Efficiently Answering Top-k Window Aggregate Queries: Calculating Coverage Number Sequences over Hierarchical Structures

Jianqiu Xu[1] Raymond Chi-Wing Wong[2]

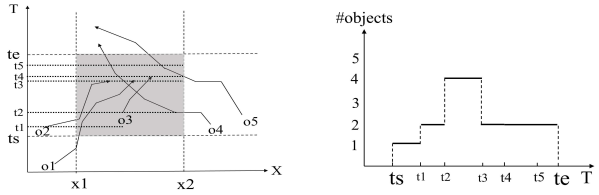
[1] Nanjing University of Aeronautics and Astronautics, China
[2] The Hong Kong University of Science and Technology, China
jianqiu@nuaa.edu.cn raywong@cse.ust.hk

***Abstract*—Given a set of spatio-temporal objects, a top-k window aggregate query reports top-k tuples that are ordered with respect to the number of objects during a given time interval and within a spatial range. For example, when analyzing traffic density in a city, one wishes to retrieve top-k time intervals in a certain area that are decreasingly ordered according to the number of vehicles passing by. As simply performing sequential scan over all objects is a costly procedure, an index structure is typically built to enhance the query performance. A crucial step during the evaluation is to determine the number of objects in an arbitrary node, called *coverage number sequence*. This is a challenging task since objects appear and disappear at different time points such that the number of objects in the query node changes over time. Also, as a hierarchical index structure, the value of a node at high level is achieved by performing the aggregation over its child nodes. Simply enumerating all objects rooted in the query node suffers from performance issues mainly due to (i) traversing the sub-tree to retrieve a large number of time points and (ii) repeatedly performing the aggregation at certain time points. We propose an efficient approach to solve the performance issue for both R-tree and Octree and support updating for new arrival data objects being inserted into the index. Our approach outperforms alternative methods in general according to a thorough analysis on the complexity. Coverage number sequences as well as proposed optimization techniques are utilized to enhance the performance of *window aggregate queries*. We confirm the superiority of our approach over alternative methods by performing a comprehensive experimental evaluation over large real datasets in a database system.**

I. Introduction

Spatio-temporal databases manage spatial and time-varying characteristics of entities and have been extensively studied in a variety of applications including traffic management, weather monitoring and mobile commerce [3], [8], [7]. There is a kind of spatio-temporal queries that report k results based on a ranking function. Application users may not need to have the overall result as the size could be large but retrieve k best results. Example queries include k nearest neighbor queries [12] and top-k similarity queries [27]. In this paper, we study a novel kind of top-k queries that report the summarized information over a set of objects fulfilling the spatio-temporal condition, called *top-k window aggregate queries*. Such a query requests a spatio-temporal window to look for objects being located inside and performs the aggregation counting over them to report k tuples ordered by the number of objects. For example, counting the cardinality of vehicles crossing a certain area (e.g., commercial center) during rush hour is a fundamental problem in traffic management. The reported information benefits urban plans such as constructions of roads and underpasses or adjusting traffic lights.

Example 1. Assume that there are five objects $\{o_1, o_2, o_3, o_4, o_5\}$ moving long the x-axis and the query window (gray area) is $([x_1, x_2), [t_s, t_e))$, as shown in Figure 1(a). The number of objects appearing in $[x_1, x_2)$ during $[t_s, t_e)$ is reported in Figure 1(b). After traversing the moving integer, the query returns $\langle([t_s, t_1), 1)\rangle$ (k = 1) if we consider the minimum number of objects.



(a) 1D query region (b) #objects in the window

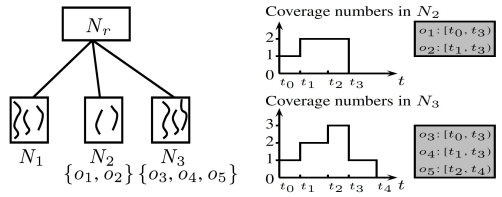
Fig. 1. Example of top-k window aggregation queries

Efficiently answering this query requires indexing techniques because the performance of sequential scan over the datasets is prohibitively expensive. We mainly focus on the R-tree family in this paper. The pruning procedure makes use of the aggregation information of objects being stored in a node. Since spatio-temporal objects appear and disappear at different time points, the number of objects varies at each time point, complicating the calculation procedure of aggregation counting. In addition to *top-k* window aggregate queries, the aggregation information of internal structures can be utilized as a black box to (i) enhance the query performance of other top-k queries such as [12] [20] [27] (e.g., [12] utilizes coverage number sequences to report the minimum number of qualified objects in a node without accessing the underlying structure), and (ii) test and analyze internal data structures for system developers, e.g., "*is there a subtree containing more objects than others at the level?*".

Given an R-tree built on a set of spatio-temporal objects [18] and an arbitrary query node, we study efficiently reporting the number of objects over time in the node, called *coverage number sequence*. Such a query reports the number of objects at each time point in the query node. For the sake of simplicity, we use the notation *CovNumSeq* in the following. *CovNumSeq* can be considered as performing an aggregation counting

query with the *structure constraint*, which means that only objects rooted in the query node (sub-tree) are processed. This differs from traditional aggregation queries which only define the time constraint.

Example 2. Assume that a 3D R-tree is built on a set of moving objects, as depicted in Figure 2(a). The indexed objects are pieces of movements and each entry of a leaf node contains a reference to one piece of movements. Given a node, its CovNumSeq is defined by a sequence of items each of which consists of a time interval and the number of objects over this time interval in the node. The CovNumSeqs of N_2 and N_3 are illustrated in Figure 2(b). Consider the node N_3. o_3 appears at $[t_0, t_3)$, o_4 appears at $[t_1, t_3)$ and o_5 appears at $[t_2, t_4)$, resulting in CovNumSeq = $\langle([t_0, t_1), 1), ([t_1, t_2), 2), ([t_2, t_3), 3), ([t_3, t_4), 1)\rangle$. To obtain CovNumSeq of N_r, we perform an aggregation over CovNumSeqs of N_1, N_2 and N_3.



(a) R-tree (b) CovNumSeqs

Fig. 2. Example of CovNumSeqs over an R-tree

In the literature, a number of temporal aggregation queries have been studied, which can be classified into (i) *cumulative temporal aggregate queries* [25], (ii) *range temporal counting* [26], [22] and (iii) *spatio-temporal aggregation counting* [14], [24].

The first type of queries reports the summarized information of objects whose intervals intersecting a query window formed by defining an offset around a time point. If the offset becomes zero, the query is called *instantaneous temporal aggregate* in which the time line is partitioned into time points. Typical aggregate operators are COUNT, SUM and AVG. Our query can be treated as *instantaneous temporal aggregation* over the relevant objects. That is, CovNumSeq is achieved by performing the aggregation over objects fulfilling the structure condition instead of the temporal condition. In Figure 2, the time interval of N_3 is $[t_0, t_4)$ which intersects with that of N_2 ($[t_0, t_3)$), but the evaluation will not involve $\{o_1, o_2\}$ in N_2 because they do not belong to N_3 (structure constraint). The second type of queries computes the total number of objects whose time intervals intersect a query interval. However, the interval of a counted object may be not identical to the query interval such that at certain time points the number of objects is actually smaller than the reported value. Considering N_2 in Figure 2(b), we set the query interval $[t_0, t_3)$. *Range temporal counting* will report 2, but there is only one object during $[t_0, t_1)$. The result of range temporal counting is in fact an instant value of CovNumSeqs that represents the maximum value over the query.

The third type of queries returns the number of objects fulfilling the spatio-temporal condition. Those algorithms can be adapted to report CovNumSeqs but the performance is sub-optimal. One extracts temporal (spatio-temporal) data from the query node to form the temporal (spatio-temporal) window. Considering N_2, we have the temporal window $[t_0, t_3)$ as the query time. The evaluation will process $\{o_3, o_4, o_5\}$ in N_3 since their intervals intersect with $[t_0, t_3)$. As a result, at $[t_0, t_1)$ we have $\{o_1, o_3\}$, at $[t_1, t_2)$ we have $\{o_1, o_2, o_3, o_4\}$, and at $[t_2, t_3)$ we have $\{o_1, o_2, o_3, o_4, o_5\}$, resulting in CovNumSeq = $\langle([t_0, t_1), 2), ([t_1, t_2), 4), ([t_2, t_3), 5)\rangle$. This leads to false dismissals because $\{o_3, o_4, o_5\}$ is not in the query node N_2. One can remove objects not in the query node after the spatio-temporal evaluation but this requires extra storage and query costs, significantly decreasing the performance. On the one hand, evaluating which node an object belongs to needs to maintain the relationship between objects and nodes (i.e., the leaf node an object belongs to and all nodes on the path from the leaf node to the root node). This is done for all data objects. On the other hand, if a large query window is requested, e.g., a non-leaf node, a number of objects will be processed but they may not contribute to the result.

CovNumSeq reports a *distinct count* at each time point and such a value changes over time as objects appear and disappear at different time points. There is no way to exactly summarize distinct objects substantially better than by simply enumerating all of them [21]. The number of processed objects is small if a leaf node is requested because only a few objects are evaluated. However, the number of processed objects is large for a non-leaf node since all objects in the subtree are evaluated. An extreme case occurs when a root node is queried. To report the result, the evaluation traverses down to leaf nodes to collect data objects and perform the aggregation over all objects rooted at the node, which is a costly procedure.

We propose a *divide-and-conquer* strategy that calculates CovNumSeq for each child node and merges these CovNumSeqs to be the final result. Based on a thorough analysis, we find the bottleneck is that merging CovNumSeqs over child nodes involves repeatedly accessing a large number of duplicate time points. Motivated by this, we create the set of time points in advance and study common endpoints among different sub-spaces. The computation cost is reduced by determining the subset of endpoints involved for computation. We use an R-tree as the study case because the structure has been popularly used in spatio-temporal databases, while our solution applies to Octree [15] as well. We develop a method to efficiently update CovNumSeqs for on-line applications. One increases the counters for time points overlapping with the historical data and performs the computation for new time points.

CovNumSeqs accompanied with spatial distances are utilized to enhance the performance of *window aggregate queries*. Given a set of moving objects and a spatio-temporal window, one calculates the number of objects moving in the query area at each time point and reports k time intervals with the minimum number of objects. The performance of traditional spatio-temporal counting methods is sub-optimal

because iteratively evaluating each object is costly for large datasets. We enhance the query performance by making use of CovNumSeqs and the pruning strategy based on approximate counting.

The contributions are summarized as follows:

- We formalize the problem of CovNumSeqs and top-k window aggregate queries.
- We design an efficient algorithm to compute CovNumSeqs and perform a thorough theoretical analysis on time and space complexities. The solution applies to both R-tree and Octree.
- An efficient updating method is developed for CovNumSeqs for new arrival data, and the complexity is analyzed.
- By employing CovNumSeqs, we design efficient algorithms to answer window aggregate queries.
- We develop all techniques in a prototype database system SECONDO and perform a comprehensive experimental evaluation using large real datasets. The results demonstrate that our proposal offers more than an order of magnitude performance improvement in comparison with four alternative methods.

The rest of the paper is organized as follows: The related work is reviewed in Section II. We formalize the problem in Section III, present the approach of computing CovNumSeqs in Section IV and provide a thorough analysis in Section V. Updating CovNumSeqs is introduced in Section VI. Algorithms of answering window aggregate are introduced in Section VII. The evaluation is conducted in Section VIII and we conclude the paper in Section IX.

II. Related work

Temporal aggregation. Computing temporal aggregates is a fundamental operator in temporal databases, which reports summarized values about tuples with time-evolving attributes [2], [28], [26], [10], [19]. Typical functions include COUNT, SUM, AVG, MIN and MAX. They can be classified into two categories: *instantaneous* and *cumulative*. The former partitions the time line into time points and the value at each time point is computed from the set of tuples valid at the point. The latter reports the value that is computed over all tuples whose intervals overlap the window formed by placing an offset around the time point.

Temporal aggregation algorithms include *nonindexed evaluation* and *indexed evaluation* algorithms. The former scans the temporal relation every time a query is issued, and the latter uses a disk-based data structure. According to [22], aR-tree (aggregate R-tree) [17] is the most "practical" method to process temporal aggregates. The method augments the R-tree with aggregate information in intermediate entries. If a node is totally contained in the query region, the aggregate number is directly retrieved without accessing the sub-tree. Existing approaches such as *Balanced tree* [16], *Parallel fashion* [9] and *Timeline* [13] perform the temporal aggregation over objects overlapping query timestamps, but this is not the case for coverage number sequences. In our query, only *relevant* objects are considered which are the objects located in the query node. The evaluated data is in fact a subset of objects overlapping query timestamps and efficiently determining the subset is not a trivial task. Each object belongs to a list of nodes in the structure.

Approximate temporal aggregation with nearby coalescing returns an interval for most users when a complete interval for all users is not available [4]. An *approximate temporal count* retrieves a value that deviates from the precise result by less than an error bound ϵ [22]. The approximation ration can be set by zero such that the accurate result is returned. However, the approach setting $\epsilon = 0$ does not count the number of objects at each time point but only the objects whose intervals intersect the query. We evaluate whether the keys (set the leaf node as key) are identical to the query node. Besides, comparing keys for non-leaf nodes involves a number of node ids, complicating the evaluation procedure.

Spatio-temporal aggregation. One solution is to decompose a spatio-temporal object into different extents including spatial extent, temporal extent and the combination of spatial and temporal extents, and then perform the aggregation over each extent [14]. *Papadias et al.* propose the *aRB-tree* (aggregate R-B-tree) for the efficient processing of spatio-temporal average count queries [18]. Spatial regions are indexed in an R-tree and each entry of the R-tree is associated with a pointer to a B-tree that stores historical aggregate data (the overall number of objects) about the entry. The *aRB-tree* facilitates aggregating processing by eliminating the procedure of accessing nodes totally enclosed by the query. However, the structure is not directly applicable for distinct counting queries because it does not take into account multiple object occurrences. The structure only maintains summarized data without information about individual objects. Thus, duplicate elimination techniques cannot apply. One can use a B-tree on node ids to find the requested node. However, the aggregate data in each non-leaf entry of the B-tree defines the overall number of objects instead of the count at each time point. Given a query region and an interval, a *spatio-temporal aggregate query* retrieves summarized information about objects that appeared in the region during the query interval [21]. The reported counting is an instant value instead of the cardinality of objects at each time point. The method [23] finds the average number of objects per time-stamp during a time interval, but the proposed structures only store summarized data without information about individual objects. This does not support our issue because the number of objects at each time point is required.

III. Problem definition

A. Formalizing CovNumSeqs

Coverage number sequences are represented by moving integers. Let *Instant* denote a set of all possible time points each of which is represented by a real number.

Definition 1 ***Moving integers***

$D_{\underline{mint}} = \{\langle([t_1, t_2), c_1), ..., ([t_n, t_{n+1}), c_n)\rangle | t_i < t_{i+1} \wedge t_i, t_{i+1} \in \mathit{Instant}, c_i \geq 0 \wedge c_i \in D_{\underline{int}}\}$

Let $\mathcal{O}$ be the set of data objects in a node and $T(o)$ ($o \in \mathcal{O}$) return the time interval of an object.

Definition 2 ***CovNumSeq***

Given an R-tree node N, its CovNumSeq denoted by cov(N) $\in D_{\underline{mint}}$ represent the number of objects at each time interval over $\bigcup_{o \in \mathcal{O}} T(o)$. That is, $\forall$ ($[t_1, t_2)$, c) $\in$ cov: $c = |\mathcal{O}'|$, $\mathcal{O}' = \{o |\ o \in \mathcal{O} \wedge [t_1, t_2) \subseteq T(o)\}$.

Each item ($[t_1, t_2)$, c) defines the number of objects in the node during a time interval. An object may contribute to the counters in several items if its time period overlaps with those time intervals. If a leaf node is processed, the item for each child (a data object) is simple, i.e., ($[t_1, t_2)$, 1). If a non-leaf node is processed, CovNumSeq is achieved by performing the aggregation over all child nodes.

Example 3. As reported in Figure 3, there are four objects in N_1 ($\{[t_0, t_2)$, $[t_1, t_3)$, $[t_3, t_5)$, $[t_4, t_5)\}$) and two objects in N_2 ($\{[t_0, t_2)$, $[t_1, t_3)\}$). In N_1, the object with $[t_0, t_2)$ contributes to counters in items ($[t_0, t_1)$, 1) and ($[t_1, t_2)$, 2). CovNumSeqs of N_r is achieved by aggregating *cov*(N_1), *cov*(N_2) and *cov*(N_3).

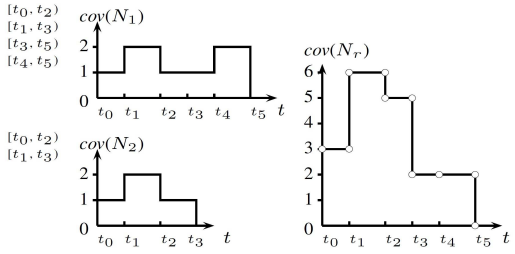


Fig. 3. CovNumSeqs of nodes N_1, N_2 and N_r

B. Top-k Window Aggregate Queries

The query finds data objects whose time intervals intersect the query window and returns k values summarizing the information contained in the result.

Definition 3 ***Window aggregation minimum queries*** *(Win-min for short)*

The query, defined by $q(r, t, k)$, calculates the number of objects appearing in the region $q.r$ during $q.t$, and reports k time intervals with the minimum number of objects. We use Cov_q to denote the result such that

(i) $|Cov_q| = q.k$;

(ii) $\forall$ ($[t_1, t_2)$, c) $\in Cov_q$: $c = |O_q| > 0$, $O_q = \{o |\ [t_1, t_2) \subseteq T(o) \wedge o$ intersects $q.r$ during $[t_1, t_2)\}$;

(iii) $\nexists$ ($[t_s, t_e)$, c') $\notin Cov_q$, $c' > 0$, $[t_s, t_e) \subseteq q.t : \exists$ ($[t_1, t_2)$, c) $\in Cov_q$, $c' < c$.

The shape of the query region is arbitrary such as rectangle, triangle and circle. In the following, we consider a rectangle because other shapes can be processed by first using the rectangle to perform an approximate query and then doing the refinement.

IV. Compute CovNumSeqs

A. The framework

We outline the calculation procedure in Figure 4. The input parameters are an R-tree R, an auxiliary structure denoted by $\mathcal{P}$ and a node id. Our solution named *Cov* consists of three steps: (i) building a partition list on data objects, (ii) updating counters for child nodes and (iii) building the overall CovNumSeqs.

We explain the procedure by considering the root node as the procedure of other nodes is a subroutine of processing the root node. Initially, one builds a structure called *partition list* that records the number of objects at each time interval. The partition list, represented by a binary tree, contains a sequence of called *elementary intervals*, which are obtained by sorting the endpoints of argument intervals. Each pair of consecutive endpoints defines an elementary interval. Next, for each child, we access the partition list to update counters. In the end, we build CovNumSeq on the partition list. The structure $\mathcal{P}$ maintains intermediate results and contains a set of elements in the form of (*nid*, p), where *nid* is a node id and p is a partition list.

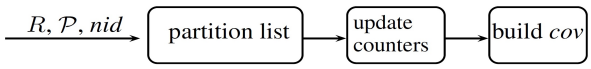


Fig. 4. An outline of the solution

Definition 4 ***Partition list***

Let $p \in D_{\underline{mint}}$ be a partition list built on data objects in which each element consists of a time interval and a counter representing the number of objects during the interval.

We build a partition list to maintain CovNumSeqs of each accessed node and progressively update the structure by values of its child nodes. Depending on whether a leaf node or a non-leaf node is processed, different procedures are executed. A partition list will become CovNumSeqs after the completion of the algorithm. We give the main algorithm in Algorithm 1.

B. Processing Leaf Nodes

Given a leaf node, we scan all data objects and insert start and end time points with counters into the partition list. All counters are initialized to be 0. For each object, we access the partition list to update the counter at each time point contained by the object. A time point may be contained by several data objects and the counter increases as each object is evaluated. Next, we merge consequent time points if their counters are the same. This is to reduce the length of the partition list. Later, data objects in non-leaf nodes will be processed by traversing the partition list to update the counters. The smaller the list is, the smaller cost will be. In the end, we build the coverage number sequence and return the result. The algorithm is given in Algorithm 2.

Example 4. Figure 5 illustrates the procedure of calculating CovNumSeq for N_1. The node contains four objects and the partition list is initialized first. Then, the counter at each time

Algorithm 1 *Cov*

Input: R: a 3D R-tree;
$\mathcal{P}$: an auxiliary structure;
nid: query node id.
Output: *cov*

1: $N \leftarrow$ *GetNode*(R, *nid*);
2: **if** N *is a leaf node* **then**
3: **return** *Cov_Leaf*(N, $\mathcal{P}$); ▷ *Algorithm 2*
4: ***else***
5: ***for all*** $N[i]$ ***do*** ▷ *build the partition list*
6: *Cov*(R, $\mathcal{P}$, $N[i]$); ▷ *process child nodes*
7: *for each* $(t,\ c) \in \mathcal{P}[N[i]].p$ *insert* $(t,\ 0)$ *into* $\mathcal{P}[N].p$;
8: ***for all*** $N[i]$ ***do*** ▷ *update counters*
9: *let* $\langle t_s, ..., t_e \rangle$ *be all endpoints contained by* $N[i]$;
10: ***for all*** $([t_s,\ t_{s+1}],\ c') \in \mathcal{P}[N].p$ ***do***
11: *find* $([t_s,\ t_{s+1}],\ c) \in \mathcal{P}[N[i]].p$ *and update* $c' \leftarrow c'+c$;
12: *calculate cov on* $\mathcal{P}[N]$;
13: ***return*** *cov*;

Algorithm 2 *Cov_Leaf*

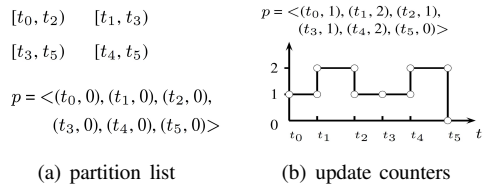
Input: N: a leaf node;
$\mathcal{P}$: an auxiliary structure.
Output: *cov*

1: $p \leftarrow \emptyset$;
2: **for all** objects in $N[i]$ **do** ▷ initialize the partition list
3: *let* t_s and t_e *be start and end points of* $N[i]$;
4: $p \leftarrow (t_s,\ 0)$, $p \leftarrow (t_e,\ 0)$;
5: **for all** objects in $N[i]$ **do** ▷ update counters
6: **for all** endpoints contained by $N[i]$ **do**
7: *find the corresponding* $p[i]$ and *update* $p[i].c$++;
8: **for all** $p[i]$ **do** ▷ merge time points
9: **if** $p[i+1].c = p[i].c$ **then**
10: *remove* $p[i+1]$;
11: $\mathcal{P} \leftarrow (N,\ p)$;
12: *build cov on the partition list* p;
13: **return** *cov*;

point is updated. For simplicity, we only provide the start point of an interval in the partition list since the end point is equivalent to the start point of its consequent interval. There are two objects between $[t_1,\ t_2)$ and $[t_4,\ t_5)$, and only one object exists during other time intervals. Next, we merge two intervals $[t_2,\ t_3)$ and $[t_3,\ t_4)$ into $[t_2,\ t_4)$ as the counters at the two intervals are the same. This is done by removing $(t_3,\ 1)$ from the partition list. Finally, we build CovNumSeq and return *cov* = $<([t_0,\ t_1),\ 1),\ ([t_1,\ t_2),\ 2),\ ([t_2,\ t_4),\ 1),\ ([t_4,\ t_5),\ 2)>$ as the result.

C. Processing Non-leaf Nodes

We build CovNumSeq for each child node during which the partition list is created. If the child node is still a non-leaf node, the procedure traverses down the tree until the leaf



(a) partition list (b) update counters

Fig. 5. Build *cov* for N_1

level (lines 5-7 in Algorithm 1). Then, for each child node we update the counter for each $(t,\ c)$ in the partition list. The counter is increased by iteratively adding the number of objects at t for each child (lines 8-11 in Algorithm 1). After merging CovNumSeq of a child node into that of the parent node, we release the space maintained for the partition list as the value for such a child node has been determined already. Finally, we compute CovNumSeq based on the partition list.

Example 5. Consider building CovNumSeq for N_r, as illustrated in Figure 6. For each partition list of its child node, i.e., $\{p(N_1),\ p(N_2),\ p(N_3)\}$, we insert time points into $p(N_r)$ and receive $p(N_r)$ = $<(t_0,\ 0),\ (t_1,\ 0),\ (t_2,\ 0),\ (t_3,\ 0),\ (t_4,\ 0),\ (t_5,\ 0)>$. Next, the counter at each time point is updated by adding the values of child nodes. The dashed line in Figure 6 shows the result after processing $p(N_1)$ and the solid line indicates the final result.

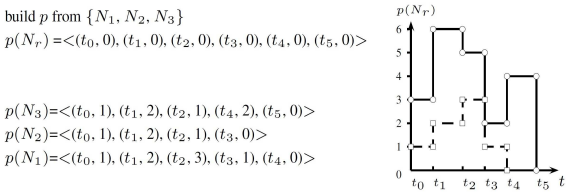


Fig. 6. Build *cov* for N_r

D. Applying to Octree

Algorithm 1 also applies to Octree [15] which is a three-dimensional analog of quadtree. The space is recursively divided into cubes in an adaptive way and data objects will be inserted into corresponding cells. However, the procedure of processing nodes in Octree differs from that of processing an R-tree in two aspects: (i) The node capacity of non-leaf nodes in R-tree is determined by block size, but the node capacity of non-leaf nodes in Octree is 8; (ii) To build a compact structure, an Octree defines a threshold to restrict the minimum number of data objects in a leaf node. As a result, leaf nodes without containing enough data objects will be merged into the parent node and become *virtual* nodes which will not be processed for computing coverage number sequences. For these merged leaf nodes, CovNumSeqs are retrieved in the parent node.

V. Theoretical Analysis

Given a query node, let cov_i be CovNumSeq of the ith child and $cov_{1,i}$ $(i \in \{1, ..., f\})$ be CovNumSeq aggregated from cov_1 to cov_i, respectively. We use $|cov|$ to return the number of time points and have

$$|cov_{1,i}| \leq |cov_{1,i-1}| + |cov_i| \quad (1)$$

Let us consider a leaf node. $cov_{1,2}$ is built on the first two objects and has 4 time points at most. Since time points in cov_i may not exist in $cov_{1,i-1}$, inserting cov_i into $cov_{1,i-1}$ will create time points at which the number of objects changes. The procedure of determining CovNumSeqs is in fact calculating $cov_{1,f}$ (f is the node capacity). One starts from the first two objects to achieve $cov_{1,2}$ and then inserts cov_3 to achieve $cov_{1,3}$. Repeat inserting cov_i into $cov_{1,i-1}$ until the fth object is processed. The time cost is measured by the number of processed time points.

Time complexity. *Given an R-tree node located at level h, we need $O(f^h \cdot f_{avg})$ time to compute CovNumSeq in which f_{avg} is the average number of time points contained by an object in a leaf node.*

Proof

Leaf nodes. *We have $h = 1$. Building the partition list requires $O(f \cdot \log f)$ time as each time point is inserted into a binary tree, resulting in $O(2 \cdot f)$ time points. Updating counters takes $O(f \cdot (\log f + f_{avg}))$ time as we need $O(\log f)$ time to find the start point in the partition list for a data object and then iteratively update f_{avg} counters. Merging time points needs $O(f)$ time as all time points are accessed once. To sum up, we need $O(f \cdot (\log f + f_{avg})$ time in total.*

Non-leaf nodes. *The ith child node of a non-leaf node N is denoted by $N[i]$. To compute cov(N), we calculate cov($N[i]$) for each child, insert time points contained by $N[i]$ into the partition list and update the counters. The time costs of building cov(N) and cov($N[i]$) are denoted by Cost(N) and Cost($N[i]$), respectively.*

$$Cost(N) = f \cdot (Cost(N[i]) + |cov(N[i])| \cdot \log(f \cdot |cov(N[i])|)) \quad (2)$$

$$|cov(N[i])| = f^{h-1} \cdot f_{avg} \quad (3)$$

$$\Rightarrow Cost(N) = f \cdot (Cost(N[i]) + f^{h-1} \cdot f_{avg} \cdot \log(f^{h-1} \cdot f_{avg}))$$

$$\Rightarrow Cost(N) = f \cdot Cost(N[i]) + f \cdot f_{avg} \cdot \log(f^{h-1} \cdot f_{avg})$$

if $h = 2$,

$Cost(N[i]) = f \cdot f_{avg} \Rightarrow Cost(N) = f^2 \cdot f_{avg}$

if $h = 3$,

$Cost(N[i]) = f^2 \cdot f_{avg} \Rightarrow Cost(N) = f^3 \cdot f_{avg}$

$$\Rightarrow Cost(N) = f \cdot f^{h-1} \cdot f_{avg} + f^{h-1} \cdot f_{avg} \cdot \log(f^{h-1} \cdot f_{avg})$$

$$= O(f^h \cdot f_{avg} \cdot (1 + \frac{(h-1) \cdot \log f \cdot f_{avg}}{f}))$$

$$= O(f^h \cdot f_{avg})$$

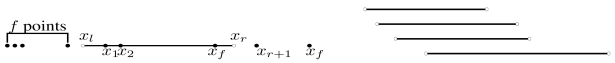
□

We analyze f_{avg} as follows. The lower bound is straightforward $f_{avg} = 2$ because each object at least contains two time points. Consider the upper bound.

Lemma 1 *The upper bound for f_{avg} is $f + 1$.*

Proof *We prove that (i) $f_{avg} = f + 2$ is not possible and (ii) $f_{avg} = f + 1$.*

Case (i): This is proved by contradiction. Let $[x_l, x_r)$ be the time interval of an object which contains $f + 2$ time points. By removing endpoints x_l and x_r, the rest f time points are denoted by $<x_1, ..., x_f>$, as demonstrated in Figure 7(a). Consider the object containing x_1 as one of endpoints. Since the object contains $f + 2$ time points, the other endpoint can only be located at $x_{r+1} > x_r$. This is because if the other endpoint is smaller than x_l, there will be f time points located on the left of x_l (the object contains $f + 2$ time points). The object already contains $<x_1, ..., x_f>$. Then, the overall number of time points will be $f + f + 2$, contradicting the condition that f objects can have $2 \cdot f$ time points in maximum. If the other endpoint is located between $[x_2, x_r)$, the object cannot contain $f + 2$ time points. Repeating the same inference for $\{x_2,...,x_f\}$, there will be $\{x_{r+2},...,x_f\}$ time points on the right of x_r. As a result, there will be $f + 2 + f$ time points. This contradicts the condition that f objects can have $2 \cdot f$ time points in maximum. Case (ii): f objects are distributed in such a way that each object contains $f + 1$ points among which $f - 1$ points come from the other objects, as illustrated in Figure 7(b). □



(a) f_{avg} = f + 2 (b) f_{avg} = f + 1 (f = 4)

Fig. 7. The upper bound for f_{avg}

Let $F_{avg} = \{2, 3, ..., f + 1\}$ be the set of all possible values for f_{avg}. We analyze the probability of each value, denoted by $P(F_{avg})$. The value depends on two factors: (i) the overall number of time points and (ii) the distribution of time point. If $f_{avg} = f$, the overall number of time points cannot be smaller than f. If the overall number of time points is $2 \cdot f$, f_{avg} can be small (e.g., 2, the case that no two objects intersect) or large (e.g., $f + 1$). Given f objects, the number of time points belongs to $\mathcal{F} = \{2, ..., 2 \cdot f\}$ and the probability of each case is denoted by $P(\mathcal{F}) = \frac{1}{2 \cdot f - 1}$.

Lemma 2 *Given f objects each of which contains f_{avg} time points, the overall number of time points is at least $2 \cdot f_{avg} - 2$.*

Proof *Let $[x_l, x_r]$ be an object containing f_{avg} points, denoted by $\{x_l, x_1, ..., x_{f_{avg}-2}, x_r\}$. Consider the object containing x_1 as the endpoint. If the other endpoint is located at $x_{l-1} < x_l$, there will be $f_{avg} - 2$ time points on the left of x_l, resulting in $f_{avg} - 2 + f_{avg}$ time points. If the other endpoint is located at $x_{r+1} > x_r$, we have one more point besides f_{avg} time points. Repeating the same procedure for $x_2, ..., x_{f_{avg}}$, we have f_{avg}*

- 2 endpoints $> x_r$. Thus, there will be at least f_{avg} + f_{avg} - 2 time points. □

Let *Count*(t) and c be the average number of endpoints contained by an interval and the average number of alive objects at a time point overall the dataset, respectively. Given a set of n objects, the number of endpoints is between $O(2)$ and $O(2 \cdot n)$. Theoretically, we have *Count*(t) $\in \{2, ..., n + 1\}$ and $c \in \{1, ..., n\}$. The lower bound is simple. Consider the upper for *Count*(t). This occurs when the data follows the distribution in Figure 7(b). The upper bound for c occurs when interval endpoints are identical to each other. Based on Lemma 2, we derive some values of $P(F_{avg})$ as follows.

$$\begin{aligned} P(\{f, f+1\} \subset F_{avg}) &= P(\mathcal{F}) \cdot \frac{2}{f-1} \\ &= P(\{2f-2, 2f-1, 2f\}) \cdot \frac{2}{f-1} \\ &= \frac{3}{2 \cdot f - 1} \cdot \frac{2}{f-1} \end{aligned} \tag{4}$$

$$\begin{aligned} P(\{2, ..., \frac{f}{2}\} \subset F_{avg}) &= P(\mathcal{F}) \cdot \frac{1}{2} \\ &= P(\{2, ..., f-2\}) \cdot \frac{1}{2} = \frac{f-3}{2 \cdot f - 1} \cdot \frac{1}{2} \end{aligned} \tag{5}$$

A summarization of complexities of our solution and alternative methods is provided in Table I [1]. In a database system, f is determined by block size and is usually between some dozens and a hundred. According to Equations (4) and (5), the probability for a large value f_{avg} is low and in most cases small values appear. For example, by setting f = 50, we have $P(\{f, f+1\}) = \frac{6}{4851} = 0.1\%$ and $P(\{2, ..., \frac{f}{2}\}) = \frac{47}{198} = 23.7\%$. To conclude, in quite a few cases ($f_{avg} \in \{f, f + 1\}$) our solution achieves the same performance as [12], but in most cases our solution significantly outperforms the existing technique as f_{avg} is a small value, e.g., 2 or 3. We will demonstrate this in the experimental evaluation.

TABLE I
TIME COMPLEXITIES OF COMPUTING COVNUMSEQS

Cov	$O(f^h \cdot f_{\text{avg}})$
SB-tree [25]	$O(f^{h-1} \cdot c \cdot \log f^h)$
SegB$^+$-tree [4]	$O(f^h \cdot c \cdot \log f^h)$
BTA [5]	$O(f^h \cdot \text{Count}(t) \cdot \log f^h)$
NN [12]	$O(f^{h+1})$
f is the node capacity and h is the tree height. c is the average number of objects at a time point.	
$f_{\text{avg}} \in \{2,..., f+1\}$ and $c \gg \text{Count}(t)$, $c \gg f$. Consider N_2 in Figure 2(b). o_1 contains 4 points and o_2 contains 3 points, resulting in $f_{\text{avg}} = \frac{4+3}{2} = 3.5$.	
Count(t) is the average number of time points contained by an object overall the data set.	

Note that if we compare $\log f$ and f_{avg}, it is possible that $\log f < f_{avg}$ or $\log f > f_{avg}$ because of $f_{avg} \in F_{avg} = \{2, 3, ..., f + 1\}$. The value f depends on the block size of the system setting. f could be adjusted by us. In our experiment, the block size is set to 4k and $\log f$ = 5. When the block size is set to a larger value, $\log f$ will increase. A linear increase of the block size could lead to a log scale increase of $\log f$. E.g., f = 48 $\times$ 4 and $\log f$ = 7. Theoretically, we could set parameter f (by setting an appropriate block size) such that $\log f < f_{avg}$. In practice, we have $\log f < f_{avg} \in \{2, 3, ..., f + 1\}$ in majority cases. To sum up, the time complexity for computing an arbitrary node is $O(f^h \cdot f_{avg})$ in which h is the node level (h = 1 for a leaf node).

Lemma 3 *Storage overhead*

Given an R-tree node located at level h, the storage overhead of CovNumSeqs is $O(f^h \cdot \lambda)$ in which f is the node capacity (h = 1 for leaf nodes) and λ denotes the physical storage size for time points, integers and other variables.

Proof *The storage size of a node depends on the number of time points since the counter for each time point is recorded. A node located at level h contains $O(f^h)$ data objects.* □

For a leaf node, the lower and upper storage bounds are $\Omega(2 \cdot \lambda)$ and $O(f \cdot \lambda)$, respectively. The lower bound occurs when all data objects in the leaf node have the same endpoints.

VI. UPDATE

We support processing new arrival trajectories for on-line applications. One first inserts new trajectories into the index structure and then updates CovNumSeqs. We apply the sort-based bulk load approach [6] to update the index structure. The updating task for CovNumSeqs involves two parts: computing CovNumSeqs for nodes containing new trajectories and updating CovNumSeqs for nodes in the existing structure affected by new trajectories. Specifically, the procedure works as follows: (i) a subtree is built to store new trajectories, CovNumSeqs for each node in the subtree are calculated and then appended to the list of CovNumSeqs. (ii) the subtree is inserted into the existing structure by inserting the root node of the subtree as an entry into an appropriate node of the index for the historical data. (iii) CovNumSeqs for each node in the inserting path are updated, resulting in processing both existing and new trajectories. The path starts from the root node of the historical index and ends at the node into which the subtree is inserted.

Lemma 4 *Complexity of a single update.*

Given a new arrival trajectory, we need $O(f \cdot f_{avg} + H)$ time to update CovNumSeqs. The time cost includes inserting the trajectory into the index structure and updating CovNumSeqs for all related nodes.

Proof *Assume that the new trajectory is put into one leaf node. We require $O(f \cdot f_{avg})$ time to compute CovNumSeq for the leaf node. Inserting the new leaf node into the existing R-tree needs $O(H)$ (H is the tree height) and update CovNumSeqs needs $O(H)$ as each node in the path is processed. To sum up, the overall time complexity is $O(f \cdot f_{avg} + H)$.* □

[1]Due to space limitation, the analysis of alternative methods is omitted. Details refer to our manuscript: link

Lemma 5 *Complexity of updating by bulk load.*

Given a group of $O(f)$ new arrival trajectories, we need $O(f \cdot (\log f + f_{avg} + H))$ time to update CovNumSeqs. The time cost includes (i) inserting new trajectories into the index structure and (ii) updating CovNumSeqs for all related nodes.

Proof *(i): Sequentially putting $O(f)$ trajectories into a leaf node requires $O(f)$ time. Then, We compute CovNumSeqs for the node and have $O(f \cdot (\log f + f_{avg}))$ time cost. (ii): Inserting the new leaf node into the existing R-tree needs the time $O(H)$ (H is the tree height). Updating CovNumSeqs needs $O(f \cdot H)$ as all nodes in the path are processed and each contains $O(f)$ time points for updating the counters. This is because new arrival trajectories have $O(f)$ time points overlapping with the historical data space. To sum up, the overall time complexity is $O(f \cdot (\log f + f_{avg} + H)$.* □

VII. Window aggregate queries

The basic method traverses the spatio-temporal index to look for objects fulfilling the condition. Trajectories may be split because only pieces of movements falling in the query window contribute to the result. Then, one performs the aggregation over qualified trajectories. Finally, k intervals with the minimum number of objects are returned.

A. Utilizing CovNumSeqs

We provide the algorithm in Algorithm 3, named *Win-min-Cov*. By utilizing CovNumSeqs, if a node's bounding box is contained by the query window, all objects in the node contribute to the result and CovNumSeqs are used to report the value. One does not require to traverse the index in a top-down manner to leaf nodes to retrieve data objects and perform the aggregation.

However, there are still some drawbacks that inhibit the performance: (i) The number of nodes whose bounding boxes are contained by the query window may be small because this depends on the size and location of the query window. If only a few nodes fulfill the condition, the effect of CovNumSeqs is marginal and the performance is similar to the basic method. (ii) If a large query window is issued, not only the number of processed nodes becomes large but also the partition list contains a large number of time points. Updating counters in the partition list is a costly procedure as one needs to increase the counters for all time points contained by the query interval. Data objects are iteratively processed each of which incurs updating counters at all involved time points. (iii) Calculating accurate counters is a costly procedure if a large number of objects. This motivates us to further optimize the procedure.

B. Enhancing the algorithms

We propose two strategies to further reduce the number of accessed nodes: (i) performing an approximate calculation for nodes whose spatio-temporal contents intersect the query and (ii) initializing the maxheap as early as possible, and updating the structure in a progressive way. The procedure consists of three steps, as outline in Figure 8. Step 1 initializes an extended partition list (see Definition 5) and reports a set of leaf nodes whose bounding boxes intersect the query window. We provide the framework in Algorithm 4.

Algorithm 3 *Win-min-Cov*

Input: R: a 3D R-tree;
p: the partition list on all elementary intervals;
Cov: CovNumseqs; q: query parameter.
Output: Cov_q

```
1: L ← R.Root;
2: restrict p according to q.t ;
3: while L is not empty do
4:     N ← GetNode(R, L.top());
5:     if box(N) is contained by q then
6:         update p by Cov(N);
7:     else
8:         if box(N) intersects q then
9:             if N is a leaf node then
10:                for all object N[i] do
11:                    update p by N[i] and q;
12:            else
13:                for all entry in N do
14:                    put N[i] into L;
15: Initialize Cov_q by a maxheap with the size q.k;
16: for all (t, c) ∈ p do
17:     if |Cov_q| < k then
18:         Cov_q ← (t, c);
19:     else
20:         if c < Max(Cov_q) then
21:             update Cov_q by (t, c);
22: return Cov_q;
```

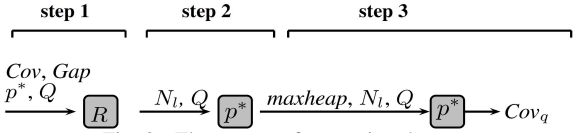


Fig. 8. Three steps of answering the query

Algorithm 4 *Win-min-CovOP*

Input: R: a 3D-Rtree; *Cov*: CovNumSeqs;
$p*$: extended partition list; q: query parameter;
Gap: an auxiliary structure
Output: Cov_q

```
1: N_l ← TraverseIndex(R, Cov, p*, q, Gap);        ▷ step 1
2: Cov_q ← Initialize(N_l, p*, q);                 ▷ step 2
3: return Refine(Cov_q, N_l, p*, q);               ▷ step 3
```

Definition 5 ***Extended partition list***

The extended partition list is denoted by p^* = <($[t_1, t_2)$, c, c′), ..., ($[t_{n-1}, t_n)$, c, c′)> in which each element maintains two counters c and c′ recording the number of objects and the number of leaf nodes in a time interval, respectively.

Example 6. The extended partition list p^* for N_r is provided in Figure 9. During $[t_0, t_3)$, all leaf nodes $\{N_1, N_2, N_3\}$ have

objects whose time intervals intersect with $[t_0, t_3)$ and we have $c' = 3$ at $[t_0, t_1)$, $[t_1, t_2)$ and $[t_2, t_3)$. During $[t_3, t_4)$, only objects in N_1 and N_3 are defined, resulting in $c' = 2$ at $[t_3, t_4)$. During $[t_4, t_5)$, only objects in N_1 are defined, resulting in $c' = 1$ at $[t_4, t_5)$.

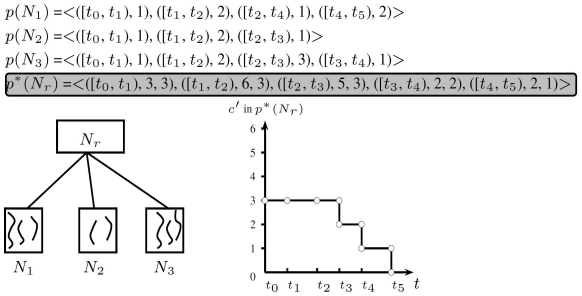


Fig. 9. Example of p^* in the root node N_r

The counter c' actually denotes the minimum number of objects in a time interval. A leaf node contains $O(f)$ objects and is counted as long as one of its data objects is defined in the time interval. Retrieving such a value does not incur any I/O cost as the node is not opened to retrieve data objects. For time intervals that do not contribute to the result, it is not necessary to obtain the accurate value if the approximate result is sufficient. We set counters in p^* for non-leaf nodes in the following two cases: (i) if the node's bounding box is contained by q, we utilize CovNumSeqs to set the counters; (ii) if the node's bounding box overlaps q, we access its child nodes.

The procedure *TraverseIndex* follows the same step as Algorithm 3 but processes leaf nodes in a different way. If a leaf node is encountered, we put it into a priority queue N_l and process the node later (opened or pruned). Nodes in the queue are increasingly sorted by time. If a leaf node's bounding box is contained by q, there will be at least one object during the query time and we update the counter c' in p^*. A leaf node may contain a *gap* meaning that there is a time interval during which no object exists. In this case, one cannot update the counter for the overall time but only these pieces of intervals at which objects are defined. The auxiliary structure *Gap* stores valid time intervals for all leaf nodes containing gaps. This is computed off-line. For on-line applications, the structure is computed when all data objects are inserted into a leaf node. This is because we can only determine whether there is a gap in the node when all data objects are inserted. If yes, the created structure is appended to the list.

Example 7. Consider a new trajectory arriving at the leaf node N_4 whose start time point is t_7, see Figure 10. There is no object between $[t_6, t_7)$. If the bounding box of N_4 is contained by the query window, there will be at least one object at $[t_3, t_6)$ and $[t_7, t_9)$ but not the overall interval $[t_3, t_9)$. Therefore, we increase c' at $[t_3, t_6)$ and $[t_7, t_9)$ rather than $[t_3, t_9)$.

Step 2 calls Algorithm 5 which processes leaf nodes in the priority queue N_l. The task includes (i) updating the value c in p^* for part of the query time, e.g., 10%, and (ii) building a

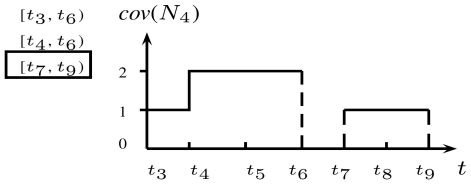


Fig. 10. Example of Gap

maxheap Cov_q with the size k. The cost of setting the counters overall the query time is high for a large query window. Therefore, we at first perform the calculation for part of the time and process the rest of them in Step 3. Since the number of leaf nodes is usually large, we do not open all of them but only take the first k leaf nodes such that there will be enough objects to initialize the maxheap. The top element in the maxheap is denoted by $Max(Cov_q)$.

Algorithm 5 Initialize

Input: N_l: a priority queue;
p^*: extended partition list;
q: query parameter.
Output: Cov_q

```
1: set a threshold t*;        ▷ for calculating the accurate value
2: while T(N_l) < t* do
3:     open each leaf node to update counters in p*;
4: Cov_q ← ∅;
5: for all p*[i].t_1 < t* do
6:     if |Cov_q| < q.k then
7:         Cov_q ← p*[i];
8:     else
9:         if p*[i].c < Max(Cov_q) then
10:            update Cov_q by p*[i];
11: return Cov_q;
```

Step 3 visits the list p^* to sum up two counters for each $p^*[i]$ during which either (i) leaf nodes whose bounding boxes intersect the time interval are opened to retrieve data objects such that the accurate number of objects is computed or (ii) the element is skipped if the number of objects from two counters is larger than the top value in the maxheap. If $p^*[i]$ is pruned according to Lemma 6, we will not access leaf nodes as the time interval cannot be in the result. Otherwise, we open leaf nodes and update counters in the maxheap. Since objects in p^* and N_l are ordered by time, they are scanned once to report the result.

The enhanced method opens a few leaf nodes to initialize the maxheap and does not access nodes (i) whose bounding boxes are contained by the query and (ii) pruned by Lemma 6.

Lemma 6 *We prune $p^*[i]$ if $p^*[i].c + p^*[i].c' \geq Max(Cov_q)$.*

Proof *$p^*[i].c$ and $p^*[i].c'$ record the numbers of objects and leaf nodes, respectively. Since there is at least one object in a leaf node, the overall number of objects is at least $p^*[i].c +$*

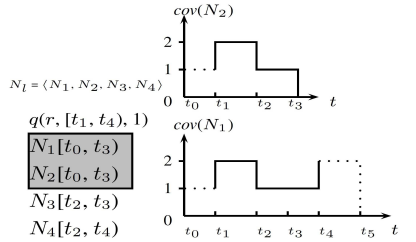
Algorithm 6 Refine

Input: Cov_q: maxheap; N_l: priority queue;
p^*: extended partition list; q: query parameter
Output: Cov_q

```
1: for all p*[i] do
2:     if p*[i] is not pruned then                ▷ Lemma 6
3:         for all nodes T(N_l) intersecting p*[i] do
4:             retrieve objects to update counters in p*[i];
5:         if p*[i].c < Max(Cov_q) then
6:             update Cov_q by p*[i];
7: return Cov_q;
```

$p^*[i].c'$, which cannot be smaller than Max(Cov_q). Thus, $p^*[i]$ will not update Cov_q. □

Example 8. Suppose that the query is $q(r, [t_1, t_4), 1)$ and leaf nodes $\{N_1, N_2, N_3, N_4\}$ are returned after step 1, as demonstrated in Figure 11. At step 2, we use $t^* = t_2$ to set the counters in p^* such that N_1 and N_2 are opened. After performing the aggregation over $cov(N_1)$ and $cov(N_2)$, we have 4 objects at $[t_1, t_2)$ and 2 objects at $[t_2, t_3)$, respectively. We process all $p^*[i] < t^*$ ($t^* = t_2$) and build the maxheap with one element ($[t_1, t_2)$, 4). At step 3, we process the elements in p^*. There are 2 leaf nodes $\{N_3, N_4\}$ whose time intervals are contained by q during $[t_2, t_3)$. The interval $[t_2, t_3)$ will be omitted according to Lemma 6. Consequently, N_3 is not accessed. The next interval in p^* is $[t_3, t_4)$. As the counter is smaller than $Max(Cov_q)$, we access the leaf node N_4 to update the counter.



(a) step 1

$N_l = \langle N_3, N_4 \rangle$
$p^* = \langle ([t_1, t_2), 4, 0), ([t_2, t_3), 2, 2), ([t_3, t_4), 1, 0) \rangle$
$\downarrow \{N_3, N_4\}$

(b) steps 2 and 3

Fig. 11. Example of the procedure

VIII. Experimental evaluation

We implemented the proposal in C/C++ and performed the evaluation in SECONDO [11]. A desktop PC (Xeon(R) E5-2620 v4, 2.1GHz, 64GB memory, 2TB hard disk) running Ubuntu 14.04 (64 bits, kernel version 4.4.0) is used.

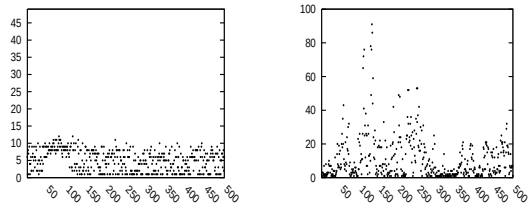
A. Setup

R-tree and Octree are built on trajectory data produced from GPS records of Beijing and ShangHai taxis [1], named BT and ST, respectively. The statistics of data sets and R-tree are reported in Table II. The R-tree node capacity is f = 49 and the split threshold of Octree is φ = 80. We also report *Count*(t) and c in the experimental data sets, which confirm the superiority of our approach ($c \gg$ Count(t), $c \gg f$). They are used to express time complexities in Table I.

TABLE II
Data sets and settings

Name	#Objects	#Endpoints	H	# Nodes	*Count*(t)	c	*Avg*($T(o)$) (sec)
BT1	663,849	6,942	4	13,980	23	2,091	84.6
BT2	1,082,450	15,487	4	22,714	33	2,257	58.58
BT3	2,908,990	44,224	4	60,777	41	2,661	58.58
BT4	5,565,884	71,235	5	124,075	49	3,974	53
BT5	11,629,616	84,223	5	242,288	53	7,130	53
ST	18,098,030	82,739	5	377,252	25	5,196	25

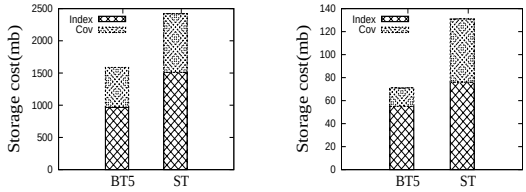
The distributions of f_{avg}. We randomly selected 500 leaf nodes and report f_{avg} in Figure 12. The results demonstrate that in most cases f_{avg} is smaller than 10 and 20 for 3D R-tree and Octree, respectively. This is consistent with the analysis in Section V. That is, the probability of an object containing a large number of time points is low. As a result, the time complexity of *Cov* in practice is $O(f^h \cdot f_{\text{avg}}) \approx O(f^h)$.



(a) 3D R-tree (b) Octree

Fig. 12. f_{avg} values over 500 leaf nodes (BT5)

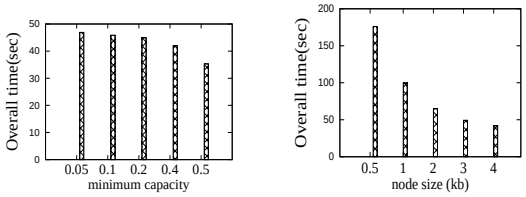
Storage size. We report the storage overhead of CovNumSeqs and index structures in Figure 13. The storage cost of CovNumSeqs depends on the number of time points contained by data objects in the node. Octree has much less storage overhead than R-tree because each non-leaf node of Octree only contains 8 entries ($\ll f$).



(a) 3D R-tree (b) Octree

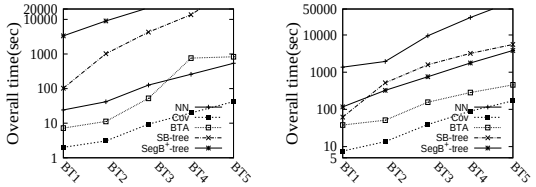
Fig. 13. Storage overhead

Effect of fill factor and node capacity. We report the experimental results in Figure 14. The performance is slightly affected by *fill factor* because the R-tree is built by bulkload. That is, the majority of nodes are *full* after creating the index except that in some leaf nodes a few objects are stored due to the spatio-temporal deviation. When the node capacity decreases, the number of nodes increases and thus the evaluation involves calculating more nodes than before, decreasing the performance.



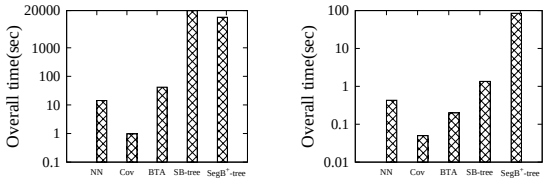
(a) Fill factor (b) Node capacity

Fig. 14. Effect of Index parameters



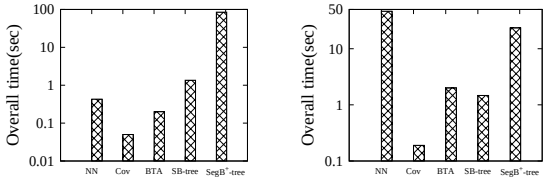
(a) R-tree (b) Octree

Fig. 15. Scalability evaluation (Beijing taxis)



(a) BT5 (b) ST

Fig. 16. Performance of querying one node (R-tree)



(a) BT5 (b) ST

Fig. 17. Performance of querying one node (Octree)

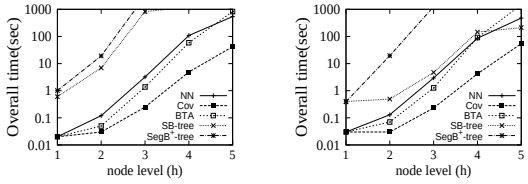
B. The evaluation of CovNumSeqs

We do the performance evaluation by comparing *Cov* with four alternative methods: (i) NN [12], (ii) BTA [5], (iii) SB-tree [25] and (iv) SegB$^+$-tree [4]. We set the root node to be the query node to compute CovNumSeqs for the overall structure. If an arbitrary node is requested, we randomly select one among the set of all nodes and report the average cost over 20 nodes as the final result.

Scaling the data size. We calculate CovNumSeqs for all nodes in R-tree and Octree and report the time costs in Figure 15. Our solution processes $O(f_{\text{avg}})$ time points ($f_{\text{avg}} \in \{2, ..., f\}$) for each aggregation operation rather than $O(f)$ in NN. In the worst cast, we have $f_{\text{avg}} = f$ but the probability is low. BTA is competitive for small data sets but the performance decreases significantly for large data sets. This is because one needs to perform the aggregation for a parent node over a list of balanced trees built from child nodes. In particular, the balanced tree becomes quite large for nodes located at high level, resulting in processing a large number of nodes and data objects. In contrast, there is no overhead of building and traversing balanced trees in our proposal. We build a partition list in advance and progressively update the structure to maintain CovNumSeqs when traversing the subtree rooted in the query node. We reduce the size of the partition list by merging time points at which the numbers of objects are the same. This enhances the performance of looking for endpoints to update counters. The approach SegB$^+$-tree cannot efficiently determine which (leaf or non-leaf) node an object belongs to (also SB-tree). One first finds data objects overlapping the query interval and then determines data objects belonging to the R-tree node. The procedure iteratively evaluates data objects overlapping the endpoints intersecting the query. The higher level the node is located at, the more data objects are processed. Another interesting finding is that the time cost for an Octree is higher than that for an R-tree. This is because an Octree leaf node contains more data objects (φ = 80) than an R-tree (f = 49).

CovNumSeq for one node. We report the costs in Figures 16 and 17, respectively. In all settings, our method achieves the best performance.

Effect of node height. We randomly selected 20 nodes at each level (if there are enough nodes, otherwise we select all) and report the average time cost in Figure 18. The time increases accordingly when nodes are located at high level. This is because the number of data objects becomes large for nodes at high level, e.g., the root node. In all cases, our method achieves the best performance.



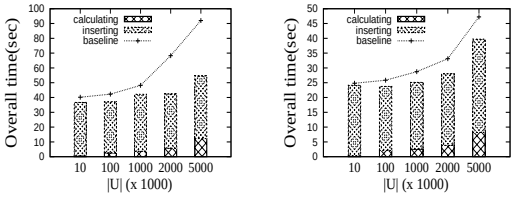
(a) BT5 (b) ST

Fig. 18. Nodes at different levels (h = 1 for leaf level)

C. Update Evaluation

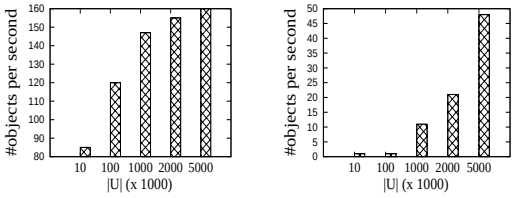
The update cost includes two parts: (i) calculating CovNumSeqs for updated nodes; and (ii) inserting CovNumSeqs of updated nodes into the existing structure. We report the time cost and the updating rate in Figures 19 and 20, respectively. The overall running time includes I/O cost and CPU time since CovNumSeqs are disk-resident structures. Majority of the time is for part (ii) because CovNumSeqs for each node in the inserting path are updated. Time intervals of new arrival objects usually overlap with time intervals of historical data. Thus, CovNumSeqs for time points contained by both historical and new objects are updated. This results in recalculating CovNumSeqs for a branch of the structure. If the historical

data size is large, updating the structure is a costly procedure. Although the number of updating objects increases several orders of magnitude, the time cost only increases marginally.



(a) BT5 (b) ST

Fig. 19. Update performance



(a) BT5 (b) ST

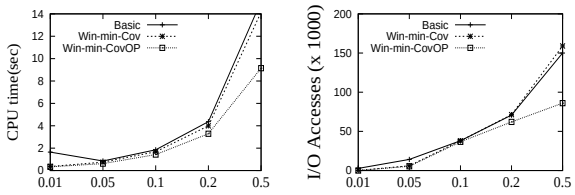
Fig. 20. Arrival rate

D. Window Aggregate Query Evaluation

We evaluate the performance by comparing with (i) *Basic* and (ii) *Win-min-Cov* [2]. The spatio-temporal query window is generated as follows. First, we determine the overall range for each dimension, that is, minimum and maximum values for x, y and t. Second, the size of each dimension in the query is determined according to the defined setting $q.r$ and $q.t \in \{0.01, 0.05, 0.1, 0.2, 0.5\}$. For example, if the overall time is 2 hours, the setting of 0.01 means a query interval representing $2 \times 0.01 = 0.02$ hours. Third, for each dimension in the query we randomly select the start point in the domain and set the end point by adding the size calculated in the second step. The value of each dimension in the query window is defined by the ratio of the query size to the size of the overall dimension and the result is averaged over 20 runs.

As provided in Figure 21, there is no obvious performance difference among the three methods when $q.r$ and $q.t \in \{0.01, 0.05, 0.1, 0.2\}$. A small query window involves evaluating a few nodes and the number of processed nodes by the three methods is almost the same. When the query window increases up to 0.5, the cost of our algorithm is almost half of the other two methods. This demonstrates that our optimization technique effectively prunes the search space for a large query window. According to experimental statistics, the ratio of nodes contained by the query box to nodes intersecting the query box is only 19% when the size of the query window is 50%. In this context, *Win-min-Cov* is sub-optimal because the number of nodes without accessing is small, which is consistent with our analysis in Section VII-A.

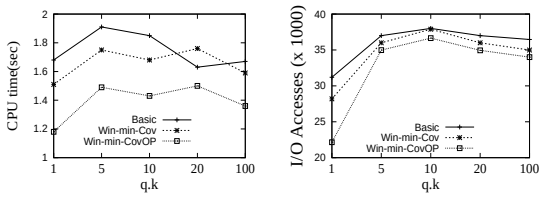
[2]The *aggregate R-B-tree* [18] can be adapted to our problem and the method is similar to *Win-min-Cov*. The aggregate information of each R-tree node is maintained by a B-tree. Entries in the B-tree are of the form (t, c) in which t is a time point (the key) and c is the number of objects at t. Two consecutive entries (t_i, c_i) and (t_{i+1}, c_j) mean that there are c_i objects between [t_i, t_{i+1}) and the value changes to c_j at t_{i+1}.



(a) CPU (b) IO

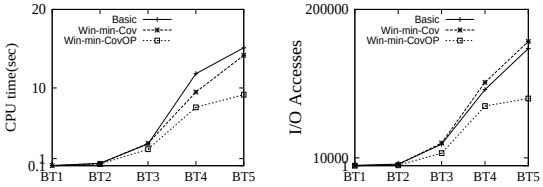
Fig. 21. Effect of scaling the query window (BT5)

We report the effect of k and scalability evaluation in Figures 22 and 23, respectively. The parameter k does not have a significant impact on the performance. The cost of all methods is not sensitive to k, but our method still achieves the best performance. The results of scalability evaluation show that our proposal outperforms alternative methods by a factor of 2 in the largest dataset (BT5 containing 11M objects).



(a) CPU (b) IO

Fig. 22. Effect of scaling k (BT5)



(a) CPU (b) IO

Fig. 23. Scalability evaluation ($q.r$ = 0.5, $q.t$ = 0.5, k = 10)

IX. Conclusion

We propose an efficient method to calculate coverage number sequences over R-tree and Octree. A thorough theoretical analysis is provided. The coverage number sequence is utilized to answer window aggregate queries. A comprehensive experimental evaluation is conducted to confirm the superiority of our proposal over other solutions. One interesting task in the future is to investigate window threshold queries that report areas containing a certain number of objects objects during the query time.

Acknowledgment. This work is sponsored by NSFC under grants 61972198 and CCF-Huawei Populus euphratica Innovation Research Funding.

REFERENCES

[1] http://factory.datatang.com/en/ (2019).

[2] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.

[3] G. Y. Chan, F. Chirigati, H. Doraiswamy, C. T. Silva, and J. Freire. Querying and exploring polygamous relationships in urban spatio-temporal data sets. In *SIGMOD*, pages 1643–1646, 2017.

[4] K. Cheng. Approximate temporal aggregation with nearby coalescing. In *DEXA*, pages 426–433, 2016.

[5] K. Cheng. On computing temporal aggregates over null time intervals. In *DEXA*, pages 67–79, 2017.

[6] J. V. den Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *VLDB*, pages 461–470, 2001.

[7] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao. Ultraman: A unified platform for big trajectory data management and analytics. *Proc. VLDB Endow.*, 11(7):787–799, 2018.

[8] H. Doraiswamy, E. Tzirita Zacharatou, F. Miranda, M. Lage, A. Ailamaki, C. T. Silva, and J. Freire. Interactive visual exploration of spatio-temporal urban data sets using urbane. In *SIGMOD*, pages 1693–1696, 2018.

[9] D. Gao, J. A. G. Gendrano, B. Moon, and et al. Main memory-based algorithms for efficient parallel aggregation for temporal databases. *Distributed Parallel Databases*, 16(2):123–163, 2004.

[10] J. Gordevicius, J. Gamper, and M. H. Böhlen. Parsimonious temporal aggregation. *VLDB J.*, 21(3):309–332, 2012.

[11] R. H. Güting, T. Behr, and C. Düntgen. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, 33(2):56–63, 2010.

[12] R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB Journal*, 19(5):687–714, 2010.

[13] M. Kaufmann, A. Amiri Manjili, P. Vagenas, and et al. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *ACM SIGMOD*, pages 1173–1184, 2013.

[14] I. López, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: a survey. *IEEE Trans. Knowl. Data Eng.*, 17(2):271–286, 2005.

[15] D. J. Meagher. Octree encoding: a new technique for the representation, manipulation and display of arbitrary 3-d objects by computer, 1980.

[16] B. Moon, I. López, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. Knowl. Data Eng.*, 15(3):744–759, 2003.

[17] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.

[18] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *ICDE*, pages 166–175, 2002.

[19] D. Piatov and S. Helmer. Sweeping-based temporal aggregation. In *SSTD*, pages 125–144, 2017.

[20] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen. Scalable top-k spatio-temporal term querying. In *IEEE ICDE*, pages 148–159, 2014.

[21] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–225, 2004.

[22] Y. Tao, D. Papadias, and C. Faloutsos. Approximate temporal aggregation. In *ICDE*, pages 190–201, 2004.

[23] Y. Tao, D. Papadias, and J. Zhang. Aggregate processing of planar points. In *EDBT*, pages 682–700, 2002.

[24] I. Timko, M. H. Böhlen, and J. Gamper. Sequenced spatio-temporal aggregation in road networks. In *EDBT*, pages 48–59, 2009.

[25] J. Yang and J.Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, 2003.

[26] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *ACM Trans. Database Syst.*, 33(2):12:1–12:39, 2008.

[27] B. Zheng, L. Weng, X. Zhao, K. Zeng, X. Zhou, and C. S. Jensen. REPOSE: distributed top-k trajectory similarity search with local reference point tries. In *ICDE*, pages 708–719, 2021.

[28] E. Zimányi. Temporal aggregates and temporal universal quantification in standard SQL. *SIGMOD Record*, 35(2):16–21, 2006.