Received May 20, 2020, accepted June 4, 2020, date of publication June 10, 2020, date of current version June 22, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3001214

Efficient Processing of Spatio-Temporal Joins on IoT Data

KI YONG LEE[1], (Member, IEEE), MINJI SEO[1], RYONG LEE[2], MINWOO PARK[2], AND SANG-HWAN LEE[2]

[1]Department of Computer Science, Sookmyung Women's University, Seoul 04310, South Korea
[2]Scientific Data Research Center, Korea Institute of Science and Technology Information, Daejeon 34141, South Korea

Corresponding author: Ryong Lee (ryonglee@kisti.re.kr)

This work was supported by a Research and Development project, Enabling a System for Sharing and Disseminating Research Data of Korea Institute of Science and Technology (KISTI), South Korea, under Grant K-20-L01-C04-S01.

ABSTRACT As the Internet of Things (IoT) has become widespread, the demand for storing and querying data generated by things (e.g., moving sensors) is growing to obtain more useful information. One of the emerging queries on such IoT data is the *spatio-temporal join*, which joins data generated by different things but generated at (almost) the same time and location. In this paper, we propose an efficient method for processing spatio-temporal joins on IoT data. The proposed method divides the 3D spatio-temporal space into small, equal-sized spaces, called cells. As data is generated by things, the proposed method maintains the information about which thing's data are in which cells. When a spatio-temporal join between specified things is requested, the proposed method first identifies cells, each of which has data of all the specified things within or near it. The proposed method then retrieves only the data within or near the identified cells and performs the join only between the retrieved data. Consequently, compared with previous methods where the processing cost increases rapidly as the size of data or the number of things being joined increases, the processing cost is greatly reduced. The experimental results on a real IoT dataset show that the proposed method significantly reduces the execution time compared with the existing methods.

INDEX TERMS Spatio-temporal join, spatio-temporal index, Internet of Things, IoT data.

I. INTRODUCTION

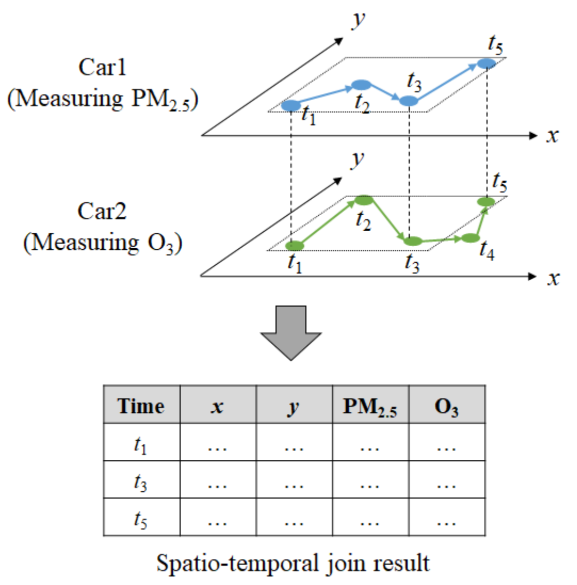
Recently, with the advance of sensor devices, communication technology, and computing power, the Internet of Things (IoT) has become prevalent in many areas including smart homes, smart cities, connected vehicles, and environmental monitoring [1]. As IoT is becoming more widespread, the massive amount of data is being generated by things (i.e., devices connected to the Internet). Along with this, the demand for storing and querying data generated by things is also growing rapidly to obtain more useful information (e.g., the correlation between the measurements of two things). For this reason, there are an increasing number of IoT platforms being developed to provide the ability to store and query data generated by things. For example, AWS IoT [2], Microsoft Azure IoT [3], Oracle IoT [4], Google Cloud IoT [5], and IBM Watson IoT [6] provide the ability to store the vast amount of data generated by things, as well as to query the stored data.

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaolong Li.

In the IoT, each piece of data generated by a thing typically contains information about the time and location at which it was generated. Thus, IoT data, which means data generated by things in the IoT, can be viewed as *spatio-temporal* data [7]. Most current IoT platforms, however, do not provide query processing especially designed or optimized for such spatio-temporal data. Most current IoT platforms use traditional generic SQL or NoSQL query processing (e.g., relations, key-value pairs, wide columns, or documents) to process queries on IoT data. However, this generic approach may not give the best performance for spatio-temporal data, which requires consideration in terms of both time and location aspects.

One of the emerging queries on such IoT data is the *spatio-temporal join*, which joins data generated by different things but generated at (almost) the same time and location [8]. A spatio-temporal join is especially useful when we want to collect data generated by different things but generated at the same time and location.

*Example 1: Suppose two cars Car*1 *and Car*2 *have moved around a city, measuring air quality, i.e., $PM_{2.5}$ and O_3*



Time	x	y	$PM_{2.5}$	O_3
t_1	…	…	…	…
t_3	…	…	…	…
t_5	…	…	…	…

FIGURE 1. Example of a spatio-temporal join.

*respectively. The upper figure in Fig.1 shows the locations and times at which Car*1 *and Car*2 *measured $PM_{2.5}$ and O_3 respectively, where the x- and y-axis represent longitude and latitude, respectively, and $t_1, \ldots, t_5$ represent time points. If we perform a spatio-temporal join between data generated by Car*1 *and Car*2*, the $PM_{2.5}$ values measured by Car*1 *at time t_1, t_3, and t_5 are joined with the O_3 values measured by Car*2 *at t_1, t_3, and t_5, respectively, because those measurements have the same location and time values. Note that the measurements at t_2 and t_4 are not in the join result because they don't have measurements with the same location and time values. These joined measurements can be used to analyze them in an integrated way (e.g., finding a relationship between $PM_{2.5}$ and O_3).*

There has been work on efficient processing of spatio-temporal joins until recently [9], [10]. However, if we apply existing spatio-temporal join algorithms to IoT data, most of them suffer from the following problems. First, the amount of IoT data keeps increasing as time goes by. However, the processing cost of most existing spatio-temporal join algorithms increases directly with increasing size of the input data. Second, for IoT data, a spatial-temporal join can be requested over a number of things. However, the processing cost of most existing algorithms increases rapidly as the number of things involved in the join increases. Third, most current IoT platforms use relational database management systems (RDBMS) or NoSQL as their data storage. However, many spatio-temporal join algorithms are difficult to deploy on existing IoT platforms without modifying existing RDBMS or NoSQL.

Therefore, in this paper, we propose an efficient method for processing spatio-temporal joins on IoT data. The proposed method divides the 3D (x, y, t) spatio-temporal space into small, equal-sized 3D spaces, called *cells*, where x and y are the spatial coordinates and t is the time. If new data are generated by things, the proposed method identifies the cells to which the new data belong (i.e., the cells in which the (x, y, t) values of the new data fall) and the cells near the new data. Then, for each of those cells, the proposed method records the IDs of the things that generated the new data. When a spatio-temporal join between specified things is requested, the proposed method first finds cells, each of which has data of all the specified things within or near it. The proposed method then retrieves only the data within or near the identified cells and performs the join only between the retrieved data. Consequently, the cost of processing a spatio-temporal join is greatly reduced because we can easily identify data close to each other in the 3D space and only need to access them, without accessing unnecessary data with no possibility of being joined.

As a result, the proposed method has three main advantages: (1) Its processing cost does not increase directly with the size of the input data. Rather, its processing cost depends more strongly on the number of cells having data of all the specified things. (2) Even if the number of things involved in the join increases, its processing cost does not increase rapidly, because we can easily identify the cells that have data of all the specified things. (3) It can be easily deployed on top of existing RDBMS and NoSQL without modifying them. It can be easily implemented using the available index structures provided by most RDBMS and NoSQL. Through extensive experiments on a real dataset, we show that the proposed method reduces the execute time significantly compared with the existing representative spatio-temporal join algorithms.

The remainder of the paper is organized as follows: Section II reviews previous work on spatio-temporal joins. In Section III, we formally define our problem and present the proposed spatio-temporal join method in detail. Section IV analyzes the correctness and time complexity of the proposed method. We present our experimental results in Section V and conclude in Section VI.

II. RELATED WORK

A. QUERY PROCESSING IN IoT PLATFORMS

Along with the growth of IoT, many IoT platforms are currently used to connect different devices via the Internet. The basic functionality of an IoT platform is to transfer data between things or between things and servers. However, with the growth of storage capacity and the development of big data technology, many IoT platforms are increasingly providing the ability to store and query data generated by things. For example, most of recently developed IoT platforms, such as AWS IoT [2], Microsoft Azure IoT [3], Oracle IoT [4], Google Cloud IoT [5], IBM Watson IoT [6], SAP Leonardo IoT [11], GE Predix Platform [12], Bosch IoT Suite [13], Siemens MindSphere [14], Cisco IoT [15], and PTC ThingWorx [16], basically provide the ability to transfer data between things and monitor and manage those things.

In addition to this functionality, they provide the ability to store data generated by things in existing RDBMS (e.g., Oracle, MySQL) or NoSQL (e.g., MongoDB, DynamoDB) and query the stored data using SQL or other query languages provided by the underlying storage (i.e., RDBMS or NoSQL). However, most IoT platforms use only traditional generic SQL or NoSQL query processing and provide little functionality especially designed or optimized for spatio-temporal data and queries.

B. SPATIO-TEMPORAL JOINS

Let $R(x, y, t, v_1)$ and $S(x, y, t, v_2)$ be two relations, where x and y are the spatial coordinates, t is the time attribute, and v_1 and v_2 are some value attributes. Given R and S, the spatio-temporal join considered in this paper can be expressed using the SQL syntax as follows:

$$\begin{aligned}&\texttt{SELECT}\ R.x, S.x, R.y, S.y, R.t, S.t, R.v_1, S.v_2\\ &\texttt{FROM}\ R, S\\ &\texttt{WHERE}\ |R.x - Sx| \leq \theta_x\ \texttt{AND}\\ &\qquad\quad |R.y - S.y| \leq \theta_y\ \texttt{AND}\\ &\qquad\quad |R.t - S.t| \leq \theta_t,\end{aligned}$$

where θ_x, θ_y, and θ_t are user-specified thresholds. That is, given two relations R and S, the spatio-temporal join returns all pairs of tuples, one from R and one from S, whose differences in the x, y, and t values are less than or equal to θ_x, θ_y, and θ_t, respectively. The thresholds θ_x, θ_y, and θ_t are used in the spatio-temporal join because data independently generated by different things are difficult to have exactly the same values of the x, y, and t attributes in practice and, in most IoT applications, it is enough if their values are sufficiently close.

Existing algorithms for processing the spatio-temporal join are largely divided into three categories: (1) Non-index algorithms: These algorithms assume that no indexes are available for the input relations (i.e., R and S). If a spatio-temporal join is requested, these algorithms partition each relation on-the-fly into single or multiple levels based on the x, y, and t values using various strategies [17]–[20]. Once the relations are partitioned, each pair of overlapping partitions, one from each relation, is read and joined using an in-memory join algorithm (e.g., the nested-loop join, the plane-sweep algorithm [21], and the Z-order algorithm [22]). Recently, non-index algorithms for big data platforms (i.e., MapReduce, Hadoop, and Spark) have been proposed in [9], [10], [23], [24]. These algorithms perform partitioning and partition-wise joins in parallel on the big data platforms. However, because the amount of IoT data is very large in practice, the cost of partitioning the input data on-the-fly is prohibitive in the IoT environment, especially when spatio-temporal joins are frequently requested.

(2) One-index algorithms: These algorithms use only one index built on either side of the input relations. Most of these algorithms are based on the index nested loop join [8], which uses a spatio-temporal index built on either R or S. A spatio-temporal index is an index allowing efficient access to spatio-temporal data based on their x, y, and t values, such as RT-tree (R-tree with time intervals) [25], MR-tree (multiple R-trees) [25], 3D R-tree (3-dimensional R-tree) [26], HR-tree (historical R-tree) [27], and MV3R-tree (multi-version R-tree) [28]. Suppose among the input relations R and S, only S is indexed with a spatio-temporal index on the x, y, and t attributes. Then, for each tuple r in R, S is searched using the spatio-temporal index to find all tuples in S satisfying the spatio-temporal join predicate with r. Thus, unnecessary access to S is reduced. However, because the search is repeatedly performed for each tuple in R, the processing cost increases directly with increasing size of the input data.

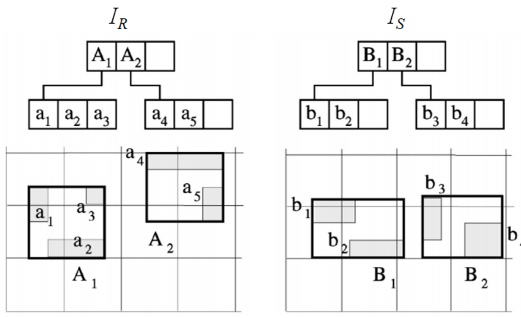


FIGURE 2. Example of a synchronized tree traversal.

(3) Multi-index algorithms: These algorithms assume that all the input relations are indexed with the same type of hierarchical spatio-temporal index, such as 3D R-tree. In this case, a spatio-temporal join can be performed efficiently using a synchronized traversal of the indices [29]–[31]. Suppose both R and S are indexed with 3D R-trees I_R and I_S, respectively. This approach starts from the root nodes of I_R and I_S and synchronously traverses I_R and I_S to the leaf level. If the currently visited node in I_R, say n_R, intersects with the currently visited node in I_S, say n_S (i.e., the minimum bounding boxes of n_R and n_S overlap), then this approach finds pairs of nodes, one from the child nodes of n_R and one from the child nodes of n_S, that overlap each other. This approach then continues to traverse only those overlapping nodes. This process is recursively performed until the leaf nodes are reached. For example, Fig.2 illustrates an example of two indexes I_R and I_S built on R and S, respectively. Starting from the root nodes of I_R and I_S, this approach synchronously traverses to pairs of nodes (A_1, B_1) and (A_2, B_2), respectively. It then traverses only to (a_1, b_1) and (a_2, b_2) and stops because the leaf levels are reached. Finally, the join is performed only between objects in pairs (a_1, b_1) and (a_2, b_2), respectively. Thus, only tuples in R and S that have the possibility of being joined with each other are accessed. However, a problem of this approach is that exhaustive enumeration of all combinations at each level is prohibitive because of their large number [32]. Moreover, the number of the combinations of overlapping nodes can grow extremely large as the number of things being joined increases [33].

In addition to the spatio-temporal join algorithms described above, there have been much work on similar but different problems. Reference [34] has proposed a spatio-temporal join method between time-series grid data and time-series point data. Reference [35] has proposed a join method between trajectories and points to discover sequences of activities from trajectories. Reference [36] has proposed a method for discovering the cascading spatio-temporal patterns, which uses a spatio-temporal join internally. Reference [37] has proposed probabilistic, spatio-temporal joins on symbolic tracking data with uncertainty. Reference [38] has proposed an algorithm for a predictive spatio-temporal join, which finds all pairs of objects that are expected to be close to each other in the future. Reference [39] has proposed an algorithm for processing continuous spatio-temporal joins over spatio-temporal data streams. Reference [40] has developed models for estimating spatio-temporal join selectivity based on probabilistic analysis. However, all these works consider different problems from ours or their definitions of spatio-temporal join are different from ours (e.g., the join predicate, the type of input data).

Compared to the previous spatio-temporal join algorithms, the proposed spatio-temporal algorithm has the following advantages: (1) Unlike non-index algorithms and one-index algorithms, the processing cost does not increase directly with increasing input size. (2) Unlike multi-index algorithms, the processing cost does not increase much even if the number of things involved in the join increases. (3) It can be easily implemented on top of existing RDBMS and NoSQL without modifying their internal implementation. In the following section, we describe the proposed algorithm in detail.

III. PROPOSED METHOD

A. PROBLEM DEFINITION

Let $o_1, o_2, \ldots, o_n$ be n things whose data to be joined. Let $D_i(x, y, t, v_i)$ be the relation that stores data generated by o_i $(i = 1, 2, \ldots, n)$, where x and y represent the location where the tuple is generated (e.g., longitude and latitude), t represents the time when the tuple is generated, and v_i represents the actual data sensed or measured by o_i. Thus, we can think of each D_i as storing the measurements of a moving object over time, where x and y represent the spatial property of the object and t represents the temporal property of the object. Here, v_i can be different types of data for different o_i (e.g., $PM_{2.5}$, O_3, pressure, temperature, etc.). Although we assume that each D_i has only one value attribute v_i, the proposed method can be trivially applied when each D_i has multiple value attributes. For a tuple $d \in D_i$, its x, y, t, and v_i attribute values are denoted by $d.x$, $d.y$, $d.t$, and $d.v_i$, respectively.

Given $D_1, D_2, \ldots, D_n$, where D_i is the relation that stores data generated by o_i $(i = 1, 2, \ldots, n)$ and user-specified thresholds θ_x, θ_y, and θ_t, the spatio-temporal join of $D_1, D_2, \ldots, D_n$, denoted by $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$, is formally defined as follows:

$$
\begin{aligned}
&STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t]) \\
&\quad = \{(d_1, d_2, \ldots, d_n) | \forall i \in \{1, 2, \ldots, n\}(d_i \in D_i) \\
&\qquad \wedge \forall i, j \in \{1, 2, \ldots, n\}(|d_i.x - d_j.x| \leq \theta_x \\
&\qquad \wedge |d_i.y - d_j.y| \leq \theta_y \\
&\qquad \wedge |d_i.t - d_j.t| \leq \theta_t)\}.
\end{aligned}
$$

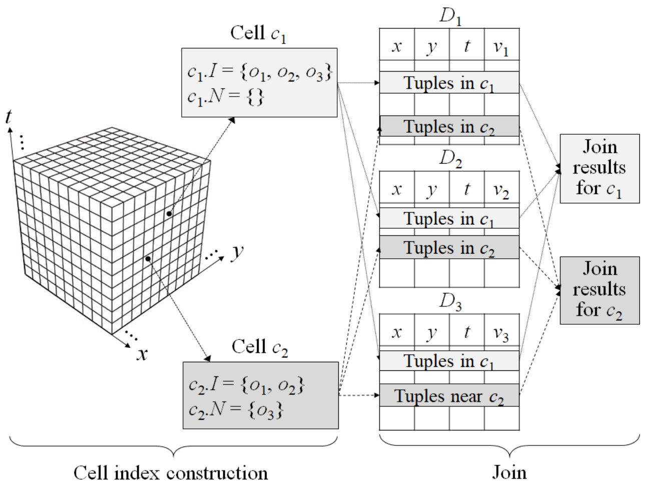


FIGURE 3. Overview of the proposed method.

That is, the spatio-temporal join considered in this paper combines tuples, one from each of $D_1, D_2, \ldots, D_n$, that have the same x, y, and t values within errors of θ_x, θ_y, and θ_t, respectively. The aim of this paper is to develop an efficient method for processing the spatio-temporal join defined above.

Note that we do not use the Euclidean distance to compute the adjacency between tuples. Because the unit of the t attribute values is different from those of the x and y attribute values, to make the problem simple, we compute the adjacency between tuples on each axis.

B. ALGORITHM OVERVIEW

In order to process the spatio-temporal join on IoT data efficiently, we set three goals in designing our proposed method. First, we aim to avoid accessing the whole of any of $D_1, D_2, \ldots, D_n$. This prevents the processing cost from increasing directly with increasing input size. Second, we aim to lower the cost of identifying tuples in $D_1, D_2, \ldots, D_n$ that can be joined together. This prevents the processing cost from increasing rapidly as the number of things being joined increases. Third, we aim to design a method that can be easily implemented using the available features provided by existing RDBMS and NoSQL. This makes the proposed method easy to deploy on existing IoT platforms.

Fig.3 shows the overview of the proposed spatio-temporal join method. Before processing spatio-temporal joins, we conceptually partitions the 3D (x, y, t) spatio-temporal space into small, equal-sized spaces, which we call *cells*, where x and y are the coordinates in the plane and t is the time dimension. We then consider each tuple $d \in D_i$ $(i = 1, 2, \ldots, n)$ as a point in the 3D space whose coordinates are $(d.x, d.y, d.t)$.

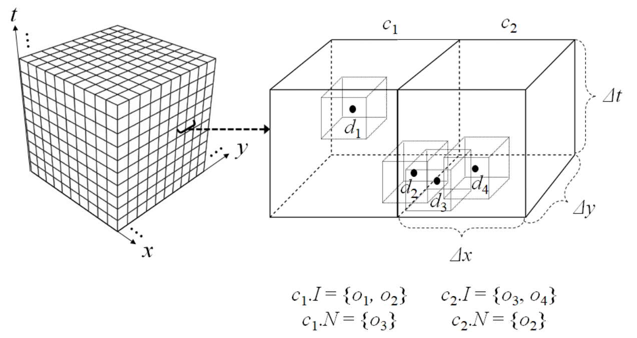


FIGURE 4. Example of cell index construction.

If a new tuple d is generated by o_i and inserted into D_i, we identify the cell c containing d and insert o_i into $c.I$, which represents the set of IDs of things that generated tuples in c. Also, we identify the cells near d (except the cell containing d) and, for each c of those cells, insert o_i into $c.N$, which represents the set of IDs of things that generated tuples near c.

Once this information is constructed to reflect all new tuples, we can process the spatio-temporal join efficiently. If a spatio-temporal join between $D_1, D_2, \ldots, D_n$ is requested, we first find the cells, each c of which has all of $o_1, o_2, \ldots, o_n$ in $c.I \cup c.N$. Then, for each of those cells, we retrieve only those tuples in $D_1, D_2, \ldots, D_n$ that are inside or near that cell. Finally, we perform the join only between the retrieved tuples and output the results. This approach, however, may produce duplicate results, which we will describe in detail in Section III.E. Thus, before outputting the joined tuples, we perform an additional filtering step to eliminate duplicate results.

By using this approach, the proposed method has the following advantages: (1) Because we access only tuples in the cells that have tuples of all of $o_1, o_2, \ldots, o_n$ inside or near them, an increase in the input size does not directly lead to an increase in the processing cost. (2) Even if the number of things being joined increases, the processing cost does not increase rapidly because we can easily find the cells that have tuples of all of $o_1, o_2, \ldots, o_n$ inside or near them by a simple intersection operation, which will be described in detail in Section III.D. (3) The proposed method can be easily implemented on top of existing RDBMS and NoSQL using the existing index structures (i.e., multidimensional index and inverted index) provided by most RDBMS and NoSQL.

The proposed method consists of three main steps: (1) cell index construction, (2) join, and (3) duplicate filtering. In the next subsections, we describe each step in detail.

C. CELL INDEX CONSTRUCTION

As described in the previous section, the proposed method partitions the 3D (x, y, t) spatio-temporal space into equal-sized cells. Let Δx, Δy, and Δt be the size of a cell along the x-, y- and t-axis, respectively. Fig.4 shows an example of two adjacent cells, namely c_1 and c_2. We denote a particular cell as $c[i][j][k]$, where i, j, and k are the x, y, and t indices of the cell, which start from 0, respectively. The space covered by a cell $c[i][j][k]$ is expressed as follows:

$$\begin{aligned} c[i][j][k] = \{(x, y, t) \in \mathbb{R}^3 | \\ & x \in [x_{min} + \Delta x \cdot i,\ x_{min} + \Delta x \cdot (i+1)) \\ & \wedge y \in [y_{min} + \Delta y \cdot i,\ y_{min} + \Delta y \cdot (y+1)) \\ & \wedge t \in [t_{min} + \Delta t \cdot i,\ t_{min} + \Delta t \cdot (t+1))\}, \end{aligned}$$

where $[a, b)$ is a left-closed and right open interval and x_{min}, y_{min}, and t_{min} are the minimum values of x-, y-, and t-axis, respectively. If a cell c has a tuple inside or near it, we maintain the following information for c:

- $c.I$: the set of IDs of things whose tuple d is within c (i.e., $(d.x, d.y, d.t) \in c$).
- $c.N$: the set of IDs of things whose tuple d is outside c but d's *joinable* space overlaps c (i.e., $(d.x, d.y, d.t) \notin c \wedge jsp(d) \cap c \neq \emptyset$).

Here, the joinable space of a tuple d, denoted by $jsp(d)$, is defined as follows:

$$\begin{aligned} jsp(d) = \{(x, y, t) \in \mathbb{R}^3 || x - d.x| \leq \Theta_x \wedge |y - d.y| \\ \leq \Theta_y \wedge |t - d.t| \leq \Theta_t\}, \end{aligned}$$

where Θ_x, Θ_y, and Θ_t are the maximum values of θ_x, θ_y, and θ_t allowed in $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$. That is, the joinable space of a tuple d represents the space where tuples having the possibility of being joined with d can exist.

For example, in Fig. 4, suppose tuples d_1, d_2, d_3, and d_4 are generated by o_1, o_2, o_3, and o_4 and inserted into D_1, D_2, D_3 and D_4, respectively (i.e., $d_1 \in D_1$, $d_2 \in D_2$, $d_3 \in D_3$, and $d_4 \in D_4$). The boxes surrounding d_1, d_2, d_3, and d_4 represent their join spaces, respectively. In Fig. 4, $c_1.I = \{o_1, o_2\}$ because tuples generated by o_1 and o_2 (i.e., d_1 and d_2) are within c_1. Also, $c_1.N = \{o_3\}$ because a tuple generated by o_3 (i.e., d_3) is outside c_1 but its joinable space overlaps c_1. Similarly, $c_2.I = \{o_3, o_4\}$ because tuples generated by o_3 and o_4 (i.e., d_3 and d_4) are within c_2, and $c_2.N = \{o_2\}$ because a tuple generated by o_2 (i.e., d_2) is outside c_2 but its joinable space overlaps c_2. Once $c.I$ and $c.N$ are constructed for all cells that have tuples inside or near them, we can process the spatio-temporal join efficiently. We will describe the processing of a spatio-temporal join in the next subsection.

Now we describe how to construct $c.I$ and $c.N$ for cells that have tuples inside or near them. Initially, $c.I = \emptyset$ and $c.N = \emptyset$ for all cells. Suppose we are given a new tuple d, which was generated by o_i and inserted into D_i. Then we first identify the indices of the cell to which d belongs using the following equations:

$$\begin{aligned} i &= \lfloor (d.x - x_{min})/\Delta x \rfloor \\ j &= \lfloor (d.y - y_{min})/\Delta y \rfloor \\ k &= \lfloor (d.t - t_{min})/\Delta t \rfloor \end{aligned}$$

We then insert o_i into $c[i][j][k].I$. Next we identify the range of the indices of the cells overlapping the joinable space of d, $jsp(d)$, using the following inequality:

$$\lfloor (d.x - x_{min} - \Theta_x)/\Delta x \rfloor \leq i \leq \lfloor (d.x - x_{min} + \Theta_x)/\Delta x \rfloor$$

$$\lfloor (d.y - y_{min} - \Theta_y)/\Delta y \rfloor \leq j \leq \lfloor (d.y - y_{min} + \Theta_y)/\Delta y \rfloor$$
$$\lfloor (d.t - t_{min} - \Theta_t)/\Delta t \rfloor \leq k \leq \lfloor (d.t - t_{min} + \Theta_t)/\Delta t \rfloor$$

Finally, we insert o_i into $c[i][j][k].N$ where i, j, and k are in the above range *except* the cell to which d belongs.

However, note that because $\Theta_x \ll \Delta x$, $\Theta_y \ll \Delta y$, and $\Theta_t \ll \Delta t$ in practice (i.e., the maximum allowable error for the equi-join condition is much smaller than the size of a cell), there are usually no such i, j, and k for which we need to insert o_i into $c[i][j][k].N$. Also note that we need to maintain $c.I$ and $c.N$ only for cells that have tuples inside or near them. In other words, if a cell has no tuples inside and near it, we don't need to maintain any information about that cell.

Algorithm 1 Cell Index Construction

```
Input: a set $\Delta D$ of new tuples inserted into $D_1$, $D_2, \ldots, D_n$
for each $d \in \Delta D$ do
    Let $o_i$ be the ID of the thing that generated $d$
    $i = \lfloor (d_i.x - x_{min})/\Delta x \rfloor$
    $j = \lfloor (d_i.y - y_{min})/\Delta y \rfloor$
    $k = \lfloor (d_i.t - t_{min})/\Delta t \rfloor$
    $c[i][j][k].I = c[i][j][k].I \cup \{o_i\}$

    $i_{min} = \lfloor (d_i.x - x_{min} - \Theta_x)/\Delta x \rfloor$
    $i_{max} = \lfloor (d_i.x - x_{min} + \Theta_x)/\Delta x \rfloor$
    $j_{min} = \lfloor (d_i.y - y_{min} - \Theta_y)/\Delta y \rfloor$
    $j_{max} = \lfloor (d_i.y - y_{min} + \Theta_y)/\Delta y \rfloor$
    $k_{min} = \lfloor (d_i.t - t_{min} - \Theta_t)/\Delta t \rfloor$
    $k_{max} = \lfloor (d_i.t - t_{min} + \Theta_t)/\Delta t \rfloor$
    for $i = i_{min}$ to $i_{max}$ do
        for $j = j_{min}$ to $j_{max}$ do
            for $k = k_{min}$ to $k_{max}$ do
                if ($i \neq \lfloor (d_i.x - x_{min})/\Delta x \rfloor \wedge$
                  $j \neq \lfloor (d_i.y - y_{min})/\Delta y \rfloor \wedge$
                  $k \neq \lfloor (d_i.t - t_{min})/\Delta t \rfloor$) then
                    $c[i][j][k].N = c[i][j][k].N \cup \{o_i\}$
                end
            end
        end
    end
end
```

Algorithm 1 shows the procedure for cell index construction described so far. For each new tuple d, we first identify the cell c to which d belongs (lines 3-5) and insert the ID of the thing that generated d into $c.I$ (line 6). Next we identify the cells, each c of which overlaps the joinable space of d (lines 8-13) and insert the ID of the thing that generated d into $c.N$ except the cell to which d belongs (lines 14-22).

D. JOIN

Once the cell index is updated to reflect all new tuples (i.e., $c.I$ and $c.N$ are updated for all cells that have new tuples inside or near them), we can process a spatio-temporal join efficiently by using the cell index. Suppose a spatio-temporal join is requested over $D_1, D_2, \ldots, D_n$. The proposed method performs the join in three stages: (1) Finding join occurrence cells: We find the cells that have tuples of all of $o_1, o_2, \ldots, o_n$ inside or near them. (2) Retrieving candidate tuples: For each cell found, we retrieve tuples from $D_1, D_2, \ldots, D_n$ that are in or near the cell. (3) Performing joins: For each cell found, we perform a join between the tuples retrieved from D_1, $D_2, \ldots, D_n$ that are in or near the cell. Now we describe each of the three stages in detail.

1) STAGE 1: FINDING JOIN OCCURRENCE CELLS

In this step, we find the cells that have tuples of all of o_1, $o_2, \ldots, o_n$ inside or near them. Since tuples cannot be joined if they are far from each other in the 3D spatio-temporal space, it is sufficient to consider only those cells to obtain the join result. To do this, for each o_i ($i = 1, 2, \ldots, n$), we first find the set C_i of cells, each of which contains o_i in $c.I$ or $c.N$ (i.e., $C_i = \{c | o_i \in c.I \cup c.N\}$). In other words, C_i represents the set of cells that have tuples of o_i inside or near them. This can be performed very efficiently using an inverted index that maps o_i to the list of cells containing o_i in $c.I$ or $c.N$. After obtaining C_i for each o_i ($i = 1, 2, \ldots, n$), we compute $C = C_1 \cap C_2 \cap \ldots \cap C_n$. Here, C represents the set of cells that have tuples of all of $o_1, o_2, \ldots, o_n$ inside or near them. Therefore, in order to perform the join between D_1, $D_2, \ldots, D_n$, we only need to consider the tuples in D_1, $D_2, \ldots, D_n$ that are in or near the cells in C.

2) STAGE 2: RETRIEVING CANDIDATE TUPLES

Once $C = C_1 \cap C_2 \cap \ldots \cap C_n$ is obtained, for each cell $c \in C$, we retrieve the tuples of $o_1, o_2, \ldots, o_n$ that are in or near c. To do this, for each o_i ($i = 1, 2, \ldots, n$), we retrieve the tuples from D_i that are in or near c depending on whether $o_i \in c.I$ or $o_i \in c.N$. If $o_i \in c.I$, then we retrieve the tuples from D_i satisfying the following condition:

$$\delta_i = \{d \in D_i | (d.x, d.y, d.t) \in c\}$$

That is, δ_i represents the tuples of o_i that are in c. δ_i can be expressed as a simple range query on D_i and can be obtained very efficiently using any existing multi-dimensional index (e.g., R-tree) built on the x, y, and t attributes of D_i. Otherwise, if $o_i \in c.N$, then we retrieve the tuples from D_i satisfying the following condition:

$$\delta_i = \{d \in D_i | (d.x, d.y, d.t) \notin c \wedge jsp(d) \cap c \neq \emptyset\}$$

In this case, δ_i represents the tuples of o_i that are near c (i.e., the tuples of o_i that are outside c but whose joinable spaces overlap c). Also in this case, δ_i can be expressed as a simple range query on D_i and can be obtained very efficiently using any multi-dimensional index. In this way, for each cell $c \in C$, we can obtain $\delta_1, \delta_2, \ldots, \delta_n$, which represent the tuples of o_1, $o_2, \ldots, o_n$ that are in or near c, respectively.

3) STAGE 3: PERFORMING JOINS

Once we obtain $\delta_1, \delta_2, \ldots, \delta_n$ for a cell $c \in C$, we perform the join between $\delta_1, \delta_2, \ldots, \delta_n$ (i.e., $STJ([\delta_1, \delta_2, \ldots, \delta_n], [\theta_x, \theta_y, \theta_t])$) to generate join results for c. Because each δ_i ($i = 1, 2, \ldots, n$) is small in practice, we can easily perform $STJ([\delta_1, \delta_2, \ldots, \delta_n], [\theta_x, \theta_y, \theta_t])$ using an existing in-memory join algorithm such as the nested loop join, the hash join, the plane sweep, or the Z-order [33]. In the experiments, we performed $STJ([\delta_1, \delta_2, \ldots, \delta_n], [\theta_x, \theta_y, \theta_t])$ using the join operators provided by the underlying storage (i.e., Oracle DBMS and MongoDB) but any existing in-memory join algorithm can be used. By repeating Stage 2 and 3 for each cell $c \in C$ found in Stage 1, we can obtain the complete result of the requested spatio-temporal join.

However, one problem with the proposed approach described so far is that duplicate join results may be produced from different cells in C. In the next subsection, we explain this problem in detail and describe how to eliminate duplicate results efficiently.

E. DUPLICATE FILTERING

After finding the set C of join occurrence cells in Stage 1, we repeat Stage 2 and 3 for each cell $c \in C$. However, in the process of repeating Stage 2 and 3, duplicate join results may be produced from different $c \in C$. Let us illustrate this problem using the example in Fig.4. Suppose we compute $STJ([D_2, D_3], [\theta_x, \theta_y, \theta_t])$, and d_2 and d_3 satisfy the join condition (i.e., $(d_2, d_3) \in STJ([D_2, D_3], [\theta_x, \theta_y, \theta_t])$). In Stage 1 of the join step, we first find the set of cells that have tuples of o_2 inside or near them, which results in $C_2 = \{c_1, c_2\}$ because $o_2 \in c_1.I$ and $o_2 \in c_2.N$. Similarly, we find the set of cells that have tuples of o_3 inside or near them, which results in $C_3 = \{c_1, c_2\}$ because $o_3 \in c_1.N$ and $o_3 \in c_2.I$. Therefore, the set of cells that have tuples of both o_2 and o_3 inside or near them is $C = C_1 \cap C_2 = \{c_1, c_2\}$. Now suppose we perform Stage 2 and 3 for $c_1 \in C$. Because $o_2 \in c_1.I$, we retrieve the tuples of o_2 in c_1 (i.e., $\delta_2 = \{d_2\}$). Also, because $o_3 \in c_1.N$, we retrieve the tuples of o_3 outside c_1 but whose joinable spaces overlap c_1 (i.e., $\delta_3 = \{d_3\}$). Therefore, the join result produced for c_1 is $STJ([\delta_2, \delta_3], [\theta_x, \theta_y, \theta_t]) = \{(d_2, d_3)\}$. Next, suppose we perform Stage 2 and 3 for $c_2 \in C$. Because $o_2 \in c_2.N$, we retrieve the tuples of o_2 outside c_2 but whose joinable spaces overlap c_2 (i.e., $\delta_2 = \{d_2\}$). Also, because $o_3 \in c_2.I$, we retrieve the tuples of o_3 in c_2 (i.e., $\delta_3 = \{d_3\}$). Therefore, the join result produced for c_2 is $STJ([\delta_2, \delta_3], [\theta_x, \theta_y, \theta_t]) = \{(d_2, d_3)\}$. Consequently, we can see that the same join result (d_2, d_3) is produced twice for c_1 and c_2.

To avoid this redundancy problem, we use the following technique. For a cell $c \in C$, suppose tuples $d_1, d_2, \ldots, d_n$ are joined to produce a joined tuple $(d_1, d_2, \ldots, d_n)$ in Stage 3. Before outputting the joined tuple $(d_1, d_2, \ldots, d_n)$, we additionally perform the following procedure: First, we select a tuple among $d_1, d_2, \ldots, d_n$ in such a way that the same tuple is always uniquely selected. For example, in this paper, we select among $d_1, d_2, \ldots, d_n$ the tuple with the minimum sum of its attribute values. More specifically, for each tuple d_i ($i = 1, 2, \ldots, n$), we first compute the sum of its attribute values (i.e., $d_i.x + d_i.y + d_i.t + d_i.v_i$). Then, we select the tuple with the minimum value of $d_i.x + d_i.y + d_i.t + d_i.v_i$. Let d' be the tuple so selected among $d_1, d_2, \ldots, d_n$. Next, we check if d' is contained in the current cell c. If d' is contained in the current cell c, then we output the joined tuple $(d_1, d_2, \ldots, d_n)$. Otherwise, we discard the joined tuple $(d_1, d_2, \ldots, d_n)$. That is, we output the joined tuple $(d_1, d_2, \ldots, d_n)$ only when the current cell c contains the selected tuple d'. Consequently, the joined tuple $(d_1, d_2, \ldots, d_n)$ is output only for one cell (i.e., the cell containing d') among the cells containing $d_1, d_2, \ldots, d_n$. For example, in Fig.4, suppose d_2 is the tuple with the minimum sum of its attribute values among d_2 and d_3. In this case, the joined tuple (d_2, d_3) is output only for c_1 because d_2 is contained in c_1. On the other hand, (d_2, d_3) is not output for c_2 because d_2 is not contained in c_2.

Algorithm 2 Spatio-Temporal Join

```
Input: Relations $D_1, D_2, \ldots, D_n$,
       Error thresholds $\theta_x, \theta_y, \theta_t$
// Stage 1: Determining join occurrence cells
for $i = 1$ to $n$ do
    $C_i = \{c | o_i \in c.I \cup c.N\}$
end
$C = C_1 \cap C_2 \cap \ldots \cap C_n$

for each $c \in C$ do
    // Stage 2: Retrieving candidate tuples
    for $i = 1$ to $n$ do
        $\delta_i = \emptyset$
        if $o_i \in c.I$ then
            $\delta_i = \delta_i \cup \{d \in D_i | (d.x, d.y, d.t) \in c\}$
        end
        if $o_i \in c.N$ then
            $\delta_i = \delta_i \cup \{d \in D_i | (d.x, d.y, d.t) \notin c \wedge$
                                $jsp(d) \cap c \neq \emptyset\}$
        end
    end

    // Stage 3: Performing joins
    $R = STJ([\delta_1, \delta_2, \ldots, \delta_n], [\theta_x, \theta_y, \theta_t])$
    for each $(d_1, d_2, \ldots, d_n) \in R$ do
        $d'$ = the tuple with the minimum sum of its
               attribute values among $d_1, d_2, \ldots, d_n$
        if $d' \in c$ then
            $output((d_1, d_2, \ldots, d_n))$
        end
    end
end
```

Algorithm 2 provides the procedure for performing a spatio-temporal join described so far. First, for each o_i ($i = 1, 2, \ldots, n$), the proposed algorithm finds the set C_i of cells that have tuples of o_i inside or near them (lines 2-4). It then obtains the set of cells that have tuples of all of $o_1, o_2, \ldots, o_n$ inside or near them by computing the intersection of C_1, C_2, $\ldots C_n$ (line 5). Once such cells are obtained, for each c of those cells, the proposed method performs the following: (1) For each o_i ($i = 1, 2, \ldots, n$), it retrieves the tuples from D_i that are in c if $o_i \in c.I$ (lines 11-13) or retrieves the tuples from D_i that are near c if $o_i \in c.N$ (lines 14-17). (2) Once these tuples are retrieved, it performs the join between the retrieved tuples (line 21). (3) Finally, before outputting joined tuples, it performs the additional redundancy test and outputs the joined tuples only when they pass the test (lines 22-28).

IV. ANALYSIS OF THE PROPOSED METHOD

In this section, we analyze the correctness and time complexity of the proposed method. First, we prove the correctness of the proposed method.

Theorem 1: Given n relations $D_1, D_2, \ldots, D_n$ and user-specified thresholds θ_x, θ_y, and θ_t, the proposed method returns all and only the tuples in $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$.

Proof: Let $(d_1, d_2, \ldots, d_n)$ be a tuple in $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$ (i.e., $(d_1, d_2, \ldots, d_n) \in STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$). For the joined tuple $(d_1, d_2, \ldots, d_n)$, $d_1, d_2, \ldots, d_n$ are either all in the same cell or distributed over two or more (adjacent) cells.

First, suppose $d_1, d_2, \ldots, d_n$ are all in the same cell c. In this case, by Algorithm 1, $c.I$ contains all of $o_1, o_2, \ldots, o_n$ because their tuples $d_1, d_2, \ldots, d_n$ are all in c. When Algorithm 2 is performed, by lines 2-4, each C_i contains c because $o_i \in c.I$ ($i = 1, 2, \ldots, n$) and accordingly $c \in C_1 \cap C_2 \cap \ldots \cap C_n = C$. Thus, Algorithm 2 performs Stage 2 and 3 for c because $c \in C$. In Stage 2, because $o_i \in c.I$ for $i = 1, 2, \ldots, n$, each δ_i contains d_i ($i = 1, 2, \ldots, n$) by lines 11-13. Next, in Stage 3, because $d_1 \in \delta_1$, $d_2 \in \delta_2, \ldots, d_n \in \delta_n$ and d_1, $d_2, \ldots, d_n$ satisfy the join condition, $(d_1, d_2, \ldots, d_n) \in R$ by line 21. Finally, because d', which is one of $d_1, d_2, \ldots, d_n$, in c by assumption, $(d_1, d_2, \ldots, d_n)$ is output as a joined tuple by lines 25-27.

Next, suppose $d_1, d_2, .., d_n$ are distributed over two or more cells. Without loss of generality, assume that d_1 and d_2 are in different cells c_1 and c_2, respectively. Also assume that d_1 is the tuple with the minimum sum of its attribute values among $d_1, d_2, \ldots, d_n$ without loss of generality. In this case, by Algorithm 1, $c_1.I$ contains o_1 because d_1, which is a tuple of o_1, is in c_1. Now we show that $c_1.N$ contains o_2 by Algorithm 1. Consider when d_2 is generated by o_2 and inserted into D_2. After inserting o_2 into $c_2.I$ because d_2 is in c_2, Algorithm 1 inserts o_2 into $c.N$ for each cell c that overlaps $jsp(d_2)$ except c_2. Because $(d_1, d_2, \ldots, d_n) \in STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$ by assumption, d_1 is in the joinable space of d_2 (i.e., $d_1 \in jsp(d_2)$). Since $d_1 \in c_1$, c_1 must overlap $jsp(d_2)$. Therefore, Algorithm 1 inserts o_2 into $c_1.N$. If we apply the same logic to the rest of the tuples d_3, $d_2, \ldots, d_n$, we can show that either $o_i \in c_1.I$ or $o_i \in c_1.N$ for $i = 3, 4, \ldots, n$ depending on whether d_i is in c_1 or a cell other than c_1. When Algorithm 2 is performed, by lines 2-4, each C_i contains c_1 because o_i is contained in $c_1.I$ or $c_1.N$ ($i = 1, 2, \ldots, n$) and accordingly $c_1 \in C_1 \cap C_2 \cap \ldots \cap C_n = C$. Thus, Algorithm 2 performs Stage 2 and 3 for c_1 because $c_1 \in C$. In Stage 2, for $i = 1, 2, \ldots, n$, if $o_i \in c_1.I$, then δ_i contains d_i by line 12. Otherwise, if $o_i \in c_1.N$, then δ_i contains d_i by lines 15-16. Next, in Stage 3, because $d_1 \in \delta_1$, $d_2 \in \delta_2, \ldots, d_n \in \delta_n$ and d_1, $d_2, \ldots, d_n$ satisfy the join condition, $(d_1, d_2, \ldots, d_n) \in R$ by line 21. Finally, because d', which is d_1 by assumption, is in c_1, $(d_1, d_2, \ldots, d_n)$ is output as a joined tuple by lines 25-27. Therefore, we have proved that if a tuple is in $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$, the tuple is output by the proposed method, regardless of whether $d_1, d_2, .., d_n$ are all in the same cell or distributed over two or more cells.

On the other hand, let $(d_1, d_2, \ldots, d_n)$ be a tuple not in $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$ (i.e., $(d_1, d_2, \ldots, d_n) \notin STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$). In this case, we cannot reach line 23 of Algorithm 2 because $(d_1, d_2, \ldots, d_n) \notin STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$ and $STJ([\delta_1, \delta_2, \ldots, \delta_n], [\theta_x, \theta_y, \theta_t]) \subseteq STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$ so $(d_1, d_2, \ldots, d_n) \notin STJ([\delta_1, \delta_2, \ldots, \delta_n], [\theta_x, \theta_y, \theta_t])$. Therefore, if a tuple is not in $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$, the tuple cannot be output by the proposed method.

Consequently, because the proposed algorithm outputs a tuple if and only if the tuple is in $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$, the proposed algorithm outputs $STJ([D_1, D_2, \ldots, D_n], [\theta_x, \theta_y, \theta_t])$ correctly. □

Now we analyze the time complexity of the proposed method. First, we analyze the time complexity of Algorithm 1. Given a tuple d generated by a thing o_i, computing the indices of the cell c to which d belongs can be done in a constant time (lines 3-5). Let N_{cells} be the number of cells where at least one tuple exists. If we maintain an inverted index that maps each thing ID to its associated cells, inserting o_i into $c.I$ takes $O(N_{cells}/n)$ time, where N_{cells}/n represents the average number of cells per thing (line 6). After that, computing the range of indices of the cells overlapping $jsp(d)$ takes a constant time (lines 8-13). Then, for each cell c overlapping $jsp(d)$, inserting o_i into $c.N$ takes $O(N_{cells}/n)$ time if we maintain an inverted index. Because the number of cells overlapping $jsp(d)$ cannot exceed a certain number given the values of Θ_x, Θ_y, and Θ_t, the total time to insert o_i into $c.N$ for all cells overlapping $jsp(d)$ is $O(N_{cells}/n)$ (lines 14-22). Consequently, the time complexity of Algorithm 1 for one new tuple is $O(N_{cells}/n)$. If there are N_{new} new tuples, the total time to update the cell index is roughly $O(N_{new} \cdot N_{cells}/n)$, assuming that N_{new} is small compared to the number of existing tuples.

From now on, we analyze the time complexity of Algorithm 2. For each o_i ($i = 1, 2, \ldots, n$), the time required to obtain C_i using the inverted index is $O(|C_i|)$, where $|C_i|$ is the number of cells in C_i. Thus, obtaining all of $C_1, C_2, \ldots, C_n$ takes $O(\sum_{i=1}^{n} |C_i|)$ time (lines 2-4). After that, computing $C = C_1 \cap C_2 \cap \ldots \cap C_n$ takes $O(|C_1| + |C_2| + \ldots + |C_n|) =$

$O(\sum_{i=1}^{n}|C_i|)$ time, assuming that the inverted index maintains the list of cells for each thing in sorted order (line 5). Hence, Stage 1 takes $O(\sum_{i=1}^{n}|C_i|)$ time. Now consider Stage 2 of Algorithm 2. Given a cell $c \in C$, retrieving δ_i from D_i for a thing o_i takes $O(\log|D_i| + |\delta_i|)$ time if we use a hierarchical index (e.g., R-tree) built on D_i, where $|D_i|$ and $|\delta_i|$ are the number of tuples in D_i and δ_i, respectively. Therefore, obtaining all of $\delta_1, \delta_2, \ldots, \delta_n$ takes $O(\sum_{i=1}^{n}\log|D_i| + \sum_{i=1}^{n}|\delta_i|)$ time (lines 9-18), which corresponds to the time complexity of Stage 2 for a cell $c \in C$. Finally, given $\delta_1, \delta_2, \ldots, \delta_n$, Stage 3 takes $O(|\delta_1| + |\delta_2| + \ldots + |\delta_n|) = O(\sum_{i=1}^{n}|\delta_i|)$ time if we use an in-memory join algorithm (e.g., hash join) between δ_1, $\delta_2, \ldots, \delta_n$. Because Algorithm 2 performs Stage 1 once and Stage 2 and 3 for each cell $c \in C$, the total time of Algorithm 2 is $O(\sum_{i=1}^{n}|C_i| + |C| \cdot (\sum_{i=1}^{n}\log|D_i| + \sum_{i=1}^{n}|\delta_i|))$.

Here, since computing $C = C_1 \cap C_2 \cap \ldots \cap C_n$ can be performed very efficiently (i.e., merging the sorted lists C_1, $C_2, \ldots, C_n$ in memory), the execution time of Algorithm 2 is mainly determined by $|C| \cdot (\sum_{i=1}^{n}\log|D_i| + \sum_{i=1}^{n}|\delta_i|)$, which represents the time to retrieve $\delta_1, \delta_2, \ldots, \delta_n$ from D_1, $D_2, \ldots, D_n$ for each cell $c \in C$. Note that this term is directly proportional to $|C|$, where $|C|$ is the number of cells that have tuples of all of $o_1, o_2, \ldots, o_n$ inside or near them. Because $|C|$ is related to the size of the join result, the execution time of the proposed method depends heavily on the size of the join result rather than other factors (e.g., the input size or the number of things being joined). For example, even when the input size (i.e., $|D_1|, |D_2|, \ldots, |D_n|$) increases, the execution time of the proposed method increases slowly because it depends on $\sum_{i=1}^{n}\log|D_i|$, rather than $\sum_{i=1}^{n}|D_i|$. Similarly, even when the number of things being joined n increases, the execution time of the proposed method does not increase so much because the execution time is increased only by *adding* $|C| \cdot (\log|D_i| + |\delta_i|)$ for each new D_i. The increase in the execution time can be reduced even further because $|C|$ may decrease as n increases. Therefore, we can see that the execution time of the proposed method is not much affected by an increase in the input size or the number of things being joined.

For comparison, we briefly analyze the time complexity of the existing methods. First, let us consider a non-index algorithm. This algorithm first reads all the relations and partitions them, which takes $O(\sum_{i=1}^{n}|D_i|)$ time. Let N_P be the number of partitions per relation. After partitioning, for each group of partitions, one from each relation, it performs the join between the partitions in the group. This takes $O(N_P \cdot \sum_{i=1}^{n}|D_i|/N_P)$ time in total, where $|D_i|/N_P$ is the average size of a partition of D_i. Thus, the total execution time of a non-index algorithm is roughly $O(\sum_{i=1}^{n}|D_i|)$. As a result, the execution time of a non-index algorithm increases directly with increasing input size. Next, let us consider an one-index algorithm, which is based on the index nested loop join. This algorithm first performs an index nested loop join between D_1 and D_2, which takes $O(|D_1| \cdot \log|D_2|)$ time, where $\log|D_2|$ represents the depth of the R-tree built on D_2. It then performs an index nested loop join between the previous join result and the next relation repeatedly for $D_3, D_4, \ldots, D_n$, which takes $O(|T_1| \cdot \log|D_3| + |T_2| \cdot \log|D_4| + \ldots + |T_{n-2}| \cdot \log|D_n|)$ time in total, where $|T_i|$ is the size of the intermediate join result produced in the ith iteration. Thus, the total execution time of an one-index algorithm is roughly $O(|D_1| \cdot \log|D_2| + \sum_{i=3}^{n}|T_{i-2}| \cdot \log|D_i|)$. Accordingly, the execution time of an one-index algorithm increases directly with increasing size of the input data, especially the first relation D_1. Lastly, let us consider a multi-index algorithm, which is based on the synchronized index traversal. This algorithm traverses the R-trees built on $D_1, D_2, \ldots, D_n$ synchronously from the root nodes to the leaf level. If we assume that all the R-trees built on $D_1, D_2, \ldots, D_n$ have the same depth (i.e., $\log|D_1| = \log|D_2| = \ldots = \log|D_n|$), this traversal takes $O(\sum_{i=1}^{\log|D_1|}(f^i)^n)$ time, where f is the average fanout of the R-trees, f^i is the average number of nodes in the ith level of an R-tree, and $(f^i)^n$ is the number of possible combinations of nodes at the ith level of the R-trees, one from each R-tree. Finally, performing the join between tuples retrieved from $D_1, D_2, \ldots, D_n$ takes $O(\sum_{i=1}^{n}|D'_i|)$ time, where $|D'_i|$ is the number of tuples retrieved from D_i. Consequently, the total execution time of a multi-index algorithm is roughly $O(\sum_{i=1}^{\log|D_1|}(f^i)^n + \sum_{i=1}^{n}|D'_i|)$. Thus, the execution time of a multi-index algorithm increases rapidly as the number of relations increases.

V. PERFORMANCE EVALUATION

In this section, we present the performance evaluation results of the proposed method for various experiments.

A. EXPERIMENTAL SETUP

In the experiments, we compared the performance of the proposed method with three representative spatio-temporal join methods:

- **Partition-based method (PBM)**: This method represents a *non-index* algorithm, which uses the strategies proposed in [17]–[20]. It partitions the entire spatio-temporal space into cells using a uniform grid. If a spatio-temporal join is requested over $D_1, D_2, \ldots, D_n$, it partitions each relation into groups, each of which corresponds to tuples belonging to the same cell. Finally, for each cell, all groups of $D_1, D_2, \ldots, D_n$ associated with that cell are taken and compared, i.e., all tuples in the groups are tested for the join condition. We implemented PBM mainly based on [17], which uses a computational geometry based plane-sweep technique to perform a join between groups.
- **Indexed nested loop (INL)**: This method represents an *one-index* algorithm, which is based on the index nested loop join [25]–[28]. If a spatio-temporal join is requested over $D_1, D_2, \ldots, D_n$, it first performs an indexed nested loop join between D_1 and D_2 using the spatio-temporal index built on D_2. Next, it performs an index nested loop join between the previous result and D_3 using the spatio-temporal index built on D_3. It repeats this process

for $D_4, \ldots, D_n$. For this method, we used 3D R-trees as the spatio-temporal indices. To implement INL, we refer to the pseudo-code for an index nested loop join algorithm presented in [8]. Please refer to [8] to see the implementation details.

- **Synchronized tree traversal (STT)**: This method represents a *multi-index* algorithm, which is based on the synchronized index traversal [29]–[31]. If a spatio-temporal join is requested over $D_1, D_2, \ldots, D_n$, it starts from the root nodes of the 3D R-trees built on D_1, $D_2, \ldots, D_n$ and synchronously traverses the 3D R-trees to the leaf level. At each level, it finds nodes, one from each index, that all overlap each other and continues to traverse only those overlapping nodes. To implement STT, we also refer to the pseudo-code for a synchronized tree traversal algorithm presented in [8]. Please refer to [8] to see the implementation details.
- **Proposed method (OUR)**: This method is our proposed method.

To be fair, we may compare the proposed method only with spatio-temporal join methods that use all indexes built on all the relations. However, given the relations all with indexes, we can use any of PBM, INL, and STT to perform a spatio-temporal join between them in principle. Thus, for comprehensiveness, we compared the proposed method with all of PBM, INL, and STT. There is another point worth mentioning. Although there are a number of more recently proposed methods on spatio-temporal joins, all of them are based on PBM, INL, or STT, consider different problems from ours, or their definitions of spatio-temporal joins are different from ours, as we described in Section II.B. Hence, as far as we know, PBM, INL, and STT are still the most representative and best performing methods for exactly the same problem we are dealing with. Thus, we compared the proposed method with PBM, INL, and STT.

The experiments were performed on a PC running Windows 10 with an Intel i7-5820 3.3 GHz CPU, 32 GB DRAM, and 2 TB HDD. We implemented all the methods in Java on top of existing RDBMS and NoSQL, respectively.

As an existing RDBMS, we used Oracle RDBMS, which is a popular storage system adopted by a number of IoT platforms [4], [11]. We created n relations $D_1, D_2, \ldots, D_n$ and implemented the four methods using the facilities provided by Oracle DBMS. For INL, STT, and OUR, we created a 3D R-tree, which is provided by Oracle Spatial [41], on the x, y, and t attributes of each D_i ($i = 1, 2, \ldots, n$). For STT, we used the spatial operator `SDO_JOIN`, which is also provided by Oracle Spatial, to perform spatial joins based on the synchronized tree traversal. However, because the `SDO_JOIN` operator currently supports only two relations, we report the performance of STT on Oracle DBMS only for two relations. To the best of our knowledge, there is currently no well-known RDBMS that supports STT for more than two relations, which makes STT difficult to deploy on existing IoT platforms. Note that we need to modify the internal implementation of Oracle Spatial significantly to make Oracle DBMS support STT for more than two relations. This is because, in order to control the traversal of an R-tree ourselves and access the internal information of an R-tree directly, we need to modify the existing implementation of the R-tree in Oracle Spatial. Similar to this case, some proposed methods can be implemented on existing RDBMS and NoSQL only by modifying the existing implementation of the RDBMS and NoSQL (e.g., the existing index structures, the existing access methods, or the existing query processor). For OUR, we additionally created an inverted index that maps each thing ID to its associated cells.

FIGURE 5. Example of a taxi, the sensors installed on it, and the dashboard system.

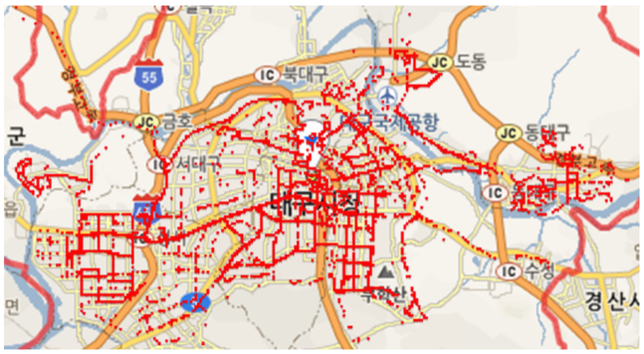


FIGURE 6. Location distribution of the real dataset.

On the other hand, as an existing NoSQL, we used MongoDB, which is another popular storage system used in many IoT platforms, especially for storing and managing large-scale heterogeneous IoT data [42]. In the case of MongoDB, we created n collections $D_1, D_2, \ldots, D_n$, each of which corresponds to a relation in RDBMS. For INL, STT, and OUR, we implemented and created a 3D R-tree on the x, y, and t fields of each D_i ($i = 1, 2, \ldots, n$). Because MongoDB currently does not provide R-tree support, we implemented the R-tree in a similar way to [43]. For OUR, we additionally created an inverted index to efficiently find cells associated with a given thing ID.

For the experiments, we used a real IoT dataset, called Mobile Urban Sensing Dataset, provided by Korea Institute of Science and Technology Information (KISTI) [44]. This dataset contains air quality data measured and collected by a number of taxis driving around Daegu city, South Korea, from September 23 to October 22, 2017. Each taxi, which corresponds to a thing in the IoT, is equipped with a number of sensors measuring air quality including $PM_{1.0}$, $PM_{2.5}$, PM_{10},

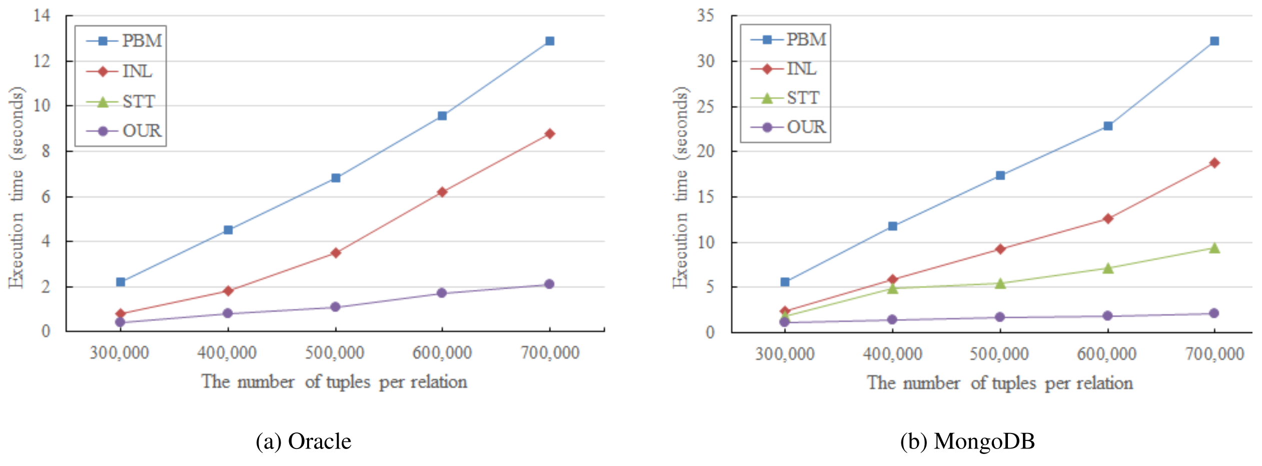


FIGURE 7. Performance evaluation with varying number of tuples per relation.

TABLE 1. Description of the air quality dataset.

Feature	*Description*
Collection period	September 23 to October 22, 2017
The number of taxis	6
Attributes	Taxi ID, latitude, longitude, time, temperature, humidity, $PM_{1.0}$, $PM_{2.5}$, PM_{10}, O_2, O_3, NH_3, CO, NO_2, CH_4, H_2, VOC(volatile organic compound)
The total number of tuples	4,413,348
The average number of tuples per taxi	735,558

TABLE 2. Experimental parameters.

Parameter	*Values*
The number of tuples per relation	300,000, 400,000, **500,000**, 600,000, 700,000
The number of relations	2, 3, **4**, 5, 6
Join thresholds $(\theta_x, \theta_y, \theta_t)$	**(10 m, 10 m, 2 min)**, (20 m, 20 m, 4 min), (30 m, 30 m, 6 min), (40 m, 40 m, 8 min), (50 m, 50 m, 10 min)
Cell size $(\Delta x \times \Delta y \times \Delta t)$	200 m × 200 m × 30 min, 400 m × 400 m × 45 min, **600 m × 600 m × 60 min**, 800 m × 800 m × 75 min, 1,000 m × 1,000 m × 90 min

O_2, O_3, NH_3, CO, NO_2, CH_4, H_2, VOC, temperature, and humidity. While driving, each taxi periodically measures air quality, generates tuples containing the measurements, and sends them to the server via cellular networks. Fig. 5 shows an example of a taxi and the sensors installed on the taxi, along with the dashboard system showing the most recent measurements of each taxi. Fig. 6 is a map showing the location distribution of the collected air quality data (red spots). Table 1 presents the description of the real dataset. In Mobile Urban Sensing Dataset, each tuple contains 17 values, which correspond to values of the 17 attributes in Table 1. For experiments, we first created six relations $D_1, D_2, \ldots, D_6$, each of which is used to store data generated by a particular taxi and has the 17 attributes in Table 1. Then, for each relation D_i $(i = 1, 2, \ldots, 6)$, we extracted tuples with Taxi ID attribute value i from the dataset, split each of them into 17 values, and inserted them into D_i. Note that the longitude, latitude, and time attributes in the dataset correspond to the x, y, and t attributes in D_i, respectively.

In the experiments, we evaluated the performance of the compared methods by varying four parameters, with their default values given in bold in Table 2. When we varied one parameter, we fixed the other parameters to their default values.

- **The number of tuples per relation**: We varied the number of tuples per relation from 300,000 to 700,000 to examine how the performance of the methods changes as the input size increases.
- **The number of relations**: We varied the number of relations being joined from 2 to 6 to investigate the performance characteristics of the methods as the number of relations increases.
- **Join thresholds**: We also varied the join thresholds $(\theta_x, \theta_y, \theta_t)$ from (10 m, 10 m, 2 min) to (50 m, 50 m, 10 min) to study the performance behavior of the compared methods.
- **Cell size**: In the proposed method, the cell size determines the number of tuples accessed during join processing. To observe how the size of cells affects the performance of the proposed method, we varied the size of a cell $\Delta x \times \Delta y \times \Delta t$ from 200 m × 200 m × 30 min to 1,000 m × 1,000 m × 90 min.

As the performance measure, we used the total execution time taken to process a given spatio-temporal join. Note that we excluded the index construction time (i.e., the time to construct R-trees and the cell index) from the execution time because once the indices are constructed, they are used repeatedly for subsequent queries. However, we separately report the cell index construction time in the proposed method in Section V.F. For PBM, we divided each of x-, y-, and t-axis into 10 equal length intervals and partitioned each relation into $10 \times 10 \times 10 = 1,000$ groups. Finally, for OUR, we set the maximum allowed values of θ_x, θ_y, and θ_t to $\Theta_x = 100$ m, $\Theta_y = 100$ m, and $\Theta_t = 20$ min, respectively.

B. VARYING THE SIZE OF RELATIONS

Figure 7 shows the execution time of all the methods when we varied the number of tuples per relation from 300,000 to 700,000. In this experiment, we performed a spatio-temporal join over 4 relations with the join thresholds $\theta_x = 10$ m, $\theta_y = 10$ m, and $\theta_t = 2$ min. For the proposed method (OUR), we set the cell size to 600 m $\times$ 600 m $\times$ 60 min. In Figures 7, 8, …, 11, the left figures (i.e., Figures 7(a), 8(a), …11(a)) show the performance of the methods implemented on top of Oracle DBMS, whereas the right figures (i.e., Figures 7(b), 8(b), …, 11(b)) show the performance of the methods implemented on top of MongoDB.

In Figure 7(a) and (b), PBM shows the worst performance among the compared methods because it does not use indices and reads all tuples in the relations to partition them on-the-fly when a join is requested. Its execution time thus increases fast directly with increasing size of relations. Among the index-based methods, INL has the longest execution time. Because INL needs to repeatedly perform index searches for each tuple in the inner relations, its execution time also increases fast with increasing size of relations. In comparison, STT shows better performance than INL in Figure 7(b). (Note that we do not report the performance of SST on Oracle DBMS in Figure 7(a) for the reason described in Section V.A.) Since STT traverses the indices built on all the relations synchronously and accesses only those tuples in their overlapping leaf nodes, STT accesses fewer tuples than INL. However, STT must enumerate all possible combinations of nodes at each level of the indices and check every combination to see if their bounding boxes overlap. On the contrary, OUR can easily identify join occurrence cells by a simple intersection operation and accesses only those tuples that are in or near those cells. As a result, as shown in Figure 7, OUR outperforms the other methods significantly. More specifically, compared with PBM and INL, OUR reduces the execution time by up to 83% and 76%, respectively, in Figure 7(a). Also in Figure 7(b), OUR reduces the execution time by up to 93%, 89%, and 78% compared with PBM, INL, and STT, respectively. Note also that the execution time of OUR increases slowly compared with the other methods, because its execution time depends on $\sum_{i=1}^{n} \log |D_i|$ as analyzed in Section IV.

C. VARYING THE NUMBER OF RELATIONS

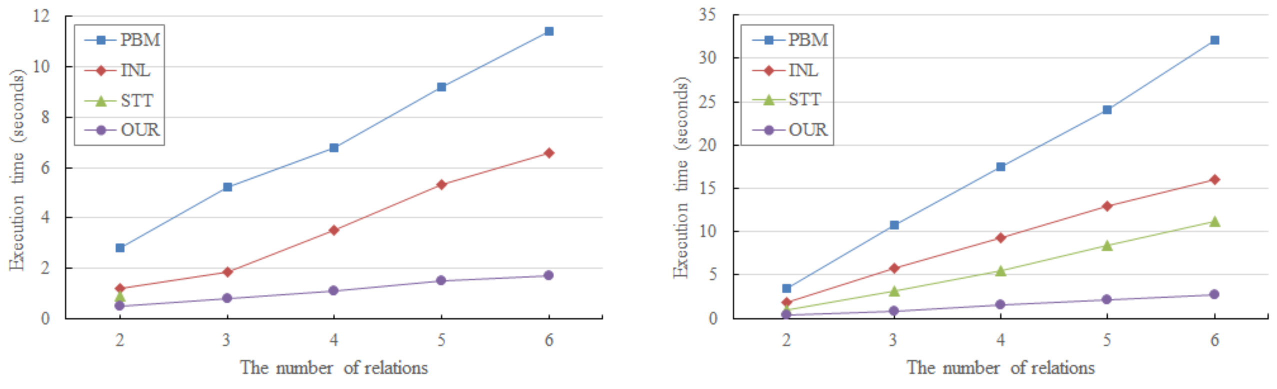
Figure 8 shows the execution time of all the methods when we increased the number of relations in the join from 2 to 6. In this experiment, we set the number of tuples per relation to 500,000 and the join thresholds to $\theta_x = 10$ m, $\theta_y = 10$ m, and $\theta_t = 2$ min. For OUR, the cell size was set to 600 m $\times$ 600 m $\times$ 60 min.

Also in this experiment, OUR shows the best performance, STT and INL are the next, and PBM shows the worst performance. As we can see in Figure 8(a) and (b), OUR outperforms the other methods significantly regardless of the number of relations. The execution time of PBM increases faster than the other methods because an increase in the number of relations leads directly to an increase in the input size. Compared with PBM, the execution time of INL increases slowly. This is because INL uses the intermediate join result between two relations as the inner relation in the next join and the size of the intermediate join result tends to decrease as we proceed to more relations. SST performs better than INL in Figure 8(b), but note that their performance gap is much smaller than that in Figure 7(b). (Note also that we report the performance of SST on Oracle DBMS only for two relations in Figure 8(a) for the same reason described in Section V.A.) In STT, as the number of relations increases, the number of possible combinations of nodes at each level of the indices and the number of overlapping nodes grow rapidly. As a result, this causes a relatively larger increase in its execution time. The performance of OUR is also affected by the number of relations because the cost of retrieving candidate tuples from the relations (i.e., $|C| \cdot (\sum_{i=1}^{n} \log |D_i| + \sum_{i=1}^{n} |\delta_i|)$) increases. However, also in this case, the execution time of OUR does not increase much compared to the other methods because the cost of retrieving candidates tuples increases only by $|C| \cdot (\log |D_i| + |\delta_i|)$ for each new D_i. Consequently, OUR shows consistently better performance than the other methods.

D. VARYING THE JOIN THRESHOLDS

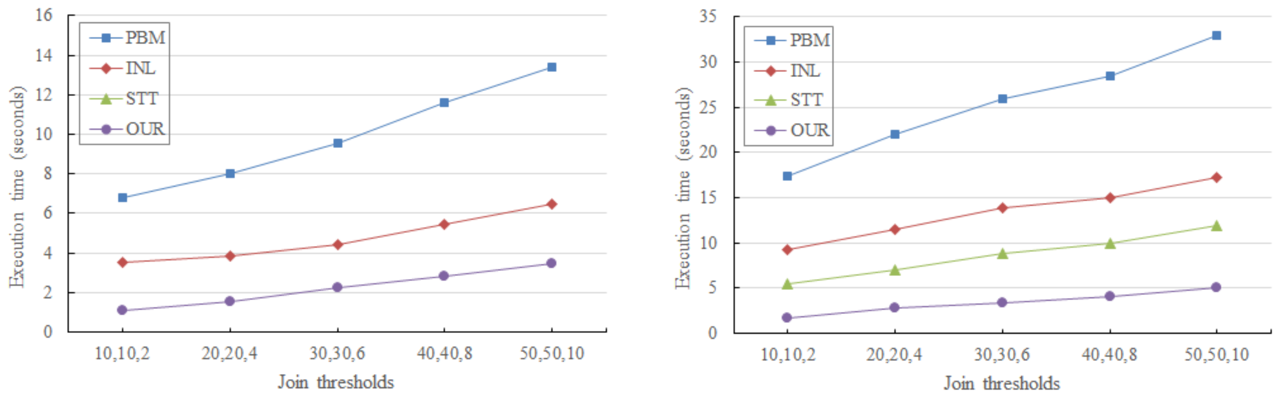
Figure 9 shows the execution time of all the methods when we varied the join thresholds from $\theta_x = 10$ m, $\theta_y = 10$ m, and $\theta_t = 2$ min to $\theta_x = 50$ m, $\theta_y = 50$ m, and $\theta_t = 10$ min. Here, we set the number of relations in the join to 4 and the number of tuples per relation to 500,000. The cell size for OUR was set to 600 m $\times$ 600 m $\times$ 60 min.

We can see that the execution time of all the methods increases as the join thresholds increase. This is because the size of the join results (i.e., the number of output tuples) increases as the join thresholds increase. More specifically, in PBM, the execution time increases because the cost of writing output tuples increases. Similarly, the execution time of INL increases because the cost of writing the intermediate and final join results increases. The execution time of STT also increases because the number of visited nodes in the indices and the cost of writing output tuples increase. Also in OUR, the cost of retrieving candidate tuples from the relations and writing output tuples increases. However, as we can see in Figure 9, the execution time of any method does not increase particularly fast or slowly as the join thresholds increase. Again in this experiment, OUR consistently outperforms the other methods for all the join threshold values.



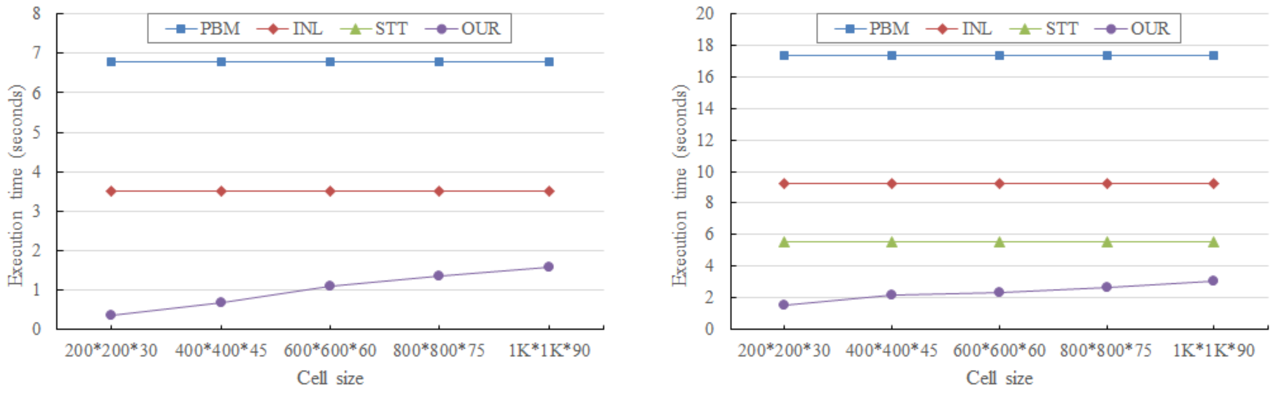
(a) Oracle (b) MongoDB

FIGURE 8. Performance evaluation with varying number of relations.



(a) Oracle (b) MongoDB

FIGURE 9. Performance evaluation with varying join threshold values.

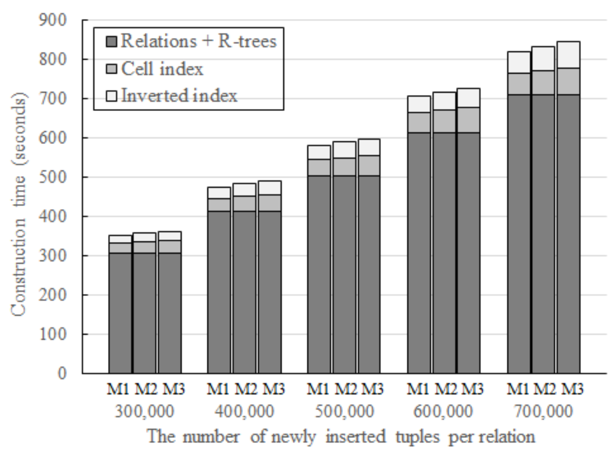


(a) Oracle (b) MongoDB

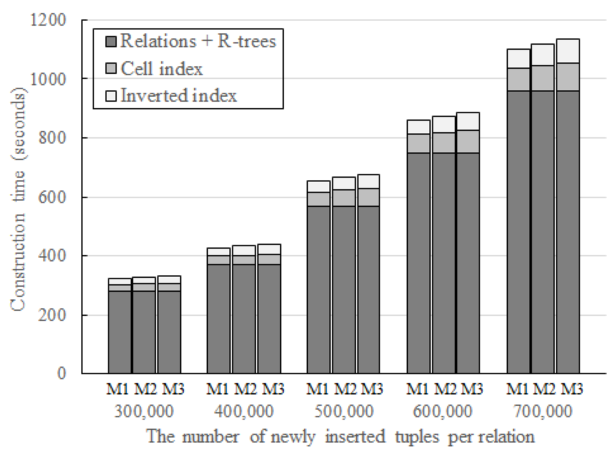
FIGURE 10. Performance evaluation with varying cell sizes.

E. VARYING THE CELL SIZE

The proposed method partitions the 3D space into cells and finds the join occurrence cells when a spatio-temporal join is requested. Thus, the performance of OUR is affected by the cell size because the number of join occurrence cells and accordingly the number of candidate tuples retrieved from the relations vary. Figure 10 shows the execution time of all the methods when we varied the cell size. Note that only OUR is



(a) Oracle



(b) MongoDB

FIGURE 11. **Data insertion and index construction time in the proposed method.**

affected by the cell size because the other methods do not use the cell index. However, we present the execution time of the other methods for comparison only.

In this experiment, we set the number of relations in the join to 4 and the number of tuples per relation to 500,000. The join thresholds were set to $\theta_x = 10$ m, $\theta_y = 10$ m, and $\theta_t = 2$ min. As we can see in Figure 10(a) and (b), the execution time of OUR increases slightly as the cell size increases from 200 m $\times$ 200 m $\times$ 30 min to 1,000 m $\times$ 1,000 m $\times$ 90 min. In general, as the cell size increases, the number of join occurrence cells decreases because each cell occupies more space. However, because the number of tuples in each cell increases, the total number of retrieved tuples increases. On the other hand, if the cell size decreases, although the number of join occurrence cells increases, the total number of retrieved tuples decreases. This is because the spaces where tuples are joined are more precisely identified. As a result, the execution time of OUR increases slightly as the cell size increases, because more candidate tuples are retrieved from the relations. However, in Figure 10, we can observe that OUR always shows better performance than the other methods even when the cell size increases up to 1,000 m $\times$ 1,000 $\times$ 90 min. Obviously, the performance of OUR is expected to degrade if the cell size is excessively large, because too many candidate tuples will be retrieved from the relations. On the other hand, if the cell size is extremely small, we can also expect that the performance of OUR will deteriorate because the number of cells to process becomes too large. In general, we recommend setting the cell size so that all tuples within or near n cells fit easily in memory, where n is the maximum number of relations being joined. This allows $STJ([\delta_1, \delta_2, \ldots, \delta_n], [\theta_x, \theta_y, \theta_t])$ to always be performed in memory. The size of all tuples within or near n cells can be easily obtained from sample data.

F. INDEX CONSTRUCTION TIME

In the proposed method, the cell index must be constructed in advance, before processing a spatio-temporal join. If the cell index takes too long to construct, the usability of the proposed method will be very limited. In the final experiment, we measured the time taken to construct all the indexes in the proposed method (i.e., the cell index, an inverted index, and R-trees). Figure 11 shows the time taken to insert a set of new tuples, including the time taken to construct all the indexes. Here, we set the number of relations to 6, and we increased the number of newly inserted tuples per relation from 300,000 to 700,000 (thus, a total of 1,800,000 to 4,200,000 tuples). The cell size was set to 600 m $\times$ 600 m $\times$ 60 min. Also, because the time taken to construct the cell index is affected by Θ_x, Θ_y, and Θ_t, we varied Θ_x, Θ_y, and Θ_t for each fixed number of newly inserted tuples. In Figure 11, M1 represents when $\Theta_x = 50$ m, $\Theta_y = 50$ m, and $\Theta_t = 10$ min, M2 represents when $\Theta_x = 100$ m, $\Theta_y = 100$ m, and $\Theta_t = 20$ min, and M3 represents when $\Theta_x = 150$ m, $\Theta_y = 150$ m, and $\Theta_t = 30$ min. Also in Figure 11, the dark gray portion of bars (‘Relations + R-trees’) represents the time taken to insert all the new tuples into the relations, including the time taken to construct R-trees on the relations. Thus, this portion is equal to the data insertion and index construction time in INL and STT. On the other hand, the light gray portion of bars (‘Cell index’) and the white portion of bars (‘Inverted index’) represent the time taken to construct the cell index and inverted index in OUR, respectively. Thus, these portions represent the additional cost incurred in OUR to construct the cell and inverted indexes additionally.

As we can see in Figure 11, the data insertion and index construction time in OUR is increased by about 16-17% compared to INL and STT due to the additional indexes (i.e., the cell and inverted indexes). However, even when the number of new tuples per relation is 700,000, the total time to construct the cell and inverted indexes does not exceed 3 minutes in both Figure 11(a) and (b). In the case of our real dataset, the total number of tuples arriving each day is about 150,000 on average. Considering the fact that the cell index only needs to be updated once to reflect a given set of new tuples, this level of execution time is not expensive and acceptable in most real applications. For example, we can update the cell index once every hour or shorter periods to reflect the new tuples that have arrived in the meantime. Furthermore, we can see that the time to construct the cell and inverted indexes increases almost linearly as the number of new tuples increases. Note that this result is consistent with our analysis in Section IV.

Finally, we can see that the time taken to construct the cell and inverted indexes increases slightly as Θ_x, Θ_y, and Θ_t increase, for a fixed number of new tuples. This is because when a tuple d generated by o_i is inserted, the joinable space of d increases so o_i can be inserted into $c.N$ of more cells. However, because such cases do not occur very often, the time taken to construct the cell and inverted indexes does not increase significantly. Hence, we can set Θ_x, Θ_y, and Θ_t to values that are sufficiently larger than the expected values of θ_x, θ_y, and θ_t. Note that because the user can freely set the values of θ_x, θ_y, and θ_t as desired by the application, the expected values of θ_x, θ_y, and θ_t are application dependent, not system dependent.

VI. CONCLUSION

In this paper, we propose a new method for processing spatio-temporal joins on IoT data. Compared with the previous spatio-temporal join methods, the proposed method has the following advantages, especially in the IoT environment: (1) Even when the size of IoT data increases, its processing cost does not increase directly. (2) Even if the number of things being joined increases, its processing cost does not increase rapidly. (3) It can be easily implemented on top of existing IoT storage (i.e., RDBMS and NoSQL) without modifying their internal implementation.

To achieve these advantages, the proposed method partitions the 3D spatio-temporal space into equal-sized cells and maintains the information about which thing's tuples are in which cells. When a spatio-temporal join is requested, the proposed method first identifies cells that have tuples of all the specified things inside or near them and retrieves only those tuples that are inside or near the identified cells. It then performs the join only between those retrieved tuples. As a result, the processing cost is greatly reduced because only tuples close to each other in the spatio-temporal space are efficiently accessed. In addition, the proposed method eliminates duplicate results by performing a simple filtering step before outputting the results.

We also provide a theoretical analysis of the correctness and time complexity of the proposed method. Finally, through the extensive experiments on a real IoT dataset, we show that the proposed method outperforms the existing methods significantly in terms of the execution time. From these results, we can conclude that the proposed method is more efficient and scalable than the existing methods, especially for performing spatio-temporal joins on IoT data. As future work, we are investigating how to efficiently support continuous spatio-temporal joins over continuous IoT data streams.

REFERENCES

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[2] (2020). *AWS IoT*. [Online]. Available: https://aws.amazon.com/iot/

[3] (2020). *Microsoft Azure IoT*. [Online]. Available: https://azure.microsoft.com/overview/iot

[4] (2020). *Oracle IoT*. [Online]. Available: https://www.oracle.com/internet-of-things/

[5] (2020). *Google Cloud IoT*. [Online]. Available: https://cloud.google.com/solutions/iot

[6] (2020). *IBM Watson IoT*. [Online]. Available: https://www.ibm.com/internet-of-things

[7] N. Jiang, Y. Deng, X. Kang, and A. Nallanathan, "Random access analysis for massive IoT networks under a new spatio-temporal model: A stochastic geometry approach," *IEEE Trans. Commun.*, vol. 66, no. 11, pp. 5788–5803, Nov. 2018.

[8] S.-H. Jeong, N. W. Paton, A. A. A. Fernandes, and T. Griffiths, "An experimental performance evaluation of spatio-temporal join strategies," *Trans. GIS*, vol. 9, no. 2, pp. 129–156, Mar. 2005.

[9] L. Alarabi, M. F. Mokbel, and M. Musleh, "ST-Hadoop: A MapReduce framework for spatio-temporal data," *GeoInformatica*, vol. 22, no. 4, pp. 785–813, Oct. 2018.

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

[11] (2020). *SAP Leonardo IoT*. [Online]. Available: https://www.sap.com/products/leonardo-iot-data-services.html

[12] (2020). *GE Predix Platform*. [Online]. Available: https://www.ge.com/digital/iiot-platform

[13] (2020). *Bosch IoT Suite*. (2020). [Online]. Available: https://www.bosch-iot-suite.com/

[14] (2020). *Siemens MindSphere*. [Online]. Available: https://siemens.mindsphere.io/

[15] (2020). *Cisco IoT*. [Online]. Available: https://www.cisco.com/c/en/us/solutions/internet-of-things/overview.html

[16] (2020). *PTC ThingWorx*. [Online]. Available: https://www.ptc.com/products/iiot/thingworx-platform

[17] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," *ACM SIGMOD Rec.*, vol. 25, no. 2, pp. 259–270, Jun. 1996.

[18] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *Proc. VLDB*, Aug. 1998, pp. 570–581.

[19] E. H. Jacox and H. Samet, "Iterative spatial join," *ACM Trans. Database Syst.*, vol. 28, no. 3, pp. 230–256, Sep. 2003.

[20] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki, "TOUCH: In-memory spatial join by hierarchical data-oriented partitioning," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2013, pp. 701–712.

[21] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York, NY, USA: Springer-Verlag, 1985.

[22] J. A. Orenstein, "Spatial query processing in an object-oriented database system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1986, pp. 326–336.

[23] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop-GIS: A high performance spatial data warehousing system over MapReduce," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[24] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "SJMR: Parallelizing spatial join with MapReduce on clusters," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, Aug./Sep. 2009, pp. 1–8.

[25] X. Xu, J. Han, and W. Lu, "RT-tree: An improved R-tree indexing structure for temporal spatial databases," in *Proc. Int. Symp. Spatial Data Handling*, Jul. 1990, pp. 1040–1049.

[26] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-temporal indexing for large multimedia applications," in *Proc. 3rd IEEE Int. Conf. Multimedia Comput. Syst.*, Jun. 1996, pp. 441–448.

[27] M. A. Nascimento and J. R. O. Silva, "Towards historical R-trees," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 1998, pp. 235–240.

[28] Y. Tao and D. Papadias, "MV3R-Tree: A spatio-temporal access method for timestamp and interval queries," in *Proc. Int. Conf. Very Large Data Bases*, Sep. 2001, pp. 431–440.

[29] O. Gunther, "Efficient computation of spatial joins," in *Proc. IEEE 9th Int. Conf. Data Eng.*, Apr. 1993, pp. 50–59.

[30] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1993, pp. 237–246.
[31] Y.-W. Huang, N. Jing, and E. A. Rundensteiner, "Spatial joins using R-trees: Breadth-first traversal with global optimizations," in *Proc. 23rd Int. Conf. Very Large Data Bases*, vol. 1997, pp. 396–405.
[32] N. Mamoulis and D. Papadias, "Multiway spatial joins," *ACM Trans. Database Syst.*, vol. 26, no. 4, pp. 424–475, Dec. 2001.
[33] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Trans. Database Syst.*, vol. 32, no. 1, pp. 7–45, 2007.
[34] H. Hayashi, A. Asahara, N. Sugaya, Y. Ogawa, and H. Tomita, "Spatio-temporal join technique for disaster estimation in large-scale natural disaster," in *Proc. 6th ACM SIGSPATIAL Int. Workshop GeoStreaming (IWGS)*, 2015, pp. 49–58.
[35] K. Xie, K. Deng, and X. Zhou, "From trajectories to activities: A spatio-temporal join approach," in *Proc. Int. Workshop Location Based Social Netw.*, 2009, pp. 25–32.
[36] P. Mohan, S. Shekhar, J. A. Shine, and J. P. Rogers, "Cascading spatio-temporal pattern discovery," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 11, pp. 1977–1992, Nov. 2012.
[37] H. Lu, B. Yang, and C. S. Jensen, "Spatio-temporal joins on symbolic indoor tracking data," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 816–827.
[38] W.-S. Han, J. Kim, B. Suk Lee, Y. Tao, R. Rantzau, and V. Markl, "Cost-based predictive spatiotemporal join," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 2, pp. 220–233, Feb. 2009.
[39] M. F. Mokbel and W. G. Aref, "SOLE: Scalable on-line execution of continuous queries on spatio-temporal data streams," *VLDB J.*, vol. 17, no. 5, pp. 971–995, Aug. 2008.
[40] J. Sun, Y. Tao, D. Papadias, and G. Kollios, "Spatio-temporal join selectivity," *Inf. Syst.*, vol. 31, no. 8, pp. 793–813, Dec. 2006.
[41] (2020). *Oracle Spatial Graph*. [Online]. Available: https://www.oracle.com/database/technologies/spatialandgraph.html
[42] (2020). *The Internet Things | MongoDB*. [Online]. Available: https://www.mongodb.com/use-cases/internet-of-things
[43] L. Xiang, J. Huang, X. Shao, and D. Wang, "A mongodb-based management of planar spatial data with a flattened R-tree," *ISPRS Int. J. Geo-Inf.*, vol. 5, no. 7, pp. 119–135, 2016.
[44] (Oct. 2017). *Mobile Urban Sensing Dataset*. KISTI. [Online]. Available: http://220.123.184.109:8080/KISTI_Web/

KI YONG LEE (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 1998, 2000, and 2006, respectively. From 2006 to 2008, he worked for Samsung Electronics Company Ltd., Suwon, South Korea, as a Senior Engineer. From 2008 to 2010, he was a Research Assistant Professor with the Department of Computer Science, KAIST. He joined the Faculty of the Division of Computer Science, Sookmyung Women's University, Seoul, in 2010, where he is currently a Professor. His research interests include database systems, query processing, data mining, data streams, and scientific data processing.

MINJI SEO received the B.S. degree from the Division of Computer Science, Sookmyung Women's University, South Korea, in 2018, where she is currently pursuing the master's degree in computer science. Her research interests include databases, data mining, deep learning, and graph embedding.

RYONG LEE received the B.S. degree from the School of Electronics, Telecommunication and Computer Engineering, Korea Aerospace University, South Korea, in 1998, and the M.S. and Ph.D. degrees from the Department of Social Informatics, Kyoto University, Japan, in 2001 and 2003, respectively. From 2003 to 2008, he was a Research Staff Member with the Samsung Advanced Institute of Technology (SAIT), South Korea. Since 2013, he has been with the Korea Institute of Science and Technology Information (KISTI), South Korea. He is currently a Senior Researcher of Research Data Sharing Center, KISTI. His research interests include spatial data analysis, the Internet of Things, smart city, and artificial intelligence.

MINWOO PARK received the B.S. and M.S. degrees from the Division of Computer Convergence, Chungnam National University, South Korea, in 1992 and 2004, respectively. Since 1996, he has been with the Korea Institute of Science and Technology Information (KISTI), South Korea. He is currently a Team Manager of the Research Data Sharing Center, KISTI. His research interests include system architecture, information security, the Internet of Things, smart city, and artificial intelligence.

SANG-HWAN LEE received the B.S. degree from the Department of Electronic Computing, University of Ulsan, South Korea, in 1992, and the M.S. degree in software engineering from Korea University, in 2004. Since 1995, he has been with the Korea Institute of Science and Technology Information (KISTI), South Korea. He is currently the Director of the Research Data Sharing Center, KISTI. His research interests include big data analysis, large research data, data governance, data ecosystems, and artificial intelligence.