2012 IEEE 28th International Conference on Data Engineering

SWST: A Disk Based Index for Sliding Window Spatio-Temporal Data

Manish Singh [†], Qiang Zhu [‡], H.V. Jagadish [†]

[†]*Electrical Engineering and Computer Science, University of Michigan*
Ann Arbor, USA
{singhmk,jag}@umich.edu

[‡]*Computer and Information Science, University of Michigan*
Dearborn, USA
qzhu@umich.edu

***Abstract*— Numerous applications such as wireless communication and telematics need to keep track of evolution of spatio-temporal data for a limited past. Limited retention may even be required by regulations. In general, each data entry can have its own user specified lifetime. It is desired that expired entries are automatically removed by the system through some garbage collection mechanism. This kind of limited retention can be achieved by using a sliding window semantics similar to that from stream data processing. However, due to the large volume and relatively long lifetime of data in the aforementioned applications (in contrast to the real-time transient streaming data), the sliding window here needs to be maintained for data on disk rather than in memory. It is a new challenge to provide fast access to the information from the recent past and, at the same time, facilitate efficient deletion of the expired entries. In this paper, we propose a disk based, two-layered, sliding window indexing scheme for discretely moving spatio-temporal data. Our index can support efficient processing of standard timeslice and interval queries and delete expired entries with almost no overhead. In existing historical spatio-temporal indexing techniques, deletion is either infeasible or very inefficient. Our sliding window based processing model can support both current and past entries, while many existing historical spatio-temporal indexing techniques cannot keep these two types of data together in the same index. Our experimental comparison with the best known historical index (i.e., the MV3R tree) for discretely moving spatio-temporal data shows that our index is about five times faster in terms of insertion time and comparable in terms of search performance. MV3R follows a partial persistency model, whereas our index can support very efficient deletion and update.**

I. Introduction

With increasing use of cellular devices, cellular service providers are facing the problem of improving efficiency of resource management. To provide reliable and quality services with minimum resources, they need to have a good understanding of users' usage statistics. For instance, they may need to know how the density of users varies with time at a particular location or region. To accomplish this task, they can maintain a standard historical spatio-temporal database that records users' positions and the time intervals for trhe positions. The usage statistics can be obtained through conventional spatio-temporal timeslice and interval queries. With such a spatio-temporal database, the service providers can also explore users' travel patterns, which may help them, for example, provide targeted advertisements to specific users.

However, maintaining a spatio-temporal database with a full history may lead to privacy violations since a service provider may perform extensive data mining on users' huge amount of available location information. Although users want more and better quality services, they do not want to sacrifice personal privacy of free movement. Fortunately, to determine usage statistics and provide many services to users, the service providers do not need to maintain the full history of data for each user, rather they may just need to maintain a limited history from the recent past, e.g., retaining data for the past week or month. In fact, keeping data with limited retention is also beneficial from the maintenance point of view. Since the utility of location information decreases with time, the service providers would like to have a database in which the outdated information are discarded.

Data privacy can be further improved by providing different lengths of historical information to different service providers. For example, the spatio-temporal data of the past month for all the users may be (physically) maintained at a central data repository (CDR). The various service providers are allowed access to CDR with different (logical) lenghts of history (e.g., past three days, one week, or half month) from the past month. Such access capabilities can help the system in attaining two of the ten goals of data privacy [1], namely limited disclosure (permitting users to access data with different lengths of history) and limited retention (automatically deleting the expired sensitive information).

From the aforementioned application example, we can see that there is an increasing demand to efficiently manage a spatio-temporal database with limited retention in contemporary application domains, such as wireless communication, telematics, and security services. A database management system supporting such data with limited retention should provide users with fast access to valid data as well as efficient deletion of expired data.

To capture data with limited retention, we adopt a sliding window model in this paper, which has a semantics similar to that from data stream management systems (DSMS) [2]. The limited retention is realized by the lifetime notion for data entries. Specifically, expired data entries are removed from the database as the sliding window moves forward along

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.98

with time. Different degrees of historical data for different service providers (users) can be realized by using different logical sliding windows with smaller sizes over the underlying (physical) sliding window for a given database. Unlike a conventional sliding window, the size of a sliding window in our model is usually very large and, hence, the data in the window has to reside in persistent storage devices (e.g., disks) rather than in memory due to its huge volume (e.g., customers' spatio-temporal information in the past two months for a large celluar company).

To facilitate fast access to valid data and efficient deletion of expired data for a spatio-temporal database with limited retention, we introduce a new index scheme, called SWST (Sliding Window Spatio-Temporal index), in this paper. SWST has a two-layered structure. The first layer partitions the spatial space into uniform cells. Each spatial cell has two B+ trees, as the second layer of SWST, to index its entries mainly based on the temporal information. The temporal space consists of a start timestamp dimension and a valid duration dimension.

We perform a modulo operation on the start timestamps of data entries to bound them within the size of two sliding windows. The two B+ trees are used for indexing data entries in the two sliding windows, respectively. When the second B+ tree is full, all entries in the first B+ tree are expired. All of the first B+ tree can then be safely removed, the second B+ tree becomes the first one, and a new second B+ tree can be created for the incoming entries. In this way, SWST provides a very fast (almost no overhead) lazy deletion of expired entries. It also ensures that we never maintain the amount of data greater than twice of the physical sliding window size.

To achieve improved search performance, we encode/linearize both temporal and spatial information of a data entry into a key for its B+ tree so that the spatial information can also provide some additional pruning power besides the temporal information in the second layer. Hence our index scheme does not fully decouple the temporal space from the spatial space. The encoding/linearization is done in such a way that all the entries that will be deleted together due to the move of the sliding window are placed adjacently in the B+ tree and the entries are further ordered by their spatial proximity. Since both the start timestamp (after the modulo operation) and the valid duration are bounded, the size of a key value is prevented to increase with time. Furthermore, an in-memory structure, called the *isPresent* memo, is adopted to identify those regions (cells) in the temporal space which do not actually contain any data entries so that searching for these cells can be skipped. We also classify partial and full temporal cells so that the evaluation of a query condition for data entries in full temporal cells can be avoided since these entries are guaranteed to be qualified. In addition, we also perform multi-search optimization for accessing a B+ tree in such a way that a node is never accessed more than once.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes the data model, the supported query types, and the index structure. Section IV presents our algorithms and optimization strategies for insertion, search and deletion. Section V reports performance evaluation experiments. Finally, Section VI summarizes conclusions and future work.

II. Related Work

Sliding window queries were studied extensively for Data Stream Management Systems (DSMS) in the literature [2], [3], [4], [5], [6]. However, as mentioned earlier, data in our sliding window has to be maintained on disk rather than in memory due to its large volume, which is different from a conventional sliding window in a DSMS. There are differences in terms of the supported query type, stored data and index model. Hence, most query processing techniques suggested for a DSMS become irrelevant.

To our knowledge, there are only two pieces of existing work [7], [8] that index a sliding window on disk. Both of them use a partition based strategy. The basic idea is to divide an (big) index into smaller sub-indexes so that the insertion and deletion of entries could be restricted to specific smaller sub-indexes. Here the optimization comes from the fact that the insertion and deletion would be localized to specific smaller sub-indexes, but a search may need to be performed on multiple sub-indexes. Our index scheme also employs sub-indexes, but with an optimization to use only two of them with a proper size via performing a modulo operation on the arrival time. As a result, our index scheme is able to achieve efficient insertion/deletion as well as fast searching. Indexes in [7], [8] were designed for one dimensional data and are not suitable for spatio-temporal data and queries that are supported by our index scheme. Furthermore, they have only the notion of object lifetime, i.e., the sliding window size, but no notion of valid time as our index scheme does. In fact, to the best of our knowledge, there is no existing work that maintains a sliding widow of spatio-temporal data on disk.

Existing Spatio-Temporal Database Management Systems (STDBMS) maintain full historical information, only current information, information about future trajectory, or all three types of information. There is no existing work that supports current and limited past spatio-temporal data via a sliding window, like our work does. In the existing spatio-temporal index techniques, either it is very difficult to delete the expired entries due to lack of a clear partitioning of data based on the arrival time in the index, or the search performance is low. Historical indexes such as MV3R tree [9] and 3D R-tree [10] would incur high deletion cost for removing the expired entries (MV3R currently does not support deletion). HR tree [11] and HR+ tree [12] maintain a separate R-tree for each timestamp and thus can support efficient deletion, but they are not suitable for interval queries and require very large storage space. As mentioned earlier, our index scheme supports both efficient deletion and fast searching.

Many index techniques have been developed for moving objects in the last two decades. [13], [14] give good surveys that include a brief summary and classification of these index techniques. Most of the recently proposed index schemes for moving objects use the velocity information to index the

past, present and/or the future trajectory. B(x)-tree [15] is a future trajectory index that uses the velocity information to synchronize updates to different time phases. This index has been used by Lin *et al.* [16] to index past, present and future trajectory. Pelanis *et al.* [17] have proposed a variation of TPR tree [18] (a future trajectory index) for indexing moving objects at all points in time. These methods cannot be used for discretely moving objects that are considered in our work, as they have no associated velocity.

PIST [19] and MV3R [9] are the best available historical indexes for discretely moving point objects. They both have similar query performance although they are built with quite different approaches. MV3R cannot support deletion and lacks clear partitioning of data based on the arrival time. Because of these restrictions, MV3R cannot be used for a sliding window. PIST can be modified to support a sliding window model, but it has limitations and performance issues since PIST was not designed for a sliding window. The limitations and performance issues in PIST are due to lack of support for current entries and also a strategy to split a long duration entry into multiple sub-entries. Artificial splits can make the window maintenance complex and difficult. We will discuss these issues in more detail in Section V-A. These issues are properly addressed in our index scheme.

Similar to spatio-temporal indexes SETI [20] and PIST [19], our index scheme partitions the spatial space into grids (cells) and each spatial cell has a temporal index. These indexing techniques fully decouple the spatial and temporal information in their two-layered index structures and thus cannot make use of any further spatial discrimination for entries within a spatial cell. However, our index scheme does not fully decouple the spatial and temporal spaces. The keys used for the B+ trees in the second layer of our index scheme contain both spatial and temporal information.

Initially, we had considered to adopt a current location index, such as RUM tree [21], to support the sliding window. But we did not pursue this direction further due to performance considerations. To retain only the current information, RUM tree has to keep on removing non-current entries using a garbage collection mechanism, which is an additional overhead as compared to traditional historical indexes. Moreover, to retain a limited past history in an index like the RUM tree, we need to repeatedly monitor the entries for expiration, which would be more complex. In SWST, there is almost no overhead for the sliding window maintenance.

III. Model and Structure

To facilitate the discussion of our index scheme, we use the notations given in Table I.

A. Data Model and Query Type

A two-dimensional discretely moving point object can be represented as $< oid, x_i, y_i, s, d >$, where oid is the id of the object that has a spatial location (x_i, y_i) along x and y dimensions during time interval $[s,\ s+d)$. In general, most spatio-temporal index maintain temporal information in the form of start timestamp t_{start} and end timestamp t_{end}, which means that the object oid was at location (x_i, y_i) during time interval $[t_{start},\ t_{end})$. We represent the temporal information as (Starttime s, Duration d), where $s = t_{start}$ and $d = t_{end} - t_{start}$ (> 0). For an entry whose end timestamp t_{end} is unknown at the moment, we call it as a *current entry* and assume $d = \infty$. The *valid time* of an entry $< oid, x_i, y_i, s, d >$ is the interval $[s,\ s+d)$.

TABLE I
Notations

Notation	Description
W	Sliding window size
L	Slide (sliding step size)
X^p	Number of partitions along (spatial) x-axis
Y^p	Number of partitions along (spatial) y-axis
S^p	Number of partitions along (temporal) start time axis
D^p	Number of partitions along (temporal) duration axis
Δ	Interval size along start time axis
δ	Interval size along duration axis
D_{max}	Maximum valid duration
ND	Special duration value in implementation for current entries whose final durations are unknown

To capture the limited retention of data, we adopt a time based sliding window model, which has a semantics similar to that from data stream management systems [2]. We view the aforementioned moving objects/entries as a data stream S ordered by their start timestamps. However, due to the large volume and relatively long durations, the data entries are kept on disk rather than in memory. The sliding window has a "window size" parameter W indicating its covered time length as well as a "slide" parameter L indicating the granularity with which the sliding window moves. The *lifetime* of an entry $< oid, x_i, y_i, s, d >$ is the interval $[s,\ \lceil (s+W)/L \rceil * L]$. An entry is *expired* if the current time τ is greater than $\lceil (s+W)/L \rceil * L$. Limited retention of data entries is realized via the lifetime. In other words, expired entries will never be included in a query result even if they may be still valid, i.e., $s + d > \lceil (s+W)/L \rceil * L$. Hence expired entries can be safely deleted from the database.

For the sliding window over stream S with window size W and slide L, the *queriable time period* at current time τ is the interval $T_\tau = [\tau',\ \tau]$, where $\tau' = max\{\lfloor \tau/L \rfloor * L - W, 0\}$. We define the output relation of the sliding window at time τ as follows:

$$\mathcal{R}(\tau) = \{\ \mu \mid \mu =< oid, x_i, y_i, s, d > \in S\ \wedge\ s \in T_\tau\ \}\ .$$

In other words, relation $\mathcal{R}(\tau)$ contains the entries whose start timestamps are within the queriable time period. Since the slide L can be large, in our sliding window model, the window size varies between W and $W + (L-1)$ rather than being fixed at W. Variable sliding window enables us to capture all the data from a limited past to the present without any omission.

We handle two types of historical queries, namely, (a) a time interval query, which selects all the entries from $\mathcal{R}(\tau)$ that are within a given spatial area and valid during a queried time interval within T_τ, and (b) a time slice query, which selects

all the entries from $\mathcal{R}(\tau)$ that are within a given spatial area and valid at a queried timestamp within T_τ.

To support efficient searching and deletion for data with limited retention in our model, we introduce an index scheme in the following sections. The index is designed for a given (physical) sliding window with parameters W and L, but it can also support logical windows, having a smaller window size W' (i.e., $W' \leq W$), over the physical one. W' can be specified for each user or specified as part of a query. Note that the largest query window size is kept equal to the object retention time because information beyond the object retention time cannot be queried (since it is expired).

B. Index Structure

Our index has a two layered structure, consisting of a spatial grid and a temporal index (see Fig. 1), as described below.

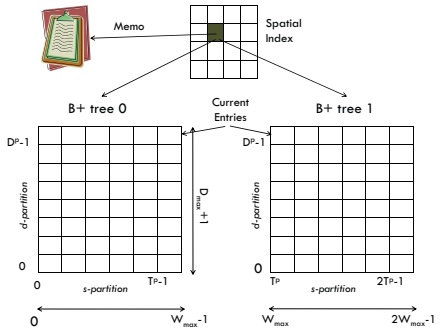

Fig. 1. Index Structure

1) Spatial and Temporal Indexes: The spatial space is divided into $X^p \times Y^p$ uniform, non-overlapping cells. Data entries are distributed into these cells based on their spatial locations. Each spatial cell has a temporal index that consists of two B+ trees. Entries in the cell are indexed in one of the two trees according to their start timestamps.

As mentioned earlier, the actual size of our sliding window varies between W and $W + (L-1)$. Let $W_{max} = W + (L-1)$. The first B+ tree contains the entries with their start timestamps in the interval $[0,\ W_{max} - 1]$, and the second B+ tree contains the entries with their start timestamps in the interval $[W_{max},\ 2W_{max} - 1]$. We perform a modulo $2W_{max}$ operation on the start timestamps of all the entries, and by this, the start timestamps remain bounded within the interval $[0,\ 2W_{max} - 1]$. When entries with start timestamps $\geq 2W_{max}$ arrive, the first B+ tree can be safely dropped, as all the entries in it would have expired. The entries with start timestamps in the interval $[2W_{max},\ 3W_{max} - 1]$ can thus be inserted into the first B+ tree. Maintaining two B+ trees enables us to remove the expired entries easily.

In SWST, spatial filtering followed by temporal filtering is better than the other two options, namely temporal filtering followed by spatial filtering or integrating the two types of filtering. As we will see later in Section IV-B, temporal filtering generates a much greater number of overlapping cells as compared to spatial filtering, and thus it would require more search. Moreover, if we perform the temporal filtering first, we cannot use simple dropping technique for removing the expired entries, which makes the sliding window maintenance very efficient in SWST (discussed later in Section IV-C). Integrating spatial and temporal filtering will significantly increase the overhead of deleting expired entries due to lack of clear temporal partitioning.

2) B+ Tree Key Computation: Keys that used in B+ trees are obtained by linearizing the temporal values, i.e., (Starttime s, Duration d), and spatial locations, i.e., (x, y), of the entries into one dimensional ordered values. For a current entry (i.e., whose final duration is unknown), we use a special value $ND = D^{max} + 1$ to represent its duration in our index scheme. The two B+ trees index two temporal spaces where start timestamp s after the modulo operation ranges in the intervals $[0,\ W_{max}-1]$ and $[W_{max},\ 2W_{max}-1]$, respectively, and duration d ranges in the interval $[1,\ D^{max}+1]$ for both spaces. We divide each of the temporal spaces into $S^p \times D^p$ uniform, non-overlapping temporal cells. We consider uniform partitioning for both spatial and temporal cells because, for a streaming sort of data, the distribution may go on changing with time. For a known and fixed data distribution, we could use better partitioning strategies such as those proposed in PIST [19]. For a given start timestamp s, duration d and location (x, y), its key is computed as follows:

$$\begin{aligned} KEY(s,d,x,y) \quad = \quad & [s\text{-}partition(s)]_2 \\ & \oplus [d\text{-}partition(d)]_2 \\ & \oplus [zc(x,y)]_2 \end{aligned}$$

where

$$\begin{aligned} &s\text{-}partition(s) = \lfloor ((s\%(2W_{max})) * S^p)/W_{max} \rfloor, \\ &d\text{-}partition(d) = \lfloor ((d-1) * D^p)/(D^{max}+1) \rfloor, \\ &zc(x,y) = \text{Z-curve value of } (x,y). \end{aligned}$$

Here $[x]_2$ denotes the (fixed-length) binary value of x and $\oplus$ denotes a concatenation. Note that $s\text{-}partition()$ and $d\text{-}partition()$ range in the domains $[0,\ S^p-1]$ and $[0,\ D^p-1]$, respectively. The top $(D^p - 1)$-th $d\text{-}partition$ contains the current entries. In our index, we assume that $S^p = \lceil W_{max}/L \rceil$ and $D^p = \lceil D^{max}/\delta \rceil$.

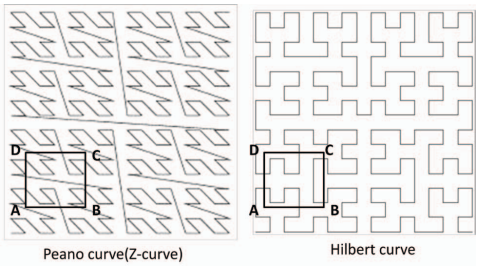

Fig. 2. Space Filling Curves

Space filling curves are used to map points from a higher dimensional space to points in a lower dimensional space. In [22], the authors have shown that both Peano (Z-curve) and Hilbert curve are effective in maintaining proximity for

two-dimensional points, i.e., both curves give a close one-dimensional mapping for points which are close in the two-dimensional space. As we will see in Section IV-B, our index requires a space filling curve to have the following property: for any given rectangle in the two-dimensional space, the lower leftmost end point should have the minimum mapping value and the top rightmost end point should have the maximum mapping value, as compared to all the points within the rectangle. For Hilbert curve shown in Fig. 2, we can see that $hc(D) > hc(C)$, which violates the required property. However, for Z-curve, the values of $zc(C)$ and $zc(A)$ are the maximum and minimum values, respectively, w.r.t. all the points within the rectangle. Hence, Z-curve is adopted here.

From the composition of our index key, we can see that the entries with closer start timestamps and closer durations are placed near to each other in the B+ trees. Furthermore, our index key contains both temporal and spatial information. The reason to include the spatial information here is to provide a further spatial pruning power via the B+ trees after the spatial grid filtering at the first layer.

3) isPresent Memo: As we will see in Section IV-B, the presence of long duration entries greatly increases the search space of a temporal index. We use an *isPresent* memo data structure to reduce the search space. The memo maintains statistical information about each temporal cell. For each temporal cell, the memo keeps a minimum bounding rectangle which covers the spatial locations of all the entries that are assigned to that temporal cell. We maintain an *isPresent* memo for each spatial cell.

The benefits of using this memo will become more clear in Section IV-B, where we discuss the search algorithms, but we can see that this kind of grid-based statistical information can be maintained only if the dimensions are bounded. If we use the start timestamp t_{start} and end timestamp t_{end} as in existing historical spatio-temporal index structures, we would not be able to divide these dimensions into grids because we cannot bound both these dimensions at the same time. By using start timestamp and duration to represent the temporal information, we can bound the temporal space, which enables us to maintain gridwise statistical information. This memo is especially helpful when there is a small fraction of entries with a long time intervals or data is skewed spatially.

For each temporal cell, we need to maintain the two end points of the minimum bounding rectangle, which requires 16 bytes of information. Thus the size of each memo would be $2 * 16 * T^p * D^p$ bytes, since in each spatial cell we have two temporal spaces with each being divided into $T^p * D^p$ temporal cells. The amount of retained statistical information does not change with the dataset size.

IV. Algorithms

A. Insertion and Updates

To insert an entry $< oid, x_i, y_i, s, d >$, first of all, we compute the spatial cell to which the spatial location (x_i, y_i) belongs. In each spatial cell, we have two B+ trees to index the temporal information. We insert the entry in $\lfloor (s/W_{max}) \rfloor \%2$-th B+ tree. The entry is inserted with key $KEY(s, d, x_i, y_i)$, as described in Section III-B.2.

For some applications, the actual value of duration d is known at time of insertion; such entries can be inserted directly into the corresponding B+ tree. Applications that support current entries may not have the value of d available at the time of insertion. They initially insert $< oid, x_i, y_i, s, NULL >$, and the actual value of d is determined later, from the start timestamp of the next entry for that object. For our implementation, we maintain current entries by initially inserting them with key $KEY(s, ND, x_i, y_i)$. When a new entry arrives, the moving object also sends the previous location information. We delete the previous entry and then re-insert it with the actual duration value. The location information of the new entries is inserted with duration value ND.

The information in *isPresent* memo is updated as we perform insertion and deletion operations in the B+ trees. Clearly, our index does not have any partial persistency restrictions like the MV3R tree. A partially persistent index allows updates only on the current or most recent entry of an object. MV3R uses this partially persistent model because it helps in separating the current and non-current entries. They have heuristics which uses this condition to support fast insertion and search. Since MV3R cannot support arbitrary deletion, it cannot support the sliding window model. The size of the MV3R index will go on increasing with time, with no systematic way to clean up the things. Our scheme is very flexible and can support update or deletion of any valid entry in the current sliding window.

B. Queries

An interval query is of the form $([x_l,\ y_l], [x_h,\ y_h], [t_l,\ t_h])$, where query's spatial area is given by rectangle $([x_l,\ y_l], [x_h,\ y_h])$ and time interval by $[t_l,\ t_h]$. A timeslice query is a special case of an interval query with $t_l = t_h$. Query evaluation is done in two stages: (a) computing the spatial overlap, and (b) computing the temporal overlap. To compute the spatial overlap, we determine the spatial cells that overlap with query's spatial area. The overlapping can be partial or full. Fig. 3 shows a shaded spatial query rectangle or area that overlaps with multiple spatial cells. For each of the overlapping cells, we need to search the corresponding temporal index for finding entries that also satisfy the temporal predicate. Computing the temporal overlap is bit

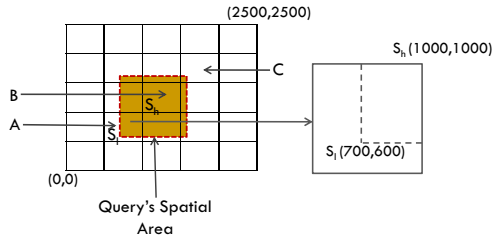

Fig. 3. Query and Spatial Cell Overlap

more complex. Temporal filtering for either type of query is done in four steps: (a) computing overlapping regions, (b) determining non-empty overlapping regions, (c) searching B+ trees, and (d) refining the search result to remove false positives. In Fig. 3, we have shown an enlarged image of spatial cell A that overlaps with query's spatial area. The query box and the spatial cell share a rectangle, with end points marked as S_l and S_h. We describe the search in the temporal index using this spatial cell.

a) Computing Overlapping Regions: The evaluations of a timeslice and an interval query differ only in this step. We first discuss the timeslice query evaluation, and then describe interval query evaluation. This step is performed statically. The result of this step will be the same for all the overlapping spatial cells.

Given a timeslice query t (i.e., $t = t_l = t_h$), we statically compute all the temporal cells in the current sliding window that overlap with $t\%2W_{max}$. Like a spatial cell, the overlaps of a temporal cell can be of two types: (a) partial, and (b) full. All the entries present in a fully overlapping temporal cell will satisfy the temporal predicate, whereas, for a partially overlapping cell, one has to do an extra refinement step to prune out the false positive entries. Determining the partial and full overlaps does not reduce the search IO, but it can greatly reduce the CPU computation time during the refinement step.

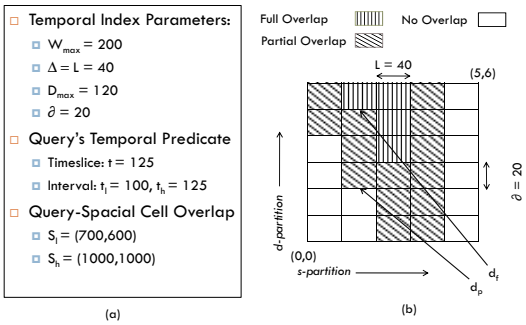

Fig. 4. Overlapping Regions

Fig. 4(a) shows an example of a temporal index and temporal query predicates. We divide the temporal space into temporal cells and each temporal cell is identified by its lower, leftmost s-$partition$ and d-$partition$ coordinates. A temporal cell with s-$partition = m$ and d-$partition = n$ can have entries with start timestamp in interval $[m \cdot L,\ (m+1) \cdot L)$ (call it $[S1, S2)$) and duration in interval $[n \cdot \delta,\ (n+1) \cdot \delta)$. Therefore, the end timestamp, $(s + d)$, of entries will lie in interval $[m \cdot L + n \cdot \delta,\ (m+1) \cdot L + (n+1) \cdot \delta)$ (call it $[E1, E2)$).

An entry with start timestamp s and duration d will overlap with a timeslice query t, if $s \leq t < s + d$, and will overlap with an interval query $[t_l,\ t_h]$, if $s \leq t_h$ and $s + d > t_l$. In our sliding window model, start timestamp s and query timestamps (i.e., $t,\ t_l,\ t_h$) should all lie within the *queriable time period* defined in Section III-A. The overlap between the start timestamp interval and the end timestamp interval of a temporal cell can be of two types: (1) disjoint (i.e., $S2 < E1$), and (b) overlapping (i.e., $E1 \leq S2$). We can divide the time interval $[S1, E2)$ into three intervals, as shown in Fig. 5.

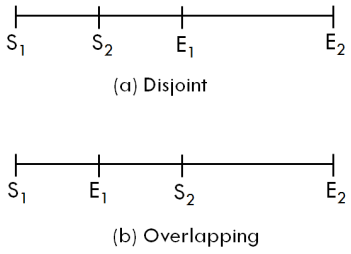

Fig. 5. Relationship between Start and End Timestamp Interval

In Fig. 4(b), we have marked the partially and fully overlapping temporal cells for a timeslice query $t = 125$. The cell with s-$partition = 1$ and d-$partition = 2$ has a partial overlap because this cell contains entries whose start timestamp s lies in interval $[40,\ 80)$ and end timestamp $s + d$ lies in interval $[80,\ 140)$. Clearly, not all the entries in this cell are guaranteed to satisfy the query overlap condition $s \leq t < s + d$, where $t = 125$. For s-$partition = 1$ and d-$partition = 5$, start timestamp s lies in interval $[40,\ 80)$ and end timestamp $s + d$ lies in interval $[140,\ 80 + ND)$ (where $ND = D_{max} + 1 = 121$). Clearly, all the entries in this cell are guaranteed to satisfy the timeslice overlapping condition. These observations can be expressed more formally through the following two Theorems.

Theorem 1: For a given timeslice query $t \in [S1, E2)$, a temporal cell C having disjoint start timestamp and end timestamp intervals has a full overlap with the timeslice query if t lies in interval $[S2, E1]$. Otherwise, it has a partial overlap.

Proof: As shown in Fig. 5(a), all entries in cell C have start timestamps in interval $[S1, S2)$ and end timestamps in interval $[E1, E2)$. The valid time of every entry in this cell contains the interval $[S2, E1]$, and thus every entry satisfies the timeslice query t if $t \in [S2, E1]$. If $t \in [S1, S2)$ or $t \in (E1, E2)$, then there may be entries with start timestamps in interval $(t, S2)$ or end timestamp in interval $(E1, t)$, respectively, which have no overlap with the given timeslice query. ■

Theorem 2: For a given timeslice query $t \in [S1, E2)$, a temporal cell C having overlapping start timestamp and end timestamp intervals will always have a partial overlap.

Proof: As shown in Fig. 5(b), if $t \in [S1, E1]$, there can be entries with start timestamps in $(E1, S2)$ which have no overlap with t. If $t \in (E1, S2)$, there may be entries whose valid time is within $[S1, t)$. Similarly, if $t \in [S2, E2)$, there may be entries with valid time within interval $[S1, t)$. ■

It is easy to verify that for a given s-$partition$, once we have a d-$partition$ that satisfies the full overlap condition for a timeslice query, all the d-$partition$s above it in the same s-$partition$ will also satisfy the full overlap condition. This is true because of Theorem 1. For a given s-$partition$, if we increase d-$partition$ value, then the interval $[S2, E1]$ of a lower

$d\text{-}partition$ is fully contained within the interval $[S2, E1]$ of a higher $d\text{-}partition$. Thus, if a timeslice query t lies within interval $[S2, E1]$ of a lower $d\text{-}partition$, then it will also lie within such an interval for a higher $d\text{-}partition$. The current entries whose start timestamp satisfies the overlapping criteria and are within the *queriable time period* will always have a full overlap.

For a given timeslice query t, one can statically compute the triplets of the form $(so_i,\ do_{ip},\ do_{if})$ for all $s\text{-}partition$s so_i such that entries in cells with $s\text{-}partition = so_i$ and $d\text{-}partition$ in interval $[do_{ip},\ do_{if})$ will have a partial overlap, entries with $s\text{-}partition = so_i$ and $d\text{-}partition$ in interval $[do_{if}, D^p)$ will have a full overlap, and entries with $s\text{-}partition = so_i$ and $d\text{-}partition$ in interval $[0, do_{ip})$ will have a no overlap. These regions are shown in Fig. 4(b). One can directly compute the values of do_{ip} and do_{if} for a given so_i. The so_i's should be within the *queriable time period* and $d\text{-}partition$'s should be within interval $[0, D^p)$. For a timeslice query we compute these triplets and pass it to the next step as a sorted list, ordered by $s\text{-}partition$ values in the ascending order.

For interval queries, we need to compute the overlapping region for the whole interval $[t_l,\ t_h]$. For this we compute overlapping regions for t_l and t_h separately, as described above, and then merge the two list of regions to generate overlapping region for the whole time interval. The merging is done columnwise (i.e., one column for each $s\text{-}partition$ in the temporal space). The merged list of overlapping column regions is also ordered by the $s\text{-}partition$ values. Given the two lists of overlapping regions for t_l and t_h, sorted by $s\text{-}partition$ values, we merge them using the following algorithm:

1) If an $s\text{-}partition$ is present in both the lists then the merged overlapping column region is the same as t_l's overlapping column region. The cells in the overlapping column region will have the same type (partial, full or none) of overlap as of t_l's.
2) If an $s\text{-}partition$ appears only in t_h's overlapping column region or is in between the overlapping column regions of t_h and t_l, then all the cells for such an $s\text{-}partition$ will satisfy the full overlapping condition.
3) For all the other $s\text{-}partition$s, there will be no overlap for the corresponding columns.

The overlapping regions for an interval query, as computed above, may contain partially overlapping temporal cells that actually are fully overlapping. Identifying as many full cells as possible is beneficial because it reduces unnecessary CPU checking overhead during the refinement step.

In Fig. 5, we had graphically shown the temporal ranges covered by a temporal cell. In Theorems 1 and 2, we had discussed the conditions for a temporal cell to have a full or partial overlap with a timeslice query t. The algorithm described above marks only those cells having a full overlap, where at least one of the end points t_l and t_h satisfies the full overlapping criteria of Theorem 1. Our next theorem shows that, even if t_l and t_h individually have a partial overlap, we can identify those temporal cells which have a full overlapping using the condition given in the following theorem.

Theorem 3: For a given interval query $[t_l,\ t_h]$ with t_l and/or t_h in $[S1, E2)$, a temporal cell C has full overlap with the interval query if query interval $[t_l,\ t_h]$ overlaps with interval $[S2,\ E1]$ (i.e., cell has disjoint start timestamp and end timestamp intervals) or fully contains the interval $[E1, S2]$ (i.e., the cell has overlapping start timestamp and end timestamp intervals). Otherwise, the cell has partial overlap.

Proof: If cell C has disjoint start timestamp and end timestamp intervals and t_l or t_h lies in interval $[S2,\ E1]$, then, according to Theorem 1, t_l or t_h has a full overlap. Thus, the interval query also has a full overlap. If both t_l and t_h lie outside interval $[S2,\ E1]$, then, according to Theorem 1, they both have a partial overlap and our merging algorithm described above would mark the cell as partial. If t_l lies in $[S1, S2)$ and t_h in $[E1, E2)$, then we can see that, even though none of the end points lie in the interval $[S2, E1]$ and individually they both have a partial overlap, the interval $[t_l,\ t_h]$ actually contains the interval $[S2, E1]$ and thus all the entries in the cell will actually satisfy the query interval. From these cases, we can conclude that, if $[t_l,\ t_h]$ overlaps with interval $[S2,\ E1]$, then we have a full overlap.

If cell C has overlapping start timestamp and end timestamp intervals, t_l and t_h will have a partial overlap (from Theorem 2), irrespective of where they lie in interval $[S1, E2)$. However, if t_l lies in $[S1, E1)$ and t_h lies in $(S2, E2]$, then all the entries must satisfy the query interval because the valid time of every entry in the cell must overlap with the time interval $[E1, S2]$. Thus, the cell has a full overlap with the interval query in this case. For all other possible locations of t_l and t_h, one can easily verify that the cell will have a partial overlap. ■

The merge algorithm described above may mark some of the temporal cells that have a full overlap as a partial overlap. For the cells that are marked partial, we can additionally check the criterion in Theorem 3 to see if they actually have a full overlap. Using Theorem 3 can maximize the number of discovered full overlap temporal cells. Note that, due to the static nature, this step of computing overlapping regions needs to be done for only one overlapping spatial cell and the result would be valid for all the overlapping spatial cells.

b) Determining Non-empty Overlapping Regions: In this step, we compute multiple small key ranges that we later use to search the underlying B+ trees in the third step. We had described in Section II, the effect of long duration entries in PIST. Similarly, the presence of long duration entries creates a large overlapping region in our index too. To mitigate the problem, we use statistical information maintained in *isPresent* memo to greatly reduce the overlapping region. In this step, we try to precisely identify the non-empty overlapping column regions and generate keys for those regions. This step shows the benefits of not fully decoupling the spatial and temporal information in our temporal index and the use of Z-curve.

Existing techniques, such as PIST and SETI, have a disadvantage due to fully decoupling the spatial and temporal

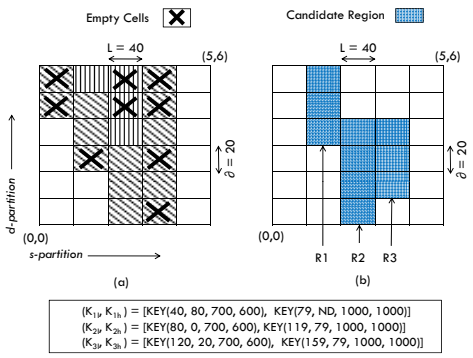

Fig. 6. Non-empty Overlapping Regions

information. For example, if we see the overlap of spatial cell C with query's spatial area (i.e., the top rightmost corner) in Fig. 3, the overlap is very small as compared to overlap with central cell B, which has a full overlap. If we do not use any spatial discrimination while searching the temporal index, then the amount of search required in a very small overlapping cell like C will be the same as that of a fully overlapping cell like B. The temporal search will generate a large number of false positive results that will have to be later removed through the spatial predicate. Our non-fully-decoupled temporal index can significantly reduce the number of false positive results by taking into account the exact amount of spatial overlap.

In Fig. 6(a), we identify those temporal cells which have an empty overlap with the query's spatial area. An empty overlap can happen either because no entry is inserted in that temporal cell or the entries in that temporal cell do not overlap with the query's spatial area. Both these types of overlap can be checked using the information contained in *isPresent* memo. We check whether the minimum bounding rectangle of the temporal cell overlaps with the query's spatial area or not. If only a small fraction of entries have a long duration or the spatial distribution is skewed, then many overlapping temporal cells can have an empty overlap. In Fig. 6(a), the cells which are not crossed may have entries that can satisfy the query predicate. The non-empty overlapping cells in a $s\text{-}partition$ column region can be merged to form a continuous non-empty overlapping column region, as shown in Fig. 6(b). The column region includes all temporal cells between the minimum and maximum non-empty overlapping cells. In other words, we remove only empty temporal cells at the top and bottom ends, or the entire column region.

After removing empty overlapping cells, let K be the number of $s\text{-}partition$s (columns) having non-empty overlaps, each of which can be represented as $(s_i,\ d_{ip},\ d_{if},\ d_{ie})$ where $i \in [1,\ K]$. Such a representation means that, for $s\text{-}partition\ s_i$, $d\text{-}partition$s in interval $[d_{ip},\ d_{if})$ have non-empty partial overlaps, $d\text{-}partition$s in $[d_{if},\ d_{ie}]$ have non-empty full overlaps, and other $d\text{-}partition$s (if any) have empty overlap. For the purpose of searching the B+ trees, we ignore d_{if}, and just consider the range $[d_{ip},\ d_{ie}]$ as having overlaps. Ignoring partial and full overlaps in searching the B+ trees does not change the number of IOs or the search procedure in the remaining steps. The partial and full overlaps only help reduce the CPU cost in the refinement step. For each $(s_i, d_{ip},\ d_{ie})$, we generate a key range $[k_{il},\ k_{ih}]$, where $k_{il} = KEY(s_i \cdot L,\ d_{ip} \cdot \delta,\ S_l(x),\ S_l(y))$ and $k_{ih} = KEY((s_i+1)\cdot L-1,\ (d_{ie}+1)\cdot\delta-1,\ S_h(x),\ S_h(y))$, where S_l and S_h are the lowest and highest overlapping spatial coordinates, respectively, as shown in Fig. 3 ($S_l(x)$ denotes x coordinate of S_l). For computing k_{il} and k_{ih}, we take the lowest and highest possible values of start timestamp, duration, and spatial coordinates of the overlapping region, respectively. At the bottom of Fig. 6, we have shown the key ranges computed for s-partition columns marked $R1$, $R2$ and $R3$ in Fig. 6(b). The advantage of using this type of key range is that it takes into account the precise spatial and temporal overlapping information to compute the search result.

The property of Z-curve that we discussed in Section III-B.2 is important because it ensures that the key generated will include all the valid entries. Z-curve satisfies that property and is also considered to be good in maintaining proximity. To the best of our knowledge, most of the commonly used space filling curves, including the Z-curve, does not ensure a strict ordering in the sense that, if we specify a rectangle and use the lowest and highest values of the points within the rectangle to form a range, the range may also cover points which are outside the specified rectangle. This property leads to false positive results, which we need to remove later in the refinement step. Using a space filling curve reduces the search space, but cannot eliminate the need of a spatial refinement.

The way we generated the keys in Section III-B.2 ensures that all the entries of an $s\text{-}partition$ will be placed adjacently in the B+ tree, and moreover the key value increases as we increase the $d\text{-}partition$ value within an $s\text{-}partition$ column. Removing the temporal cells in empty regions, as in case of $s\text{-}partition = 0,\ 1,\ 2$ and 3 in 6(a), is beneficial because we do not have to search those ranges in the B+ tree. In the case of $s\text{-}partition = 3$, even though the cell corresponding to $d\text{-}partition = 2$ is marked as empty, but we still include it within the non-empty overlapping column region. Including empty overlapping temporal cells will not affect the search if they really contain no entries. But if the cells have entries and they are marked as empty due to an empty spatial overlap, including such cells may slightly affect the search performance. Instead of generating a separate key range for each non-empty overlapping temporal cell (or segment), we chose to combine the cells (segments) in each $s\text{-}partition$ so that the number of key ranges is reduced.

If we generate the keys in the ascending order w.r.t. $s\text{-}partition$ values, the key ranges will also be sorted and disjoint. The entries would be present in the B+ tree leaves from left to right. If we search each of these key ranges individually, then we will end up traversing the same path from the root to a leaf again and again, as these ranges could be quite adjacent. In order to benefit from the fact that these

key ranges are adjacent and disjoint, we perform a multi-search optimization to speed up the search and reduce node accesses. We describe the multi-search optimization below. If the key ranges fall within one B+ tree, then all the key ranges are returned in one list, or else we create two lists, one for each of the B+ trees.

c) B+ Tree Search: Given a list of key ranges (k_{il}, k_{ih}) to be searched in a B+ tree, where $i \in [1, K]$ and $k_{ih} \leq k_{(i+1)l}$, we search the B+ tree in a level wise fashion. This ensures that we do not access any B+ tree node more than once. Furthermore, we do not access any node which has no overlap with any of the searched key ranges. The main ideas of the searching algorithm are as follows:

a We maintain two lists of B+ tree nodes. One of them contains B+ tree nodes from the current level that needs to be visited, while the other list contains the nodes from the next level that will be visited in the future.
b With each B+ tree node that we visit, we associate a list of key ranges that we need to search in that node; i.e., we associate (k_{il}, k_{ih}) with a node N, if k_{il} falls within the range of N. Note that since the key ranges are disjoint and sorted, we can do these assignments for the next level nodes while traversing the current level nodes in one pass of the key ranges. Moreover, only the highest key range allocated to a node may overlap with neighboring nodes on the right.
c If the node is a leaf node and the highest key range overlaps with the adjacent nodes, then we may use the connectors between the leaf nodes, provided the next node is not in the list of nodes that will be traversed. Otherwise, we split the key range and assign the remaining part to the adjacent node.

This algorithm is also applicable if the ranges are not disjoint, but it would require more computation or CPU time (the IO cost would remain the same).

d) Refinement: Due to the grid based index for both spatial and temporal information, the search over regions having a partial overlap needs to check whether each entry in such a region actually satisfies the query condition. The one dimensional spatial mapping, provided by the space filling curve for the overlapping bounding box, does not help in avoiding spatial refinements. As mentioned earlier, the range given by the curve may contains points which are outside the overlapping bounding box. A refinement is needed for both the spatial and temporal predicates of the query. The refinement step is not required for cells that have a full spatial and temporal overlap.

Note that SWST can be extended to support data having variable retention times which are less than the physical window size, by just modifying this refinement step. With this modification, one has to check in the refinement step if the entry has already expired. The way we drop the fully expired B+ tree in a temporal index would still remain the same.

C. Sliding Window Maintenance

Whenever the current time becomes $k \cdot W_{max}$, where $k \geq 3$, we can drop the B+ tree which contains entries with start timestamps in interval $[(k-2) \cdot W_{max}, (k-1) \cdot W_{max})$ for every spatial cell. This is because all the entries in such a tree would been have expired. When we drop a B+ tree, we also reset the corresponding *isPresent* memo entries. If required, any particular unexpired entry from any of the B+ trees can be deleted.

V. Experimental Evaluation

A. Establishing the Baseline

In Section II, we had briefly mentioned PIST [19] and MV3R [9] trees, which are currently the best historical spatio-temporal indexes for discretely moving point objects. In this section, we discuss the limitations and performance issues that arised while trying to use these index structures as a sliding window index.

PIST is a two-layered index that divides the spatial space into spatial cells and then each spatial cell has a temporal index based on the B+ tree. The temporal index is a composite index on (t_{start}, t_{end}), where t_{start} and t_{end} are the start and end timestamps, respectively. PIST requires all historical data to be available before the index is built so that it can use the knowledge of the data distribution to determine an optimal space partitioning and also the value of an optimal temporal range. If there are entries with a long temporal range, then, PIST divides the long temporal range into multiple smaller temporal ranges to maintain good query performance. Splitting an entry with a long temporal range into multiple sub-entries with a short temporal range makes deletion more complex and inefficient. In the cellular phone application described in Section I, there can be patients admitted in a hospital, who do not change their locations and thus generate entries with a long temporal range.

How long temporal range entries affect search in PIST can be understood as follows: In PIST, the overlapping condition for interval query $[t_l,\ t_h]$ is: $t_l - \lambda \leq t_{start} \leq t_h$ and $t_l \leq t_{end} \leq t_h + \lambda$, where λ is the largest temporal interval. A timeslice query is a special case where $t_l = t_h$. They search the composite index for these ranges of t_{start} and t_{end}. Clearly, the search range depends on parameter λ. In order to keep this λ value small, if there are entries with long (t_{start}, t_{end}) interval, then they split it into smaller intervals and insert all the sub-intervals into the index. This solution affects the insertion performance a bit since we have to insert multiple entries, and leads to fast search, but it can greatly affect the deletion performance, especially if one wants to maintain a sliding window. One original entry will have multiple entries in the index. If there are very few entries with a long duration and one does not allow split, then it will increase the temporal search parameter λ for all spatial cells.

In a sliding window model, data insertions and queries are interleaved. There are current entries whose t_{end} is unknown. PIST cannot support such data. One can delete or update

any entry in PIST. To maintain a sliding window (without support for current entries) in PIST, one would require to regularly locate and delete a large number of expired entries from various partitions. This would also require a lot of re-balancing of the underlying index tree.

MV3R creates R-trees for different time intervals. The temporal intervals are not pre-defined, rather they are automatically determined as the result of root splits. Due to lack of a clear partitioning for the start timestamps in MV3R, one cannot drop its R-trees which contain entries outside the sliding window. Furthermore, due to its partial persistency requirement, one cannot delete/update non-current entries in MV3R. Hence, MV3R cannot be modified to support the sliding window semantics. In MV3R, an entry with a long temporal range is also split into multiple sub-entries. Optimizations in MV3R are based on using nice splitting heuristics, and thus splits are unavoidable in MV3R.

The existing historical indexes build a complete index first and then execute a batch of queries on the index. For a sliding window index, the queries have to be within the current sliding window. This requires a workload with interleaved insertion and search operations. As pointed out earlier, PIST requires all the historical data to be available before the index is built and it cannot support current entries. Because of these restrictions, we couldn't compare SWST with PIST. We evaluated the insertion and search performance of our index with the MV3R tree, which can also support current entries. Many applications do not have a velocity associated with the moving objects, and thus we do not compare our work against the more recent velocity based indexing techniques for spatio-temporal data.

Although an efficient sliding window maintenance was the primary aim of our work, there is no existing work with which we can compare performance of SWST. Our index supports the sliding window model with almost no overhead for removing the expired entries. Because of not splitting entries with even long duration values, we can simply drop the expired B+ trees.

B. Experimental Settings

We performed our experiments on synthetic data generated by the GSTD [23] method. We used GSTD to generate a stream of two dimensional, discretely moving data points. GSTD has been used for indexing the past locations of moving objects by many access methods, such as [19], [20], [9], [11], [12]. GSTD generates two dimensional points with an object ID and the associated start timestamp. It can generate data with different distributions. A duration is determined by the difference of two consecutive start timestamp updates of an object. We used the parameters given in Table II, where values in bold denote the default values used.

For all the experiments, the initial positions and movements are uniform. We scaled GSTD data to the ranges shown in Table II. Our index performs better when the data is skewed. For skewed data, the *isPresent* memo becomes more useful. Due to the space constraint, we do not include the results for skewed data. For uniformly distributed data, memo is still useful when there is a small fraction of entries with a long duration. We have shown this in one of the experiments.

TABLE II
PARAMETERS AND THEIR SETTINGS

Parameter	Setting
Page Size	8K
Data	$X = [0 - 10000], Y = [0 - 10000]$ $T = [0 - 100000], D = [0 - 2000]$
Spatial	$X^p = 20, Y^p = 20$
Temporal	$W = 20000, L = \Delta = \delta = 100$
Query Selectivity	Spatial Extent = 0.5%, **1%**, 4% Temporal Extent = 0%, 5%, **10%**, 15%
Dataset Size	10K, 25K, **50K** objects producing 1M, 2.5M, **5M** records respectively.
Number of Queries	200

During our simulation of the sliding window, we randomly generated 200 queries within the current sliding window. These queries were then run on the MV3R tree. We compared the number of node accesses during insertion and search on both MV3R and our index. The queries were generated when the stream and index has reached steady state.

C. Insertion Performance

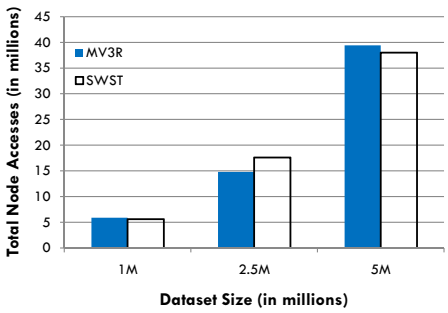

Fig. 7. Insertion IOs

As shown in Fig. 7, the total number of node accesses during insertion is comparable for SWST and MV3R. In SWST, adding every new entry (except when the last entry of the object is outside the current sliding window) requires two insertions and one deletion. In MV3R, it requires one update and one insertion.

Fig. 8, indicates the complexity difference between SWST and MV3R. As noted in [19], MV3R is quite efficient in terms of search, but it has a complex structure. PIST and MV3R have similar search performance, but the difference is in ease of implementation and complexity. The goal of MV3R is to partition the data in such a way that all the current entries are placed together. To accomplish this goal, MV3R applies a number of heuristics. Moreover, since an R-tree has overlapping search paths, one may need to traverse multiple paths during a search. Due to the simple search and split algorithms of a B+ tree, the insertion CPU time of SWST is around 5 times faster as compared to MV3R.

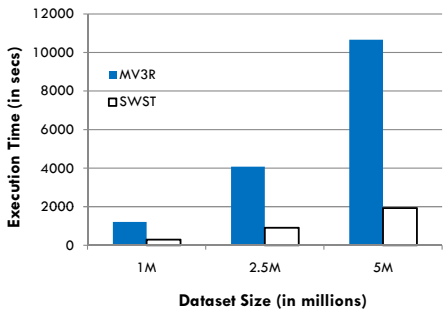

Fig. 8. CPU Execution Time

D. Search Performance

The MV3R source code [9] that we used in our experiments loads the whole index in memory before search, while our SWST is a disk based index. For this reason, in the following experiments, we compare only the average node accesses, but not the execution time.

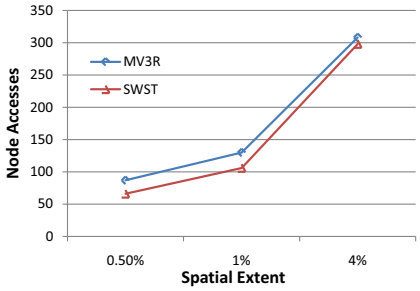

Fig. 9. Effect of Query's Spatial Extent

1) Effect of Query's Spatial Extent: This experiment was done on the 5M data set with 10% query time interval. As MV3R is based on R-tree, it performs well even for quite a large spatial extent. Its performance is low when the time interval is large, as MV3R has to query over multiple R-trees. In SWST, a large query spatial extent would overlap with a large number of spatial cells ,the temporal index of each cell has to be accessed. As shown in Fig. 9, the performance of SWST is better than MV3R till 4% spatial extent. Note that 4% spatial extent means 20% selectivity in each of the two dimensions. The performance of SWST becomes significantly better as we reduce the spatial extent. Encoding the spatial information in the key enabled us to greatly reduce the number of node accesses, even for a large query extent. Without the space filling curve, the spatial cells with very small and large query overlaps will require a similar number of node accesses.

2) Effect of Query's Time Interval: This experiment was done on the 5M data set with 1% spatial extent. We varied the time interval between 0 (i.e., timeslice query) to 15% of total temporal extent (i.e., T in Table II). As shown in Fig. 10, MV3R is more affected by the size of a time interval as compared to SWST. MV3R may need to access multiple R-trees, while SWST may need to access at most two B+ trees

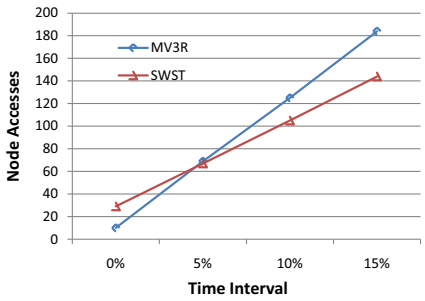

Fig. 10. Effect of Query's Time Interval

per spatial cell, even for large time interval. The performance of MV3R is better for timeslice queries because MV3R just needs to access one R-tree for such a query and, in such a scenario, it is quite difficult for a grid based approach to have a similar performance. As the size of a time interval increases (more than 4%), we can see that SWST starts performing better than MV3R.

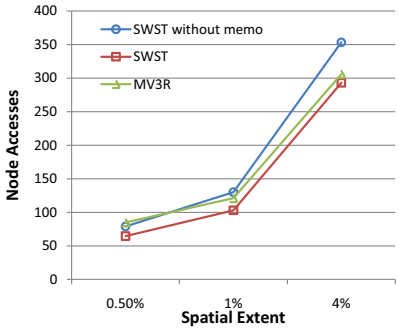

Fig. 11. Effect of Long Valid Time

3) Benefits of isPresent Memo: This experiment was designed to show the benefits of representing the spatio-temporal data as (Start timestamp s, Duration d) and of maintaining the *isPresent* memo. In this experiment, we took the 5M data set and made 4% of its entries have a long duration, i.e., the duration between $0 - 20000$. As discussed in Section III-B.3, the *isPresent* memo becomes more useful when there is a small fraction of entries with a long duration, or data is skewed. In such a scenario, the difference between overlapping and candidate regions would be quite significant. Fig. 11 shows the performance of SWST with and without the memo. We can clearly see that the memo significantly reduces the number of node accesses. In MV3R, a long duration entry undergoes a number of version splits. Thus its performance is not affected by the presence of long duration entries.

E. Effect of Parameters

In our experiments, we observed a similar effect of grid parameters as mentioned in the previous grid based indexes the SETI [20] and PIST [19]. A spatial index performs well when number of spatial cells is in the range 200-1200. If the number of spatial cells is small, then the index loses the spatial discrimination within a cell. In our experiments, the range 300-600 gave the best results. The results presented above are for

400 uniform spatial cells. Since the keys in our temporal index contain both spatial and temporal information, we observed a significant difference in the number of node accesses for spatial cells having different degrees of overlapping with the query's spatial area. Increasing the number of spatial cells increases the overhead of maintaining statistical information.

Unlike existing grid based indexes, our temporal index is also grid based. SWST's performance was affected more by the partition size along the start timestamp axis (i.e., $s\text{-}partition$) as compared to the duration axis (i.e., $d\text{-}partition$). If the $s\text{-}partition$ size was too large, then it would generate a large number of false positives. If the size was too small, then data entries which satisfy the same query condition would get quite separately placed in the underlying B+ trees, which would make the search inefficient. We know that, for an $s\text{-}partition$ column, the entries in a B+ tree are placed adjacently from left to right in the increasing order of $d\text{-}partition$ values. If the $s\text{-}partition$ size is very small, then two entries with similar start timestamps will get assigned to different $s\text{-}partitions$. This will widely separate the entries which are similar and expected to satisfy similar queries. During the search, one would have to access more number of nodes. In our experiments, we kept the window size significantly large, i.e., 20% of the total temporal length. We had 2000 temporal cells for each B+ tree. In SWST, the total space for maintaining statistical information was 25 MB.

VI. Conclusion and Research Directions

In this paper, we have proposed a disk based sliding window index for spatio-temporal data. To our knowledge, this is the first work in the spatio-temporal domain. A sliding window model is traditionally used in a data stream environment. DSMSs use the sliding window model because of the memory size constraint. In this paper, we showed that a disk based sliding window can be very useful in partially achieving two of the important goals of data privacy, namely limited retention and limited disclosure. A sliding window also helps in removing the old entries that are too old to be of any value. Unlike the previously proposed grid based historical indexing techniques (e.g., SETI and PIST) for spatio-temporal data, SWST does not fully decouple the spatial and temporal information, which leads to a greater search efficiency. Our implementation showed that SWST is better or comparable to MV3R for both insertion and search. SWST under performs MV3R for timeslice queries with a large spatial extent (which is expected as a grid based index cannot be compared with the R-tree in such a scenario), but for all other cases, SWST outperforms MV3R. SWST has an added advantage that it does not have the partial persistency constraints as in MV3R, and can support logical sliding windows. We have described an efficient way to handle entries with a long duration.

As part of future work, we plan to investigate the sliding window model for other types of queries and data. For instance, one interesting extension of this work would be to support KNN queries for sliding window spatio-temporal data. KNN queries would return the nearest neighbors from the recent past. In addition, time series data is indexed by transforming it to some other domain, such as frequency, wavelet, etc., before indexing. Time series data is generally huge, and its index lacks temporal information, which make it very difficult to remove old information. A sliding window model will be useful in comparing recent portions of time series data and can also help in removing the past sensitive information.

Acknowledgement

This work is supported in part by NSF under grant IIS-1017296. We also thank the anonymous reviewers for their time and insightful comments.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proc. VLDB*, 2002, pp. 143–154.

[2] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[3] L. Golab and M. Ozsu, "Issues in data stream management," *Sigmod Record*, vol. 32, no. 2, pp. 5–14, 2003.

[4] J. Kang, J. Naughton, and S. Viglas, "Evaluating window joins over unbounded streams," in *Proc. ICDE*, 2003, pp. 341–352.

[5] S. Viglas and J. Naughton, "Rate-based query optimization for streaming information sources," in *Proc. SIGMOD*, 2002, pp. 37–48.

[6] A. Ojewole, Q. Zhu, and W. Hou, "Window join approximation over data streams with importance semantics," in *Proc. CIKM*, 2006, pp. 112–121.

[7] L. Golab, P. Prahladka, and M. Ozsu, "Indexing time-evolving data with variable lifetimes," *SSDBM*, pp. 265–274, 2006.

[8] N. Shivakumar and H. Garcia-Molina, "Wave-indices: indexing evolving databases," in *Proc. SIGMOD*, 1997, pp. 381–392.

[9] Y. Tao and D. Papadias, "The mv3r-tree: A spatio-temporal access method for timestamp and interval queries," in *Proc. VLDB*, 2001, pp. 431–440.

[10] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-temporal indexing for large multimedia applications," in *Proc. ICMCS*, 1996, pp. 441–448.

[11] M. Nascimento and J. Silva, "Towards historical R-trees," in *Applied Computing Symposium*, 1998, pp. 235–240.

[12] Y. Tao and D. Papadias, "Efficient historical R-trees," in *Proc. SSDBM*, 2002, pp. 223–232.

[13] M. Mokbel, T. Ghanem, and W. Aref, "Spatio-temporal access methods," *Data Engineering*, vol. 26, no. 2, pp. 40–49, 2003.

[14] L. Nguyen-Dinh, W. Aref, and M. Mokbel, "Spatio-Temporal Access Methods: Part 2 (2003-2010)," *Data Engineering*, p. 46, 2010.

[15] C. Jensen, D. Lin, and B. Ooi, "Query and update efficient B+-tree based indexing of moving objects," in *Proc. VLDB*, 2004, pp. 768–779.

[16] D. Lin, C. Jensen, B. Ooi, and S. Šaltenis, "Efficient indexing of the historical, present, and future positions of moving objects," in *Proc. MDM*, 2005, pp. 59–66.

[17] M. Pelanis, S. Šaltenis, and C. Jensen, "Indexing the past, present, and anticipated future positions of moving objects," *TODS*, vol. 31, no. 1, pp. 255–298, 2006.

[18] S. Šaltenis, C. Jensen, S. Leutenegger, and M. Lopez, "Indexing the positions of continuously moving objects," in *Proc. SIGMOD*, 2000, pp. 331–342.

[19] V. Botea, D. Mallett, M. Nascimento, and J. Sander, "PIST: An efficient and practical indexing technique for historical spatio-temporal point data," *GeoInformatica*, vol. 12, no. 2, pp. 143–168, 2008.

[20] V. Prasad, C. Adam, C. Everspaugh, and J. Patel, "Indexing large trajectory data sets with seti," in *Proc. CIDR*, 2003, pp. 164–175.

[21] X. Xiong and W. Aref, "R-trees with update memos," in *Proc. ICDE*, 2006, pp. 22–22.

[22] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *KDE*, vol. 13, no. 1, pp. 124–141, 2001.

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