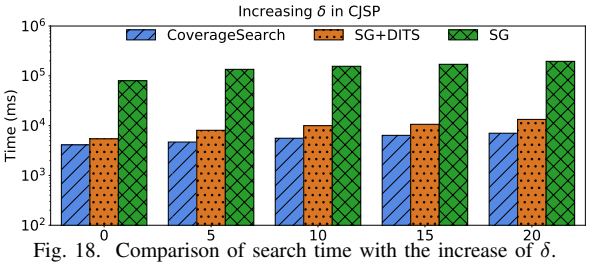


Fig. 18. Comparison of search time with the increase of δ.

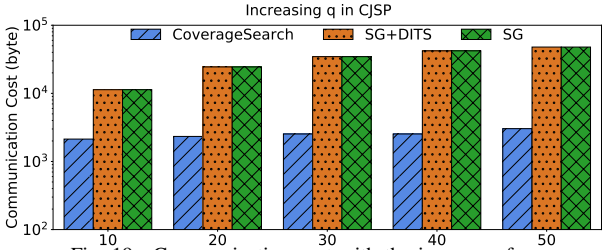


Fig. 19. Communication cost with the increase of q.

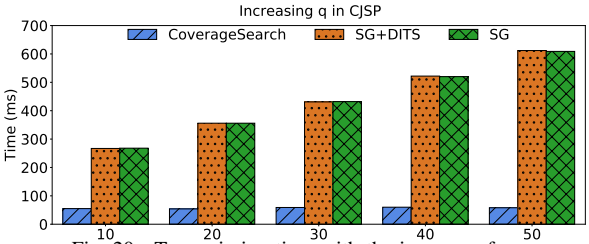


Fig. 20. Transmission time with the increase of q.

rapidly than that of the other two algorithms. This is because the number of elements in each cell-based dataset increases as the θ increases and pairwise coverage computation and comparison will take more time.

Efficiency Comparison with Different q. Fig. 17 shows the search time as q increases. We can see that CoverageSearch consistently outperforms the other two algorithms. This is because our algorithm only needs to search once in each iteration by merging all results into one large query dataset to find connected datasets, reducing the search time. Moreover, we can see that SG+DITS has better performance than SG, which proves that our lower and upper bounds based on DITS have stronger filtering power in finding the candidate datasets that satisfy spatial connectivity.

Efficiency Comparison with Different δ. Fig. 18 shows the search time of three algorithms as δ increases. We can see that the search time of the three algorithms gradually increases. This is because a higher δ means that more pairs of datasets satisfy the directly connected relation, resulting in more candidate datasets that satisfy the spatial connectivity, the algorithm needs to find the dataset with the maximum marginal gain by traversing all connected datasets. Moreover, we can see that CoverageSearch consistently outperforms the other two algorithms, which further shows the spatial merge strategy in CoverageSearch is efficient in solving **CJSP**.

2) Efficiency of Communication Cost

Figs. 19 and 20 present the communication cost and network transmission time of three algorithms. We can observe that as q increases, the number of bytes transmitted by three algorithms gradually increases. The reason is that CoverageSearch only sends the query that overlaps with the MBR of the root node of the data source each iteration, without sending the entire query dataset. Correspondingly, since the transfer time is proportional to the bytes transferred, the transfer time of CoverageSearch is more rapid than that of the other two algorithms, as shown in Fig. 20.

VIII. Conclusions

This paper mainly studied joinable search over multiple spatial data sources and formulated two search problems, including overlap joinable search problem (**OJSP**) and coverage joinable search problem (**CJSP**). To solve them, we first proposed a distributed tree-based spatial index structure DITS, which is composed of one centralized global index and distributed local indices–one per data source. Next, based on DITS, we designed query distribution strategies to reduce the communication cost. Subsequently, we proposed efficient lower and upper bounds and designed an exact OverlapSearch algorithm to solve **OJSP**. Moreover, we also deduced the lower and upper bounds between two index nodes to accelerate the computation of spatial connectivity, and designed an approximate CoverageSearch algorithm to solve **CJSP** efficiently. The experimental results showed that our proposed algorithms improved search performance and reduced communication costs. An interesting future research direction is to explore the spatial dataset search based on the data pricing to return the optimal dataset combination.

REFERENCES

[1] Geoblacklight. https://geoblacklight.org/.
[2] OpenGeoHub. https://github.com/OpenGeoHub/.
[3] Opengeometadata. https://github.com/OpenGeoMetadata/.
[4] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.
[5] A. Akdogan, U. Demiryurek, F. B. Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In *CloudCom*, pages 9–16, 2010.
[6] M. E. Ali, S. S. Eusuf, K. Abdullah, F. M. Choudhury, J. S. Culpepper, and T. Sellis. The maximum trajectory coverage query in spatial databases. *PVLDB*, 12(3):197–209, 2018.
[7] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.
[8] P. Bouros and N. Mamoulis. Spatial joins: what's next? *ACM SIGSPATIAL Special*, 11(1):13–21, 2019.
[9] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.
[10] C. Cao and M. Li. Generating mobility trajectories with retained data utility. In *KDD*, pages 2610–2620, 2021.
[11] S. Castelo, R. Rampin, A. S. R. Santos, A. Bessa, F. Chirigati, and J. Freire. Auctus: A dataset search engine for data discovery and augmentation. *PVLDB*, 14(12):2791–2794, 2021.
[12] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324. ACM, 2003.
[13] C. Chen, R. Peng, L. Ying, and H. Tong. Network connectivity optimization: Fundamental limits and effective algorithms. In *SIGKDD*, pages 1167–1176, 2018.
[14] L. Chen, G. Cong, X. Cao, and K. Tan. Temporal spatial-keyword top-k publish/subscribe. In *ICDE*, pages 255–266, 2015.
[15] Z. Chen, Y. Wang, V. R. Narasayya, and S. Chaudhuri. Customizable and scalable fuzzy join for big data. *Proc. VLDB Endow.*, 12(12):2106–2117, 2019.
[16] K. Chowdhury, V. V. Meduri, and M. Sarwat. A machine learning-aware data re-partitioning framework for spatial datasets. In *ICDE*, pages 2426–2439, 2022.
[17] R. Cohen and L. Katzir. The generalized maximum coverage problem. *Inf. Process. Lett.*, 108(1):15–22, 2008.
[18] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.
[19] D. Deng, A. Kim, S. Madden, and M. Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. *Proc. VLDB Endow.*, 10(10):1082–1093, 2017.
[20] Y. Deng, C. Chai, L. Cao, Q. Yuan, S. Chen, Y. Yu, Z. Sun, J. Wang, J. Li, Z. Cao, K. Jin, C. Zhang, Y. Jiang, Y. Zhang, Y. Wang, Y. Yuan, G. Wang, and N. Tang. Lakebench: A benchmark for discovering joinable and unionable tables in data lakes. *Proc. VLDB Endow.*, 17(8):1925–1938, 2024.
[21] Y. Dong, K. Takeoka, C. Xiao, and M. Oyamada. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In *ICDE*, pages 456–467, 2021.
[22] Y. Dong, C. Xiao, T. Nozawa, M. Enomoto, and M. Oyamada. Deepjoin: Joinable table discovery with pre-trained language models. *Proc. VLDB Endow.*, 16(10):2458–2470, 2023.
[23] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.
[24] C. Fang, Y. Mei, and S. Song. Matrix factorization with landmarks for spatial data. In *ICDE*, pages 1887–1899, 2023.
[25] R. C. Fernandez, J. Min, D. Nava, and S. Madden. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *ICDE*, pages 1190–1201, 2019.
[26] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.
[27] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
[28] D. He, T. Zhou, X. Zhou, and J. Kim. An efficient algorithm for maximum trajectory coverage query with approximation guarantee. *TITS*, 23(12):24031–24043, 2022.
[29] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, 1998.
[30] D. S. Hochbaum and A. Pathria. Analysis of the greedy approach in problems of maximum k-coverage. *Naval Research Logistics*, 45(6):615–627, 1998.
[31] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007.
[32] G. Kalamatianos, G. J. Fakas, and N. Mamoulis. Proportionality in spatial keyword search. In *ICMD*, pages 885–897, 2021.
[33] S. Khuller, A. Moss, and J. Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1):39–45, 1999.
[34] S. Kim and H. Park. Efficient distributed approximate k-nearest neighbor graph construction by multiway random division forest. In *SIGKDD*, pages 1097–1106, 2023.
[35] P. Li, H. Dai, S. Wang, W. Yang, and G. Yang. Privacy-preserving spatial dataset search in cloud. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pages 1245–1254, 2024.
[36] Q. Liu, Y. Shen, and L. Chen. Lhist: Towards learning multi-dimensional histogram for massive spatial data. In *ICDE*, pages 1188–1199, 2021.
[37] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712, 2013.
[38] S. M. Omohundro. Five balltree construction algorithms. *Science*, 51(1):1–22, 1989.
[39] J. Peng, H. Wang, J. Li, and H. Gao. Set-based similarity search for time series. In *SIGMOD*, pages 2039–2052, 2016.
[40] S. Qi, P. Bouros, and N. Mamoulis. Efficient top-k spatial distance joins. In *SSTD*, pages 1–18, 2013.
[41] S. Qi, P. Bouros, and N. Mamoulis. Top-k spatial distance joins. *GeoInformatica*, 24(3):591–631, 2020.
[42] B. Qiao, B. Hu, J. Zhu, G. Wu, C. G. Giraud-Carrier, and G. Wang. A top-k spatial join querying processing algorithm based on spark. *Inf. Syst.*, 87, 2020.
[43] A. S. R. Santos, A. Bessa, F. Chirigati, C. Musco, and J. Freire. Correlation sketches for approximate join-correlation queries. In *SIGMOD*, pages 1531–1544. ACM, 2021.
[44] A. Shrivastava and P. Li. Asymmetric minwise hashing for indexing binary inner products and set containment. In *WWW*, pages 981–991, 2015.
[45] S. Sieranoja and P. Fränti. Constructing a high-dimensional *k*nn-graph using a z-order curve. *ACM J. Exp. Algorithmics*, 23(1):1–21, 2018.
[46] D. Tsitsigkos, P. Bouros, N. Mamoulis, and M. Terrovitis. Parallel in-memory evaluation of spatial joins. In *SIGSPATIAL*, pages 516–519, 2019.
[47] F. Vandin, E. Upfal, and B. J. Raphael. Algorithms for detecting significantly mutated pathways in cancer. *J. Comput. Biol.*, 18(3):507–522, 2011.
[48] V. V. Vazirani. *Approximation algorithms.* 2001.
[49] T. Vu and A. Eldawy. R*-Grove: Balanced Spatial Partitioning for Large-Scale Datasets. *Frontiers in Big Data*, 3(August), 2020.
[50] J. Wang, G. Li, and J. Feng. Extending string similarity join to tolerant fuzzy token matching. *ACM Trans. Database Syst.*, 39(1):7:1–7:45, 2014.
[51] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, M. Sanderson, and X. Qin. Answering top-k exemplar trajectory queries. In *ICDE*, pages 597–608, 2017.
[52] S. Wang, Y. Sun, C. Musco, and Z. Bao. Public transport planning: When transit network connectivity meets commuting demand. In *SIGMOD*, pages 1906–1919, 2021.
[53] T. Wang, S. Huang, Z. Bao, J. S. Culpepper, and R. Arablouei. Representative routes discovery from massive trajectories. In *SIGKDD*, pages 4059–4069, 2022.
[54] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing. Updatable learned index with precise positions. *PVLDB*, 14(8):1276–1288, 2021.
[55] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.
[56] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15:1–15:41, 2011.
[57] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.
[58] W. Yang, S. Wang, Y. Sun, and Z. Peng. Fast dataset search with earth mover's distance. *PVLDB*, 15(11):2517–2529, 2022.
[59] Z. Yang, B. Zheng, G. Li, X. Zhao, X. Zhou, and C. S. Jensen. Adaptive top-k overlap set similarity joins. In *ICDE*, pages 1081–1092, 2020.
[60] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *ICDE*, pages 34–41, 2015.

[61] J. Yu, J. Wu, and M. Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*, pages 70:1–70:4, 2015.

[62] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers Comput. Sci.*, 10(3):399–417, 2016.

[63] H. Yuan and G. Li. Distributed in-memory trajectory similarity search and join on road network. In *ICDE*, pages 1262–1273, 2019.

[64] E. T. Zacharatou, D. Sidlauskas, F. Tauheed, T. Heinis, and A. Ailamaki. Efficient bundled spatial range queries. In *SIGSPATIAL*, pages 139–148, 2019.

[65] A. Zeakis, D. Skoutas, D. Sacharidis, O. Papapetrou, and M. Koubarakis. Tokenjoin: Efficient filtering for set similarity join with maximum weighted bipartite matching. *Proc. VLDB Endow.*, 16(4):790–802, 2022.

[66] J. Zhang, S. You, and L. Gruenwald. Parallel selectivity estimation for optimizing multidimensional spatial join processing on gpus. In *ICDE*, pages 1591–1598, 2017.

[67] L. Zhang, N. Alghamdi, M. Y. Eltabakh, and E. A. Rundensteiner. TARDIS: distributed indexing framework for big time series data. In *ICDE*, pages 1202–1213. IEEE, 2019.

[68] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *Proc. VLDB Endow.*, 3(1):805–814, 2010.

[69] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: parallelizing spatial join with mapreduce on clusters. In *Cluster*, pages 1–8, 2009.

[70] Y. Zhang and Z. G. Ives. Finding related tables in data lakes for interactive data science. In *SIGMOD*, pages 1951–1966, 2020.

[71] K. Zheng, Y. Zheng, N. J. Yuan, S. Shang, and X. Zhou. Online discovery of gathering patterns over trajectories. *IEEE Trans. Knowl. Data Eng.*, 26(8):1974–1988, 2014.

[72] J. Zhou, A. K. H. Tung, W. Wu, and W. S. Ng. A "semi-lazy" approach to probabilistic path prediction. In *SIGKDD*, pages 748–756. ACM, 2013.

[73] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *SIGMOD*, pages 847–864, 2019.

[74] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller. LSH ensemble: Internet-scale domain search. *PVLDB*, 9(12):1185–1196, 2016.

[75] M. Zhu, D. Papadias, J. Zhang, and D. L. Lee. Top-k spatial joins. *TKDE*, 17(4):567–579, 2005.

IX. Appendix

A. Notations

Frequently used notations are summarized in Table III.

TABLE III
SUMMARY OF NOTATIONS.

Notation	Description
$\mathcal{D}$	a data source consisting a set of spatial datasets
$\mathcal{S}_{\mathcal{D}}$	a collection of cell-based datasets in $\mathcal{D}$
D	a spatial dataset consisting of a set of points
Q, S_Q	a query dataset, a cell-based query dataset
D, S_D	a spatial dataset, a cell-based dataset
n	the number of datasets in $\mathcal{D}$
θ	the resolution in the grid partition
m	the number of data sources
h, w	the height and width of the whole 2D space
μ, ν	the height and width of each cell in the grid
$\mathcal{R}$	the collection of result datasets
2^θ	the number of entries in each dimension
N_Q, N_D	the query node and dataset node
N_{root}, N_{in}	the root node and internal node
$dist(S_D, S_{D'})$	the distance between S_D and $S_{D'}$

B. NP-hardness of *CJSP*

Proof. To prove **CJSP** is NP-hard, we show that any MCP instance can be reduced to an **CJSP** instance in polynomial time. Considering a collection of sets $\mathcal{A} = \{A_1, \ldots, A_{|\mathcal{A}|}\}$ ($\mathcal{U} = \cup_{A_i \in \mathcal{A}} A_i$) and a positive integer k, MCP aims to find a subset $\mathcal{A}^* \subseteq \mathcal{A}$ such that $|\mathcal{A}^*| \leq k$ and $|\cup_{A_i \in \mathcal{A}^*} A_i|$ is maximized. We show that MCP can be viewed as a special case of **CJSP**. Given an arbitrary instance of MCP, we sort the universe $\mathcal{U}$ according to the lexicographic order, then create a mapping from the sorted universe $\mathcal{U} = \{u_1, \ldots, u_{|\mathcal{U}|}\}$ to an integer set $\mathcal{I} = \{0, 1, \ldots, |\mathcal{U}-1|\}$. We can construct a $2^\theta \times 2^\theta$ grid in the space such that $2^\theta \times 2^\theta > |\mathcal{U}|$.

Then, we set the connectivity threshold $\delta = 2^\theta\sqrt{2}$, the query dataset $A_Q = \{0, \ldots, 2^\theta \times 2^\theta - 1\} \backslash \mathcal{I}$. The set $\mathcal{A} \cup \{A_Q\}$ must satisfy the spatial connectivity. This is because $dist(A_i, A_j)(A_i, A_j \in \mathcal{A} \cup \{A_Q\})$ must not exceed δ, ensuring the connectivity constraint is always satisfied. Moreover, it is evident that the total reduction is performed in polynomial time. Thus, the optimal solution for **CJSP** must also be optimal for the corresponding MCP. If **CJSP** is not NP-hard, MCP must not be NP-hard since any instance of MCP can be converted to an instance of **CJSP**. This contradicts the fact that MCP is NP-hard [30]. Therefore, **CJSP** is NP-hard. □

C. Index Update

As described in Section V, our local index is a bidirectional pointer structure, where each dataset node N_D not only contains the pointer of child nodes $N_D.ch$, but also contains the pointer of parent node $N_D.pa$. Thus it can quickly support index updates without rebuilding the entire index. Since DITS-G and DITS-L are similar structures, we take DITS-L as an example to illustrate the index update process, including dataset inserts, updates, and deletes.

Firstly, to insert a new node N_D, we need to add the dataset node to the appropriate leaf node. We first find the node N_{in} with the minimum distance $||N_{in}.o, N_D.o||_2$ in each layer of the tree, and traversal continues down to the child nodes. When the tree node is a leaf node, we insert N_D into the leaf node. If the leaf node capacity is greater than f, we need to split the node according to Algorithm 1. Finally, we iteratively update the parent node information to reflect the properties of the new child nodes accurately.

Secondly, to update the existing dataset node N_D, we can find the original dataset node in the tree according to the unique dataset identifier $N_D.id$. Subsequently, we replace the original node N_D with the updated dataset node N'_D and iteratively update $(rect, o, r, N_l, N_r, pa)$ of the parent node from the bottom up until the parent node contains the child node completely. Finally, for a dataset node N_D that needs to be deleted, we can consider it a special case of a dataset node update. We directly find and remove it from the leaf node and update the information of the parent node.

D. Complexity Analysis

We use n to represent the number of datasets in the data source and m to represent the number of data sources.

Time Complexity Analysis of Constructing DITS. Firstly, we analyze the time complexity of constructing DITS, which comprises the construction of DITS-G and DITS-L. In constructing DITS-L for each data source, Algorithm 1 splits the dataset nodes into two child sets from which trees are recursively built. Consequently, during each split process, it takes $O(n)$ time to split all dataset nodes, and the height of the tree is $\log n$. In addition, DITS-L needs to build the inverted index for each leaf node, resulting in a time complexity of $O(n|S_D|)$, where $|S_D|$ denotes the average size of the cell IDs contained by the dataset node N_D. Thus, the total time complexity of constructing DITS-L is $O(n \log n + n|S_D|)$.

In contrast, the construction process for DITS-G is similar to that of DITS-L, resulting in a time complexity of $O(m \log m)$. Since the number of data sources m and the number of cell IDs in each dataset $|S_D|$ are significantly smaller than n, they can be considered negligible in comparison to n. Thus, the total time complexity of DITS is $O(m \log m + m(n \log n + n|S_D|))) = O(n \log n)$.

Space Complexity Analysis of Constructing DITS. Next, we analyse the time complexity of Algorithm 1 for constructing DITS. The space complexity primarily depends on the number of index nodes in DITS. Firstly, in DITS-L of each data source, the height of the tree is $\log n$, resulting in a total number of index nodes given by the series $1 + 2 + 4 + \cdots + n = 2n$. Thus, the number of index nodes can be approximated as $O(n)$. Similarly, for DITS-G with height $\log m$, the number of index nodes is $O(m)$. Assuming that the space required for each index node is $|size|$, and given that the number of data sources m and $|size|$ are significantly smaller than n, they can be considered negligible in comparison. Thus, the total space complexity of DITS is $O(mn|size| + m|size|) = O(n)$.

Time Complexity Analysis of OverlapSearch. The time complexity of OverlapSearch mainly depends on the index structure and pruning strategy. In the worst-case scenario, the

search may need to traverse all nodes of the tree. Thus, the time complexity of OverlapSearch is $O(n)$. However, due to the pruning strategy, the actual number of nodes accessed is usually much less than n. Thus, the time complexity is much lower than $O(n)$ in the actual situation.

Time Complexity Analysis of CoverageSearch. Here, we show the detailed time complexity analysis of CoverageSearch. Algorithm 3 shows that we iteratively choose the dataset with the maximum marginal gain and satisfy the connectivity. In the worst-case scenario, the algorithm may need to visit n index nodes. Since the algorithm requires k iterations, the total time complexity is $O(kn)$. With the help of the acceleration strategy, the time complexity is much lower than $O(kn)$ in the actual situation.

E. Approximation Guarantee of CoverageSearch

In the following, we provide the approximation guarantee analysis of CoverageSearch, which achieves (1-1/e)-approximation under the condition that at the start of each iteration, the current result set can connect with at least one set in the optimal solution, and the marginal gain brought by the set is above the average value of all the gains brought by the optimal solution. The detailed analysis is as follows.

First, we use S_Q to denote the query set, $\mathcal{S}_\mathcal{D} = \{S_{D_1}, S_{D_2}, \ldots, S_{D_n}\}$ to denote n spatial datasets. Let $\mathcal{U} = (\cup_{i=1}^{n} S_{D_i}) \cup S_Q$ be the set of all set elements, $\mathcal{R} = \{S_Q, S'_{D_1}, \ldots, S'_{D_k}\}$ denote the output of result set after k iterations in our Algorithm 3, $C_i = (\cup_{j=1}^{i} S'_{D_i}) \cup S_Q$ denote the set of elements covered by the end of iteration i, and $V = U - C_{i-1}$ denote the set of elements which are not in C_{i-1}. In addition, let the optimal solution be $\{S_Q, S^*_{D_1}, \ldots, S^*_{D_k}\}$, and $CS_k = \cup_{j=1}^{k} S^*_{D_j}$. Then we have the following Lemma 5.

Lemma 5. *At the start of each iteration i, if the C_{i-1} is connected with at least one optimal set $S^* \in \{S^*_{D_1}, \ldots, S^*_{D_k}\}$, where $|S^* \cap V| \geq \frac{1}{k}\sum_{j=1}^{k} |S^*_{D_j} \cap V|$, then we have*

$$|S'_{D_i} \cap V| = |C_i| - |C_{i-1}| \geq \frac{|OPT| - |C_{i-1}|}{k}. \tag{6}$$

Proof. Let OPT be the union of CS_k and S_Q, i.e., $OPT = CS_k \cup S_Q$, and $|OPT|$ denote the number of elements in OPT. Thus, at the start of iteration i, the number of elements covered in OPT but not in C_{i-1} is

$$|OPT - C_{i-1}| = |OPT| - |OPT \cap C_{i-1}| \geq |OPT| - |C_{i-1}|. \tag{7}$$

Moreover, $|(CS_k \cup S_Q) \cap V| = |OPT - C_{i-1}|$, that is

$$\begin{aligned} |(CS_k \cup S_Q) \cap V| &= |OPT \cap V| \\ &= |OPT \cap (U - C_{i-1})| \\ &= |(OPT \cap U) - (OPT \cap C_{i-1})| \\ &= |OPT| - |(OPT \cap C_{i-1})| \\ &= |OPT - C_{i-1}|. \end{aligned} \tag{8}$$

Therefore, we obtain that

$$|(CS_k \cup S_Q) \cap V| \geq |OPT| - |C_{i-1}|. \tag{9}$$

In addition, we also have

$$\begin{aligned} |(CS_k \cup S_Q) \cap V| &= |(CS_k \cap V) \cup (S_Q \cap V)| \\ &= |(CS_k \cap V)|. \end{aligned} \tag{10}$$

Then we derive that

$$|(CS_k \cap V)| \geq |OPT| - |C_{i-1}|. \tag{11}$$

Dividing the two sides of Inequality (11) by k, we obtain that

$$\begin{aligned} \frac{|(CS_k \cap V)|}{k} &= \frac{|(\cup_{j=1}^{k} S^*_{D_j}) \cap V|}{k} \\ &= \frac{|\cup_{j=1}^{k} (S^*_{D_j} \cap V)|}{k} \\ &\geq \frac{|OPT| - |C_{i-1}|}{k}. \end{aligned} \tag{12}$$

We know that

$$\sum_{j=1}^{k} |S^*_{D_j} \cap V| \geq |\cup_{j=1}^{k} (S^*_{D_j} \cap V)|. \tag{13}$$

By Inequalities 12 and 13, we obtain that

$$\frac{\sum_{j=1}^{k} |S^*_{D_j} \cap V|}{k} \geq \frac{|\cup_{j=1}^{k} (S^*_{D_j} \cap V)|}{k} \geq \frac{|OPT| - |C_{i-1}|}{k}. \tag{14}$$

We know that at least one optimal set $S^* \in \{S^*_{D_1}, \ldots, S^*_{D_k}\}$ is connected with C_{i-1}, where

$$|S^* \cap V| \geq \frac{\sum_{j=1}^{k} |S^*_{D_j} \cap V|}{k}. \tag{15}$$

Combining Iequalities 14 and 15, we obtain that

$$|S^* \cap V| \geq \frac{|OPT| - |C_{i-1}|}{k}. \tag{16}$$

In addition, based on Lines 5 to 10 of Algorithm 3, we find the spatial dataset S'_{D_i} with the maximum marginal gain in the iteration i. That is

$$|S'_{D_i} \cap V| \geq |S^* \cap V|. \tag{17}$$

Thus, by Inequalities 16 and 17, we conclude that: for all $i = \{1, 2, \ldots k\}$,

$$|S'_{D_i} \cap V| = |C_i| - |C_{i-1}| \geq |S^* \cap V| \geq \frac{|OPT| - |C_{i-1}|}{k}. \tag{18}$$

Lemma 6. *For all $i = \{1, 2, \ldots, k\}, |C_i| \geq \frac{|OPT|}{k}\sum_{j=0}^{i-1}(1 - 1/k)^j + (1 - 1/k)^i |S_Q|$.*

Proof. We use induction reasoning to prove the correctness of Lemma 6. For convenience, we let $o = |OPT|/k$ and $t = (1 - 1/k)$. At the start of our algorithm, the set of covered elements can be considered as S_Q, i.e. $C_0 = S_Q$. Firstly, the base case $i = 1$ is trivial as the first choice $|C_1|$ has at least $\frac{|OPT|}{k} + (1-1/k)|S_Q|$ elements by Lemma 5. Then we suppose inductive step i holds, and prove that it holds for $i + 1$. By Lemma 5, we obtain that

$$\begin{aligned} |C_{i+1}| &\geq |C_i| + \frac{|OPT| - |C_i|}{k} \\ &= o + t|C_i| \\ &\geq o + t(o\sum_{j=0}^{i-1} t^j + t^i |S_Q|) \\ &= ot^0 + o\sum_{j=1}^{i} t^j + t^{i+1}|S_Q| \\ &= o\sum_{j=0}^{i} t^j + t^{i+1}|S_Q| \\ &= \frac{|OPT|}{k}\sum_{j=0}^{i}(1 - 1/k)^j + (1 - 1/k)^{i+1}|S_Q|. \end{aligned} \tag{19}$$

The first inequality is by Lemma 5, and the last inequality is from the inductive hypothesis. □

Finally, we have the following theorem and prove it.

Theorem 1. *At the start of each iteration i, if the C_{i-1} is connected with at least one optimal set $S^* \in \{S^*_{D_1}, \ldots, S^*_{D_k}\}$, where $|S^* \cap V| \geq \frac{1}{k}\sum_{j=1}^{k} |S^*_{D_j} \cap V|$, then Algorithm 3 is a (1-1/e)-approximation algorithm for **CJSP**.*

Proof. Based on Lemma 6, we obtain that

$$\begin{aligned} |C_k| &\geq \frac{|OPT|}{k}\sum_{j=0}^{k-1}(1-1/k)^j + (1-1/k)^k|S_Q| \\ &= \frac{|OPT|}{k} \times \frac{1-(1-1/k)^k}{1-(1-(1-1/k))} + (1-1/k)^k|S_Q| \\ &= |OPT|(1-(1-1/k)^k) + (1-1/k)^k|S_Q| \\ &\geq |OPT|(1-(1-1/k)^k). \end{aligned} \tag{20}$$

Moreover, we know that $1+x \leq e^x$ for all $x \in \mathbb{R}$, which means $(1-1/k)^k \leq (e^{-1/k})^k = 1/e$. Then $1-(1-1/k)^k \geq 1-1/e$. Thus, we can derive that

$$|C_k| \geq (1-1/e)|OPT|, \tag{21}$$

which proves that our greedy algorithm provides a (1-1/e)-approximation. □

F. Data Source Details

Here, we present the details of five data sources.

- Baidu-dataset[2] is collected from the Baidu Maps Open Platform, which contains spatial datasets from different industry categories for 28 cities in China.
- BTAA-dataset[3] is collected from the Big Ten Academic Alliance Geoportal, which contains spatial data for the midwestern US, such as Illinois, Indiana, and Michigan.
- NYU-dataset[4] is collected from the NYU Spatial Data Repository, which contains geographic information on multi-subjects like census and transportation worldwide.
- Transit-dataset[5] is collected from Big Geoportal, which contains different kinds of traffic data from Maryland and Washington D.C., such as buses, metro, and waterways.
- UMN-dataset[6] is collected from the data repository of Minnesota University, which contains geographic information from various topics like agriculture and ecology.

G. Additional Experimental Results

Index Update Time Comparison. we conduct experiments to investigate the performance of index updates, including batch inserts and updates of datasets. Specifically, we set the number of updated or inserted dataset nodes $\beta = 100, 150, 200, 250, 300$. We compare the update time of the five indexes with the increase in the number of node updates or insertions. We do not show experiments on the deletion of dataset nodes here, because the deletion of dataset nodes can be seen as a special case of the dataset node updates.

Fig. 21 shows the comparison of index updating time for five indexes as the dataset insertions increase. We observe that the index updating of DITS is slower than STS3 but faster than the others. This is because compared with STS3, DITS needs to update the tree structure according to the dataset insertion strategies described in Appendix IX-C. However, since DITS is a bidirectional pointer structure, it is faster in inserting than QuadTree and Rtree. Additionally, we can also see that Josie's index updating is consistently slower than the other indexes. This is primarily because Josie takes more time in sorting the elements in each set and inserting the dataset into the posting list in sorted order.

Fig. 22 compares the index updating time of five indexes as the number of dataset updates increases. We can observe that the index update time of STS3 is still the fastest due to it can quickly update the dataset ID information in the corresponding posting list by traversing the updated cells. In contrast, Josie is the second-fast. This is because the original dataset has already been sorted according to the dataset ID, and compared to STS3, Josie also needs to update the sorted position information of the cell ID in the dataset. In addition, DITS is ranked as the third, since it requires updating the tree structure in addition to the inverted index information. Similarly, DITS has a faster index update time than Rtree with the help of the bidirectional pointer structure. In contrast, QuadTree has the slowest index update time because it has to repeatedly find the updated cell ID for insertion and deletion.

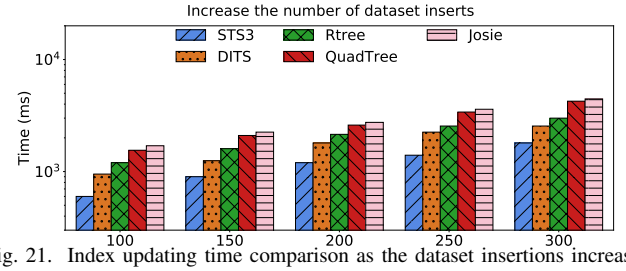


Fig. 21. Index updating time comparison as the dataset insertions increase.

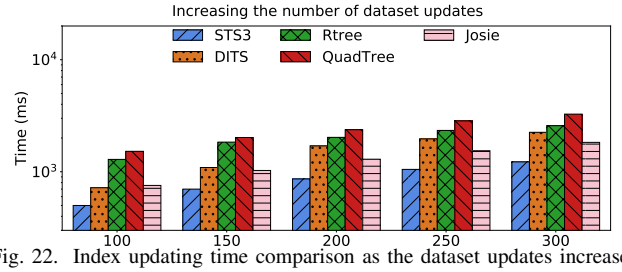


Fig. 22. Index updating time comparison as the dataset updates increase.

[2] https://lbsyun.baidu.com/
[3] https://geo.btaa.org/
[4] https://geo.nyu.edu/
[5] https://geo.btaa.org/
[6] https://conservancy.umn.edu/drum