IEICE TRANS. INF. & SYST., VOL.E96–D, NO.5 MAY 2013

LETTER

# Partitioned-Tree Nested Loop Join: An Efficient Join for Spatio-Temporal Interval Join

**Jinsoo LEE**[†], ***Nonmember***, **Wook-Shin HAN**[†a)], ***Member***, **Jaewha KIM**[†], ***Nonmember***, ***and*** **Jeong-Hoon LEE**[†], ***Member***

**SUMMARY** A predictive spatio-temporal interval join finds all pairs of moving objects satisfying a join condition on future time interval and space. In this paper, we propose a method called *PTJoin*. *PTJoin* partitions the inner index into small sub-trees and performs the join process for each sub-tree to reduce the number of disk page accesses for each window search. Furthermore, to reduce the number of pages accessed by consecutive window searches, we partition the index so that overlapping index pages do not belong to the same partition. Our experiments show that *PTJoin* reduces the number of page accesses by up to an order of magnitude compared to *Interval_STJoin* [9], which is the state-of-the-art solution, when the buffer size is small.

***key words:*** *spatio-temporal interval join, tree partition, moving objects*

## 1. Introduction

*Predictive* queries are an important query type prevalently used in moving object database systems for supporting applications such as telematics, location-based services, and air traffic control systems. As a predictive query, a predictive spatio-temporal join finds all pairs of moving objects satisfying query conditions on future time and space [2]. In this paper, we handle a *predictive spatio-temporal interval join*, which is one type of predictive spatial-temporal join query that uses a time interval as one of the join conditions. In earlier research, Sun et al. [5] investigated selectivity estimation of predictive spatio-temporal joins without considering join processing. Tao et al. [6], [7] dealt with predictive window queries. Han et al. [2] dealt with predictive spatio-temporal timestamp join only. As an example of a predictive spatial-temporal interval join, it is useful to imagine a query that finds all pairs of cars that could come closer than 10 feet apart between 05:30 and 06:00. The formal definitions for the predictive spatio-temporal interval join are given as follows:

**Definition 1** [5]. Given two sets $R$ and $S$ of spatio temporal objects, a future time interval $[t_s, t_e]$, and a distance threshold $d$, a predictive spatio-temporal interval join finds all pairs of objects $<o_1, o_2>$ such that $o_1$ in $R$, $o_2$ is in $S$, and the distance between the objects $o_1$ and $o_2$ on the time interval $[t_s, t_e]$ is shorter than $d$.

Manuscript received September 10, 2012.
Manuscript revised December 24, 2012.

[†]The authors are with the School of Computer Science and Engineering, Kyungpook National University, Buk-Gu, Daegu, Korea.
a) E-mail: wshan@knu.ac.kr (Corresponding author)

DOI: 10.1587/transinf.E96.D.1206

As the state-of-the-art solution for the spatio-temporal interval join problem, Han et al. [9] have proposed *Interval_STJoin*. In order to reduce the frequency of disk page access, the method sorts the leaf entries of the outer index based on a *space filling curve* (SFC), which is the same sorting technique used in the inner index. Then it performs join processing as the sorting sequence. The method may efficiently perform the join processing if the buffer size is large enough to store all the index pages to be accessed in the near future. However, the method suffers from performance degradation if the buffer size is small. We briefly explain *Interval_STJoin* to point out this problem.

*Interval_STJoin* is a variation of the ordered index-based nested loop joins. Assume that the indexes accessed in the outer and inner loops are $R$ and $S$, respectively. *Interval_STJoin* first sorts the leaf entries of $R$ based on an SFC. Then, it builds tightened bounding boxes, called windows, containing the sorted leaf entries, based on their spatial locations at the join time interval, and performs a window search on $S$ for each window. Performing ordered window searches improves the buffer utilization by increasing the possibility of accessing the node of $S$, which was accessed by the previous window in the consecutive window searches.

Figure 1 illustrates an example of *Interval_STJoin*. Here, we have two trees $R$ and $S$. *Interval_STJoin* first organizes four windows, $WS_1$, $WS_2$, ..., $WS_4$, by sorting the eight leaf entries, $e_1$, $e_2$, ..., $e_8$, of tree $R$. Window $WS_1$ contains leaf entries $e_1$ and $e_3$, and $WS_2$ contains $e_4$ and $e_5$, and so on. Then, for each window $WS_i$, it performs windows search on $S$ to find join results.

However, if the buffer size is small, *Interval_STJoin* cannot gain the benefit of the page buffering effect because the buffers may not store all of the pages to be accessed in the near future. It causes frequent replacement of buffer pages, and thus incurs performance degradation.

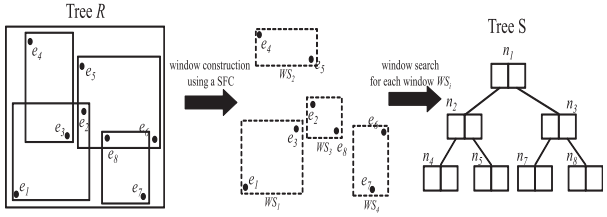


**Fig. 1** An illustration of *Interval_STJoin* [10].

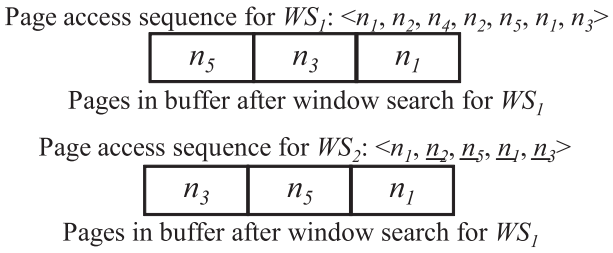


**Fig. 2** Page access sequences and a buffer space for window search $WS_1$ and $WS_2$.

Figure 2 shows page access sequences of window searches $WS_1$ and $WS_2$ and buffer pages after performing each window search. We assume that $WS_1$ accesses five pages, $n_1$, $n_2$, $n_3$, $n_4$, and $n_5$, and $WS_2$ accesses four pages, $n_1$, $n_2$, $n_4$, and $n_5$. Note that the five pages for $WS_1$ include all pages for $WS_2$. If we traverse tree S using depth first search, the page access sequences for $WS_1$ and $WS_2$ are <$n_1$, $n_2$, $n_4$, $n_2$, $n_5$, $n_1$, $n_3$> and <$n_1$, $n_2$, $n_5$, $n_1$, $n_3$> respectively. Assume that the buffer size is three, and we use the LRU buffer page replacement policy. After performing the window search for $WS_1$, index pages $n_5$, $n_3$, and $n_1$ are retained in the buffer. Then, the method performs a window search for $WS_2$. Even if all the index pages for $WS_2$ are accessed by the window search for $WS_1$, the pages except $n_1$ should be read from disk again because they have been replaced due to lack of space in the buffer.

In order to overcome this performance problem with the earlier approach [9], we propose a method, called *PTJoin*, based on a tree partitioning technique. *PTJoin* divides tree $S$ into small sub-trees. Then, it performs window searches for each small sub-tree. If we perform the series of window searches on a small sub-tree, most of the index pages of the sub-tree will be retained in the buffer during the series performance for each sub-tree.

Furthermore, we adopt a partition technique to reduce the number of index pages that are commonly retrieved by consecutive window searches. If the number of the pages is large, we require a large buffer to retain the pages. Here, we partition the index so that overlapping index pages do not belong to the same partition. This allows our method to retain only the smallest number of index pages to perform the series on each sub-tree.

The rest of this paper is organized as follows. In Sect. 2 we present our spatio-temporal interval join method. Section 3 presents the performance evaluations. Finally, we conclude the paper in Sect. 4.

## 2. Partitioned-Tree Nested Loop Join

In this section, we introduce our partitioned-tree nested loop join method. In Sect. 2.1, we introduce the overall algorithm of the partitioned-tree nested loop join, and in Sect. 2.2, the tree partition method is introduced. Hereafter, we use window together with window query. For detailed explanation of an algorithm, we refer readers to the earlier work [2], [10].

### 2.1 Main Algorithm

Algorithm 1 shows the main algorithm of our new spatio-temporal interval join method, called *PTJoin*. *PTJoin* is based on the partitioned-tree nested loop join, which is similar to the block nested loop join [9]. *PTJoin* takes eight parameters. The first two parameters, denoted *rootR* and *rootS*, are the root nodes of trees $R$ and $S$. The next two parameters, $t_s$ and $t_e$ are the starting and ending timestamps of a future time interval. The fifth parameter, $SFC$, is a space-filling curve to order the leaf entries in $R$. Parameters $D_{thr}$ and $bf$ are user defined thresholds for bounding the size of query windows. The last parameter, $sp$, is the size of the partition to be made from S. Note that all parameters except $sp$ are the same as those of Algorithm 1 in the earlier work [9]. *PTJoin* first constructs windows from leaf entries in $R$ by calling the function *MakeWindows* (line 2), which works similar to *Interval_Join* algorithm in the earlier work [9] except that the function does not perform window search whenever a window is constructed. Then, *PTJoin* partitions all leaf nodes of $S$ into $sp$ groups by calling the function *MakePartition* (line 3). Specifics of the function *MakePartition* are described in Sect. 2.2. For each element of the partition $p_i$, *PTJoin* then finds the results between all objects in each window $WS_j$ and all objects in $p_i$ by calling the function *WindowSearch* in the earlier work [9] (lines 4–9). The function Tree at line 6 builds a tree containing all leaf nodes in $p_i$ and returns the root node of that tree. *PTJoin* returns all partial results, from the *WindowSearch* function calls, as final results (line 10).

**Algorithm 1** *PTJoin*(*rootR*, *rootS*, $t_s$, $t_e$, $SFC$, $D_{thr}$, $bf$, $sp$)

**Input:** *rootR*: root node of tree $R$, *rootS*: root node of tree $S$
$t_s$, $t_e$: join time interval
$SFC$: space filling curve
$D_{thr}$: density threshold
$bf$: blocking factor
$sp$: the size of partition
**Output:** $WS$: a list of windows

```
1: initialize results to an empty set.
2: WS = MakeWindows(rootR, rootS, t_s, t_e, SFC, SFC, bf)
3: P = MakePartition(rootS, t_s, t_e, sp, ∅)
4: for each element p_i in P do
5:    for each element WS_j in WS do
6:       partial_results = WindowSearch(WS_j, Tree(p_i), t_s, t_e)
7:       results = results ∪ partial_results
8:    end for
9: end for
10: return results
```

### 2.2 Tree Partitioning

To process the spatio-temporal interval join query efficiently, it is important to choose a "good" partition. In this section, we describe the method of tree partitioning.

In the earlier work [9], to process the spatio-temporal interval join, the method builds a list of window queries $WQ$

using objects in the tree $R$. Then, for each window query $wq_i$, it performs the window search using $wq_i$. For each window query $wq_i$, the method needs to access the disk pages of tree $S$. Since we use the LRU buffer page replacement policy, if the number of window queries needs to access a larger number of disk pages than the allowed buffer size, the buffering effect will be reduced, and the query performance will also be degraded.

For each group of the partition, the goal of tree partitioning is to minimize the number of window queries, whose numbers of page accesses are larger than the buffer size. Formally, for each group $p_k$ of the partition, our goal is minimizing the following Eq. (1).

$$\sum_{i=1}^{|WS|}\left(I\left(\begin{array}{c} PA\Big(WindowSearch(wq_i, Tree(p_k))\Big) \\ > bufferSize \end{array}\right)\right) \quad (1)$$

,where $I(p)$ is an indicator function that returns one, if the predicate $p$ is true, otherwise the function returns zero. And $wq_i$ and $p_k$ are the $i$-th window query from the tree $R$ and $k$-th group of the partition respectively. $PA(WindowSearch(wq_i, Tree(p_k)))$ refers to the number of page accesses to process the window query $wq_i$ over the tree of the elements of $p_k$. Thus, Eq. (1) shows the number of window queries whose page accesses are larger than the buffer size.

Unfortunately, finding the optimal partition is a difficult problem, since the number of all possible partitions is large and increases exponentially as the number of leaf nodes of the tree $S$ increases. To reduce the partition finding time, we use a heuristic approach.

Algorithm 2 shows *MakePartition* for tree partition. *MakePartition* takes five parameters: $rootS$, $t_s$, $t_e$, $sp$, and $entrySet$, and partitions the leaf nodes of the tree $S$ into $sp$ groups. The first parameter, $rootS$, is the root node of tree $S$, which will be partitioned. The next two parameters, $t_s$ and $t_e$, are the starting and ending timestamps for the time interval query, and $sp$ is the size of the partition. $entrySet$ is a set of entries to be partitioned. The parameter, $entrySet$, is initially an empty set. Then it is filled with the entries from the tree $S$ at the first call of the function *MakePartition*(line 2). Then algorithm proceeds with the partitioning(line 3). Then, *MakePartition* partitions $entrySet$ into two groups, $tmpSet1$ and $tmpSet2$, by calling the function $SplitEntries$(line 9). The detailed description of the function $SplitEntries$ follows. Then, partitioning proceeds by calling the function *MakePartition* recursively with the reduced size of partitions(lines 10–11). The function terminates when the size of the partition becomes one or when there are no more entries to be partitioned(lines 5–7).

Now, we explain the function $SplitEntries$, shown in detail in Function 1. $SplitEntries$ first finds two entries, $e_1$ and $e_2$, that have the biggest overlap among all possible pairs of entries in the $entrySet$(line 1), and it puts $e_1$ and $e_2$ into $tmpSet1$ and $tmpSet2$ respectively(line 2). Then, for each entry e except $e_1$ and $e_2$ in $entrySet$, it selects a set between $tmpSet1$ and $tmpSet2$, which have the least overlap size with $e$, and then puts $e$ into the selected set(lines 4–8). If the size of either $tmpSet1$ or $tmpSet2$ exceeds the size of $entrySet$ by half, $SplitEntries$ puts the remaining entries of $entrySet$ into the other set to make the two sets even, and then the function terminates.

**Algorithm 2** *MakePartition*($rootS$, $t_e$, $t_s$, $sp$, $entrySet$)

**Input:** $rootS$: root node of tree $S$
$t_s$, $t_e$: join time interval
$sp$: the size of partition
$entitySet$: a set of entries to be partitioned
**Output:** a partition of all leaf nodes of $S$

1: **if** the function *MakePartition* is never called before **then**
2: $tmpSet = ChildrenNodes(rootS)$
3: $MakePartition(rootS, t_e, t_s, sp, tmpSet)$
4: **end if**
5: **if** $sp$=1 or $|entrySet|$==1 **then**
6: **report** $entrySet$ as an entry of the partition
7: **end if**
8: $tmpSet1=\emptyset$, $tmpSet2=\emptyset$
9: $SplitEntries(entrySet, t_s, t_e, \lceil sp \rceil, tmpSet1)$
10: $SplitEntries(entrySet, t_s, t_e, \lceil sp \rceil, tmpSet2)$

**Function 1** $SplitEntries(entrySet, t_e, t_s, tmpSet1, tmpSet2,)$
**Input:** $rootS$: root node of tree S
$t_s$, $t_e$: join time interval
$entitySet$: a set of entries to be split
**Output:** $tmpSet1$,$tmpSet2$: two disjoint subsets of $entrySet$

1: find two entries, $e_1$ and $e_2$, in $entrySet$, that the size of the overlap between $e_1$ and $e_2$ is the biggest among those of all possible pairs of entries.
2: $tmpSet1=\{e_1\}$, $tmpSet2=\{e_2\}$
3: **for each** entry $e$ in $entrySet$ except $e_1$ and $e_2$ **do**
4: **if** $Overlap(tmpSet1, e) < Overlap(tmpSet2, e)$ **then**
5: $tmpSet1 = tmpSet1 \cup \{e\}$
6: **else**
7: $tmpSet2 = tmpSet2 \cup \{e\}$
8: **end if**
9: **if** $|tmpSet1| > |entrySet|/2$ **then**
10: put all remaining entries of $entrySet$ to $tmpSet2$
11: **Break**
12: **end if**
13: **if** $|tmpSet2| > |entrySet|/2$ **then**
14: put all remaining entries of $entrySet$ to $tmpSet1$
15: **Break**
16: **end if**
17: **end for**

## 3. Experiments

### 3.1 Experimental Setups

We generate experimental data sets using the Generate Spatiotemporal Data (GSTD) tool [8]. The GSTD tool is a data generator used broadly in the performance evaluations on moving objects and is able to generate moving objects with various distributions [3]. In our experiments, we use GSTD to generate data sets with the uniform, Gaussian, and skewed distributions. The maximum speed of an object is set to 2 km per minute.

We also use the same data set as the one generated by Saltenis et al. [4]. This data set simulates a road network; it contains imaginary objects moving on a road. Each

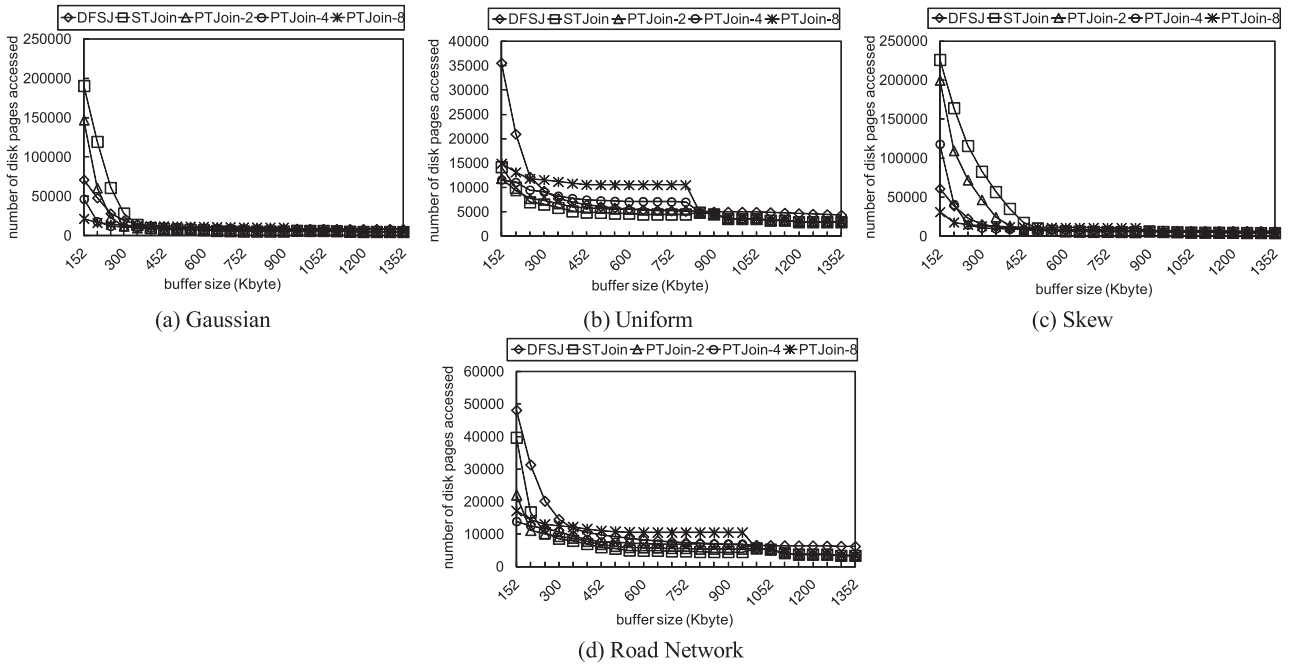


**Fig. 3** Experiment Results for four different Datasets by varying the buffer size($t_s$ = 20, $t_e$ = 30).

object randomly chooses its source and destination points from 200 fixed points randomly distributed in a 1,000 km by 1,000 km space. Once the object arrives at its destination, it randomly chooses the next destination and moves on to there. The maximum speed of each object is 0.75 km/minute, 1.5 km/minute, or 3 km per minute. While moving from a source to a destination, an object accelerates in the first 1/6 of the distance, moves at the maximum speed in the next 2/3, and decelerates in the last 1/6. During this move, each object reports its speed and location every 20 minutes on average.

The data sets indexed by $R$ and $S$ are point objects, and a join using $R$ and $S$ produces all pairs of point objects that are within less than a certain distance (0.2 km) from each other. For simplicity, both data sets have the same number of objects (i.e., | Robject | = | Sobject |); this makes analysis of the experimental results easy. For the road network data set, we use the same data set for building both $R$ and $S$; this simulates a self-join.

All experiments are done on a Windows Server 2003 PC with 512 KBytes unified (data and instruction) L2 cache, 512 Mbytes RAM, and a Pentium IV 2.8 GHz CPU. We set the page size as 4 KBytes. We use LRU as the buffer page re-placement algorithm.

### 3.2 Experimental Results

Figure 3 shows the experimental results for four different datasets (Gaussian, Uniform, Skew, and Road Network). The proposed method, *PTJoin*, is compared with *STJoin* (i.e., *Interval_STJoin*) and with the depth first search based join method DFSJ in the earlier work [9]. Note that the numbers following *PTJoin*, in the legends of the figures, represent the sizes of the partitions. In all the experiments, we evaluate the number of disk page accesses as the buffer size is varied.

Figure 3 (a) shows the result for a dataset whose distribution follows Gaussian distribution. From the figure, we see that, as the buffer size increases, the number of disk page accesses of all the methods decreases. We also see that *STJoin* (i.e., *PTJoin*-1) and *PTJoin*-2 show worse performance with the small buffer size compared to the other methods. This is because, with a small buffer size, the number of page misses increases if the number of partitions is small. We see that *PTJoin*-8 outperforms or is comparable to the other methods within all ranges of buffer size.

Figure 3 (b) shows the result for a dataset whose data are uniformly distributed. Here, we see that *STJoin* and *PTJoins* outperform or are comparable to DFSJ within all the ranges of buffer size. We also see that *PTJoin*-8 shows a performance worse than *PTJoin*-2 or *PTJoin*-4 when the buffer size is smaller than 800 KBytes. This is because the number of disk page accesses in the outer index increases as the number of partitions increases.

Figures 3 (c) and 3 (d) show the results for the datasets, Skew and Road Network, respectively. We see that Fig. 3 (c) has a similar tendency to Fig. 3 (a), and Fig. 3 (d) has a similar tendency to Fig. 3 (b). However, there are cases where *PTJoin* is slightly worse than *STJoin*. This is because *PTJoin* repeatedly accesses $R$ for each group of $S$, while *PTJoin* gains the benefit of the page buffering. Thus, *PTJoin* sometimes can be slightly worse than *STJoin*. Nevertheless, we prefer to use *PTJoin* due to its robust performance.

As explained in Sect. 2.2, we can determine the proper size $k$ of the partition, which minimizes Eq. (1) among all

possible partitions for $S$. Furthermore, if the cost of $STJoin$ by using the cost model in the earlier work [2] is cheaper than the cost of $PTJoin$ with the selected $k$, we choose $STJoin$ instead.

In all the experiments, we see that the number of page accesses for $PTJoin$s are the same as that for $STJoin$ if the buffer size is larger than 850 KBytes (approximately, 10% of the size of the inner index). This is because we do not partition the inner index if the buffer size is sufficiently large to stores the index pages to be accessed in the near future. Thus, within the range of the buffer size, $PTJoin$ performs join processing in the same manner as $STJoin$.

## 4. Conclusions

In this paper, we propose a spatio-temporal interval join algorithm, called $PTJoin$, to find all pairs of moving objects satisfying a join condition on future time interval and space. In the earlier work [9], the authors have proposed $Interval\_STJoin$ to solve this problem. However, $Interval\_STJoin$ shows bad performance results when the buffer size is small. To tackle this problem, $PTJoin$ uses a tree-partitioning technique to reduce the size of the index tree to be searched in the window searches. Moreover, we partition the index so that index pages, which are overlapping index pages do not belong to the same partition. Through extensive experiments, we show that $PTJoin$ outperforms $Interval\_STJoin$ by up to an order of magnitude when the buffer size is small.

## Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2011-0016520). The authors wish to acknowledge the contributions of Byung Suk Lee to the earlier work that was presented in IEEE TKDE 2009.

### References

[1] S. Arumugam and C. Jermaine, "Closest-point-of-approach join for moving object histories," Proc. 22nd Int'l Conf. on Data Engineering, pp.86–86, Atlanta, Georgia, USA, April 2006.

[2] W.-S. Han, J. Kim , B.S. 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, 2009.

[3] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel approaches to the indexing of moving object trajectories," Proc. 26th Intl Conf. Very Large Data Bases , pp.395–406, Cairo, Egypt, Sept. 2000.

[4] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, "Indexing the positions of continuously moving objects," Proc. ACM SIGMOD Int'l Conf. on Nanagement of Data, pp.331–342, Dallas, Texas, USA, May 2000.

[5] J. Sun, Y. Tao, D. Papadias, and G. Kollios, "Spatio-temporal join selectivity," Information Systems, vol.31, no.8, pp.793–813, 2006.

[6] Y. Tao, J. Sun, and D. Papadias, "Analysis of predictive spatio temporal queries," ACM Trans. Database Syst., vol.28, no.4, pp.295–336, 2003.

[7] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: An optimized spatio-temporal access method for predictive queries," Proc. 29th Int'l Conf. on Very Large Data Bases, pp.790–801, Berlin, Germany, Sept. 2003.

[8] Y. Theodoridis, J. Silva, and M. Nascimento, "On the generation of spatiotemporal datasets," Advances in Spatial Databases, vol.1651, pp.147–164, 1999.

[9] W.-S. Han, J. Lee, and Y.-H. Park, "On supporting spatio-temporal interval join," Information Journal (to appear).

[10] J. Lee, W.-S. Han, J. Kim, and J.-H. Lee, "Partitioned-tree nested loop join: An effective join for spatio-temporal interval join," Technical Report, 2012. (available at http://www-db.knu.ac.kr/download/PTJoin/PTJoin.pdf).