Lossy Space-Time Filters: An Application Layer for Storing and Querying Spatiotemporal Data

Nathaniel Wendt and Christine Julien
The Center for Advanced Research in Software Engineering
The University of Texas at Austin
Email: {nathanielwendt, c.julien}@utexas.edu

***Abstract*—Location is essential in today's mobile applications, and these applications need system support for efficient storage and retrieval of location information. Spatiotemporal data storage has received significant attention, most notably in spatiotemporal and moving object databases. However, a confluence of systems challenges (driven by constraints on bandwidth, latency, and interactivity) and societal challenges (motivated by desires for privacy) encourage alternatives to massive centralized indexes; *on-loading* responsibility for location awareness is becoming increasingly popular. We introduce the Lossy Space-Time Filter (LST-Filter), an application layer that enhances spatial data structures in support of this trend. Specifically, the LST-Filter allows a mobile device to assume complete control of storing and sharing the device's acquired spatiotemporal data—no location data is sent to any centralized index. An LST-Filter is tunable by the user or application to trade storage and computation resources for quality of location information. We show how the layer's API is tailored to efficiently respond to commonly used mobile application queries (e.g., queries about *coverage* and *trajectories*). Using real-world mobility trace data, we benchmark the tradeoffs of LST-Filters and show that, in comparison to naïve on-device location storage mechanisms, the LST-Filter can achieve drastic speedups for these common queries.**

I. Introduction

Many mobile applications rely on a detailed understanding of the relationship between the user and his immediate surroundings and how that relationship varies across space and time. Emerging social applications[1] enable highly localized interactions between users who inhabit the same or similar spatiotemporal footprints. Crowd-sourced applications like Waze[2] allow users to share their space-time data to address shared problems. Other applications[3] collect information about an individual to provide tailored analytics that capture the influences of space and time on a user's behavior. At the same time, privacy and availability concerns are pushing functionality back onto our mobile devices, in a trend termed *on-loading* [11]. The increasing availability of device-to-device communication, the enabling technologies of Wi-Fi Direct and Bluetooth Smart, and hyper-localized *cloudlet* services [19] all enable spatiotemporal sharing among devices, users, and applications co-located in space and time. These technologies encourage new applications that highlight location awareness and adaptation to bootstrap data sharing and coordination.

These situations motivate our research premise: applications demand expressive knowledge across space and time, while technology trends and societal pressures push the burden of supporting access to this knowledge onto our personal devices and highly-localized infrastructure. The increasing ability for our mobile devices to capture highly personal data that is often spatially and temporally tagged, coupled with emerging big data challenges, leads us towards on-device, personalized solutions to managing this data. Consider the following scenario.

As you leave home to begin your commute, your smart device constantly monitors your movements and habits and queries nearby devices for information matching your interests. Having knowledge of your coffee dependence and the direction you are headed, your device finds someone nearby that has recently discovered a coffee promotion. Using the person's path from the point of interest, your device determines directions for you to find the promotion. Once you arrive at work, you remain in the same general area throughout the day, making only small movements to interact with your coworkers or get something for lunch. At the end of the day, it is raining, and your device knows that you desire to take alternative transportation home. Your device locates a city bus just arriving at a nearby stop and queries to find that the bus stops near your home. It then notifies you of the estimated arrival time to the stop nearest your home based on the bus's most recent circuit.

This scenario is possible today, but the backend relies on centralized storage and processing of vast quantities of spatiotemporal data from multiple users. Not only is this increasingly infeasible (e.g., in terms of uploading data from users, indexing it, and searching it efficiently), such wide sharing of spatiotemporal data is also seen as a threat to users' privacy. However, while the humans about whom this data is generated are likely to be unwilling to share the information publicly (e.g., in centralized databases), they are often willing to share with other nearby, even unknown, users [12]. On the other hand, even as devices become more sophisticated, they will never be able to locally store *all* of the spatiotemporal information they can generate about their human users.

These challenges demand the ability to store and query spatiotemporal data located entirely on the mobile device. We present Lossy Space-Time Filters (LST-Filters), application layers that interface with space-time indexed data structures, creating LST-Structures that support advanced spatiotemporal queries. LST-Structures reside entirely on the device and the device owning the spatiotemporal data is able to locally query without the use of any external infrastructure or communication resources. LST-Filters introduce 3 key features:

[1]highlig.ht, circleapp.com, www.wechat.com
[2]www.waze.com
[3]nikeplus.nike.com, www.getchronos.com

- **Enhanced data model:** data contained within the underlying structure is interpreted not only as timestamped location data but as indicative of a probability of knowledge (PoK) within some space and time. Data *points* within the structure are interpreted as *regions* of knowledge and can represent many types of observations. This model is essential to the LST-Filter, and its computation provides the foundation for its application-level operations.
- **Expanded application layers:** the enhanced data model lends support to more interesting application-level operations such as determining the PoK of a given area or time window (Window Query), finding a path or trajectory nearest to spatial query points or within a time bound (Find Path), or inserting data guarded by opportunistic heuristics that limit the redundancy of spatial and temporal data defined within the structure (Smart Insert).
- **Tunability:** each application layer is tunable to adapt to a variety of application requirements. Realistic mobile environments represent a myriad of energy, storage, communication, and privacy requirements for which the LST Filter can be configured. For example, to support applications concerned with a high level of privacy, queries can be tuned to provide lossy summaries of data over larger regions, providing expressive representations of spatiotemporal coverage without revealing the raw location and time data. Alternatively, queries can be configured to provide a high degree of accuracy or to maximize computational efficiency. The LST Filter's configuration parameters can be tuned on-the-fly and on a per-query basis to support highly dynamic applications.

Contributions. The contributions in this paper make it possible to efficiently and expressively store the wealth of spatiotemporal data collected by a mobile device (i.e., a smartphone) entirely *on-device*. Section II reviews popular structures for spatiotemporal data and provides the setting for the description of our approach. To accomplish efficient *on-loading* of spatiotemporal data storage, we define the LST-Filter, which we describe in Section III. The LST-Structure's API includes a set of *layers* that enable creating lossy representations of a user's spatiotemporal data and also implement canonical forms of spatiotemporal queries such as *coverage* over space and time and *path discovery* across trajectories. We benchmark the performance of various LST-Structures in Section IV, then we provide application-level case studies in Section V that demonstrate the LST-Filter's applicability.

A Starting Point. We present a novel solution to efficiently *on-loading* spatiotemporal data storage—this new data structure is the contribution of this paper. The availability of this new structure opens a wealth of additional opportunities for applications to *collaborate* by opportunistically sharing the spatiotemporal data stored on devices. While our efforts are motivated by user's desires for privacy regarding their spatiotemporal data, we do not address privacy directly. Instead, by keeping the spatiotemporal data on the device, we allow users to retain complete control over their own data and with whom they share it. For the purposes of this paper, we assume a naïve strawman approach to sharing spatiotemporal data with other nearby devices. This sharing approach allows completely open sharing with all directly connected neighbor devices simply so that we can demonstrate potential applications that are enabled when individual devices are allowed to retain complete control over their own spatiotemporal data. Research opportunities for developing a more realistic sharing model are discussed in Section VI.

II. Background

The ability to store, query, and reason about spatiotemporal information is increasingly essential to mobile applications [3], [4], [25]. Increasing concerns about location privacy [18] have motivated researchers to explore privacy primitives associated with location sharing [9], [22]. These approaches focus on *location-sharing*, i.e., determining what location information to share and how to share it in a way that shields the user.

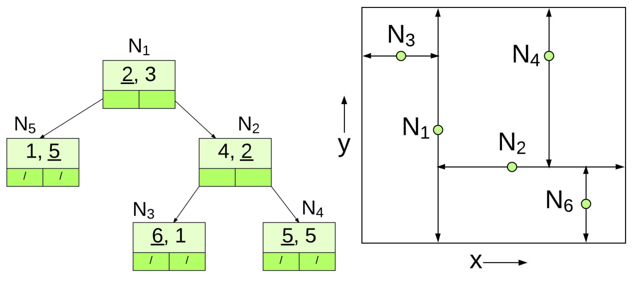
We favor *on-loading*, which has become popular for improving user experience [11], [23]. We on-load location data storage: instead of relying on a third party to store and process location information in the cloud, we push responsibility for location sensing and location data storage back onto the mobile device, both for performance (e.g., responsiveness) and for user privacy. This section investigates the history of spatiotemporal data storage and its implications on our work.

Storing Spatial Data. Storing spatial data has a rich history in image processing, geographic information systems (GIS), and robotics. Grid-based approaches, which divide space into regions and insert data into the grid square representative of the data's location, are the most straightforward [8]. Clever statistical approaches can optimize queries over this data. This type of approach is not well suited for data that is dynamic or data sets with "hot spots," i.e., spatial areas with a high number of data points. In both cases, selecting an optimal grid size is difficult, and grid bounds may not be known *a priori*.

The widely used R-Tree [10] maintains a balanced structure by representing objects within a *minimum bounding rectangle*. An R-Tree is especially useful for storing objects encompassing some area. However, maintaining an R-Tree (and avoiding worst-case query performance) requires algorithms to minimize or eliminate bounding rectangle overlap [1]. These algorithms entail higher insertion overhead and require reinserting points [1] or duplicating objects [8]. Since our primary focus is on the application layers that interface with the data structure, we use a simpler structure that lacks these drawbacks and maintains sufficient performance for storing and querying spatial information: the k-d tree [2].

A k-d tree is a binary search tree that recursively partitions space along coordinate axes alternating between k dimensions. A node in the tree represents a point that separates space along a hyperplane. Since we are using coordinate data, $k = 2$, and each hyperplane is the intersection of the inserted point alternating between the x- and y-axis; Fig. 1 shows an example[4]. The k-d tree supports our motivation because, unlike in an R-Tree, point queries are native to the structure and insertion and querying are simpler. A k-d tree can become unbalanced with incremental insertion, but our evaluation shows good performance with real-world data sets.

[4]The structure could easily expand to support elevation as a third dimension.



(a) structural representation (b) spatial representation

Fig. 1: k-d tree and corresponding representations

Storing Spatiotemporal Data. Including temporal information alongside spatial information is a natural extension and is prevalent in *moving object databases* [7]. Temporal aspects may capture a data item's relevance to the state of the database (termed *transaction time*) or relevance to the real physical world (termed *valid time*) [21]. We focus on the latter.

In support of efficient storage and retrieval of spatiotemporal data for mobile devices, methods can be loosely categorized into those that "index past positions," "index current positions," and "index current and future positions" [15]. To date, these approaches build almost exclusively on R-Trees, which come with the complexities described above. While they provide a solid foundation in centralized, heavyweight indexes, they are not well suited to the lightweight, flexible, and personalizable implementations demanded on mobile devices.

To support *on-line* location-awareness, approaches provide approximate query processing [20]. Even as this reduces the sheer volume of stored spatiotemporal data, such approaches continue to rely on a central server capable of handling the significant off-loaded storage and computation. Other approaches enable lossiness in storing the data in the first place [5], [6], for example, by storing the line segments comprising a trajectory. The most closely related trajectory simplification scheme is CDR [14], whose *online trajectory reduction* relies on the moving objects themselves (e.g., mobile embedded sensors) to store their own data for a brief time. These approaches apply only to *trajectory* data and not to spatiotemporal data in general, and they usually still rely on centralized indexing.

Work related to efficient trajectory sensing dictates how and when to activate various on-board sensors to accurately determine the device's location while incurring low energy overheads [13]. This work addresses how to acquire spatiotemporal data but does not focus on how to store that data efficiently. As such, the approach is complementary to ours. Further, this work also addresses how to efficiently summarize trajectory information (for the purpose of sending the information to a remote central server); this process is largely driven by contact with the remote server. Our approach, on the other hand, entirely on-loads the efforts associated with collecting and storing spatiotemporal data, without any access to centralized resources.

The aforementioned application-driven trends to on-load responsibility for highly personal location information necessitates addressing multiple challenges related to both storing the data and enabling expressive yet efficient querying. Enabling location data to live *on the device* requires reducing the data footprint without drastically reducing the quality of responses to queries about that information. These open challenges set the stage for our LST-Structure, described next.

III. LST-Structure

The LST-Filter is designed to interface with any spatiotemporal data structure, deemed the internal structure, supporting some key functionalities. We first overview the requirements of such a data structure and discuss our extension of the k-d tree to fulfill these requirements; other data structures that also fulfill the requirements could be used in place of our k-d tree extension. As shown in Fig. 2, the k-d tree provides basic query operations, while the k-d+ extension includes temporal sequencing. Upon this structure, we build the LST-Filter as a suite of *application layers*, resulting in the LST-Structure that provides the API for answering spatiotemporal queries.

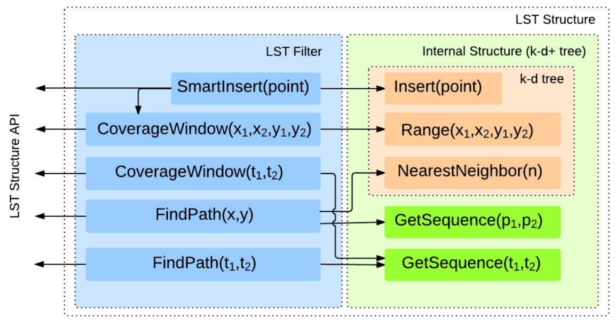


Fig. 2: LST Filter API, internal structure, and resulting LST Structure

A. Required Internal Data Structure Interface

Insert. The data structure must support inserting spatiotemporal data. Insertion should support data objects in the form $\{s_1, s_2, t_1, d_1\}$ where s is the spatial dimension, t is a temporal dimension, and d is a generic data object. We are only using two spatial dimensions for this study, but the LST-Filter can be easily extended to support more spatial dimensions.

Range Query. A key feature of a supporting data structure is the ability to query a spatial region, most commonly rectangular, for points that exist within the bounding range. For example, a query may ask for all data points within a 100 square meter area comprising a construction site. This is a common feature of data structures supporting spatial indexing and is essential to the effectiveness of the LST-Filter; as such, optimizing efficiency of this query is crucial.

Nearest Neighbor. Another required type of query is finding and returning the N nearest neighbors given a target location. Such queries are commonly made with $N = 1$. Such a query may ask for the data item nearest an historical landmark.

Delete. While deletion is typically not trivial in spatiotemporal data structures and extension of this operation is beyond the scope of this study, further work with the LST-Filter will include trimming stale or otherwise irrelevant values.

Get Sequence. The data structure must be able to return a time-ordered sequence of data points. This query may be in the form of $Q\{t_1, t_2\}$, where t_1 and t_2 are upper and lower time bounds, and all points within this time range are returned in order. The query may also be in the form $Q\{p_1, p_2\}$, where p_1 and p_2 are spatial data points previously inserted into the structure, and the query returns the sequential list of data items between these two points. An example query may ask for a sequence of data points between two times on a specific day or between two existing data points representing bus stops.

B. *k-d+ Tree*

We use the k-d tree as the underlying data structure to store location coordinates as keys and observations as values; an example k-d tree and its associated spatial data are shown in Fig. 1. For our purposes, each observation has (at a minimum): x.coord, y.coord, and timestamp. To account for the temporal sequence requirements, we embed a doubly linked list into the tree to allow sequential traversals without requiring data duplication. We call the k-d tree with this embedded linked list a k-d+ tree (Fig. 3). Each time a new data point is inserted, not only are the tree's left and right subtree references (which depend on locations) maintained, but the inserted node is also connected to the most recently inserted data item; this model assumes that node insertion is performed in chronological order, which is natural in a location-sensing application. The doubly linked list allows both chronological and reverse chronological traversals.

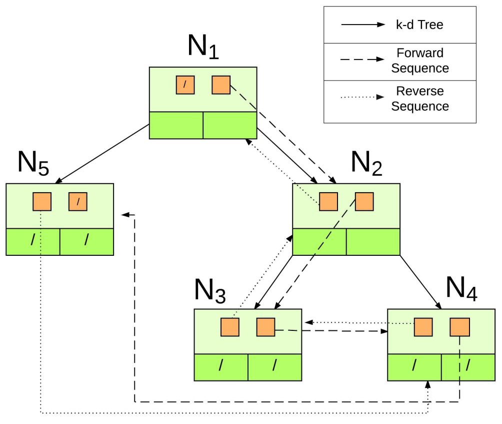


Fig. 3: k-d+ tree

C. *LST-Filter*

Each piece of spatiotemporal data we are interested in storing is an *observation*. An observation, such as viewing a "free cup of coffee" promotion sign, is made at a given location and a given time. The farther in space and time one is from the genesis of the observation (i.e., the data *point*), the less relevant the observation is. Many different types of observations can be made, and the LST-Structure must store data in a manner that supports useful matching (e.g., a single LST-Structure may provide the spatiotemporal information for all of the application-level queries described in Section I, with data about buses, coffee shops, and the user's trajectory). For simplicity, we focus on just the space-time portion of the data objects. This can be viewed as every data item containing the same observation (and we omit the observation itself from the

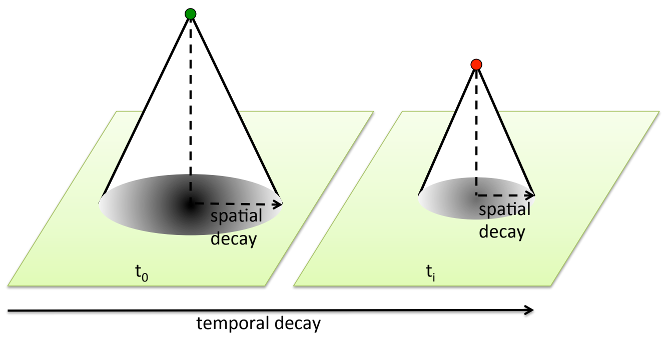


Fig. 4: Impact of spatial and temporal decays on PoK

discussion). We revisit more general purpose uses in Section V when we give example applications.

Building on the operations in our internal data structure (k-d+ tree), we introduce the LST-Filter, which establishes an API for answering important spatial and temporal questions and for controlling the contents of the spatiotemporal data store (see the outermost layer in Fig. 2). This API is tunable through configuration parameters that can be adjusted to match the current state of available resources, for different users, and for a particular user's situation or changing environment.

Probability of Knowledge (PoK). A basic question an application may ask is of the form, "What is the *coverage* of a given area or window (with respect to an observation)?" or "How much do I know about a given area?" The underlying data structure's *range query* returns a set of "candidate" points within a specified window bounding area. The LST-Filter coverage query processes these points in terms of their spatial and temporal relevance. To do this, we introduce a *probability of knowledge* (PoK) model. For each point p in the two-dimensional space, $\mathrm{PoK}(p, t)$ represents the probability that p is "covered" by the observations stored in an LST-Structure at a specified time t. For example, as our commuter's device attempts to plan a route home for him on the bus, the device may ask whether the bus's trajectory "covers" his workplace (i.e., whether the bus comes close enough to the workplace to be convenient). Fig. 4 shows this model pictorially. At the exact time and place of the observation (the center of the oval on the plane on the left of Fig. 4) the PoK is 1. As we move away from that point in space (towards the bounds of the oval) or time (from the plane depicting the world at time t_0 to the plane depicting the world at time t_i), the PoK degrades. The SPACEDECAY associated with the data item causes its influence to decay to some boundary, at which point it falls immediately to zero. We refer to the boundary as SPACETRIM; beyond the boundary, the point has no influence. One can think of SPACEDECAY as the slope of the cone in Fig. 4.

SPACEDECAY and TIMEDECAY are determined on per data item and/or per operation. That is, spatiotemporal observations are not inserted into the LST-Structure with spatial and temporal decay information; instead this information is computed each time an application executes a query, based on the application's interpretation of the spatiotemporal data.

Coverage Window (PoK). In addition to asking about a specific point in space, a coverage query may also ask about the coverage of an area. For example, the commuter may not need to be guaranteed that the bus will come exactly to his workplace, but he may want a reasonable probability that it comes within 100 meters. We refer to this challenge as computing a *coverage window*. Given a region (specified by a rectangular box in the two-dimensional space), a coverage window query returns the PoK for the region, which depends on both the spatial and temporal influences of the data items in the tree. To compute the PoK value for a region, we divide the region into a grid, compute the values for the individual grid squares, and aggregate them. This allows us to build our coverage window query out of successive calls to the range query of the underlying k-d+ tree. The size of a grid square impacts both the quality of the result and the speed with which it can be computed; a smaller grid square results in "higher resolution" but requires more operations on the underlying data structure. We evaluate these tradeoffs in Section IV.

Computing a coverage window is further complicated by the fact that multiple data items stored in the tree may influence the PoK at a given point. For example, as shown in Fig. 5, data items A, B, and C all share common influence area. Computing the PoK within any of these intersection areas involves resolving the shared influence between all relevant data points. Given a grid square, identified by a point at its center (the *target point*), we retrieve all of the nearby neighbors of the target point (the *candidate points*) using the k-d+ tree's range query. We compute the PoK of the target point using the *inclusion-exclusion* principle to account for "double counting." Consider the (simplified) example shown in Fig. 5. We cannot simply "union" the influences of the three target points' spatial zones (A, B, and C). Instead we must subtract off the pairwise overlap and then add back the space in which all three overlap.

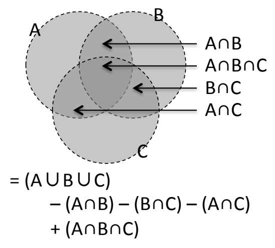


Fig. 5: Inclusion-exclusion and spatial influence

This becomes more complicated as an increasing number of candidate points are considered. In general, the following expression captures the PoK at a target point p, where K_i is the actual *knowledge* at each of the candidate points $1 \leq i \leq n$. $\mathbf{P}(K_i)$ reflects the distribution for point i after accounting for both the spatial and temporal decay as depicted in Fig. 4.

$$\mathrm{PoK}(p) = \mathbf{P}\left(\bigcup_{i=1}^{n} K_i\right) \tag{1}$$

$$\begin{aligned}\mathbf{P}\left(\bigcup_{i=1}^{n} K_i\right) = {} & \sum_{i=1}^{n} \mathbf{P}(K_i) - \sum_{i<j} \mathbf{P}(K_i \cap K_j) \\ & + \sum_{i<j<k} \mathbf{P}(K_i \cap K_j \cap K_k) \\ & - \ldots + (-1)^{n-1}\, \mathbf{P}\left(\bigcap_{i=1}^{n} K_i\right)\end{aligned} \tag{2}$$

PoK values are between 0 and 1, inclusive, and the set intersection operator in Equation 2 generates the combined probability for the considered candidate points, which cannot exceed 1. Using Equation 1, we compute the PoK for each grid square. We then compute T_o as the sum of the PoK values for all of the grid squares and T_P as the maximum possible PoK (equivalent to every grid square having a PoK of 1). We return the coverage window as the fraction T_o/T_p.

To compute the value for each grid square, one must consider *all* points to be candidate points (i.e., any point, regardless of how distant it is in space and time, exerts *some* influence on every other point). Because these influences eventually become vanishingly small, we allow coverage window queries to control their influence. This is the purpose of the SPACETRIM parameter introduced earlier; points whose influence has been reduced to a value below SPACETRIM are not considered in the PoK computation; a lower value of SPACETRIM allows more distant points to be included. TRIMTHRESH limits the number of points in a different way: once the algorithm has found TRIMTHRESH number of nearby points, it excludes any points that are more distant.

This description assumes that the algorithm performs the underlying range query on the k-d+ tree for each grid square then post-processes the results to identify the appropriate points and compute the PoK. A more efficient approach, *CoverageOpt*, builds a balanced reference k-d+ tree from the list of candidate points retrieved via the original k-d+ tree range query. During the computation for each grid square, the algorithm queries this reference tree to determine all of the nearby points instead of checking against all possible candidate points. This is the algorithm in Fig. 6, where x and y are indices over a two-dimensional array of the grid squares in the target region, and GetTileWeight computes the PoK for a given grid square, according to the inclusion-exclusion principle, also using the specified SPACEDECAY. TimeRelevance in Fig. 6 relies on the TIMEDECAY parameter.

```
totalWeight = 0;
kdPlusTree = BuildTree(points);
foreach x grid square do
    foreach y grid square do
        validPoints = kdPlusTree.range($x$,$y$);
        foreach validPoint do
            dist = CalculateDist($x$,$y$,point);
            if dist < SPACETRIM then
                PoK = dist * TimeRelevance(point);
                add PoK to nearbyList;
            end
        end
        nearbyList.trim(TRIMTHRESH);
        weight = GetTileWeight(nearbyList,$x$,$y$);
        totalWeight += weight;
    end
end
return totalWeight / maxWeight
```

Fig. 6: Coverage computation with *CoverageOpt*

The above discussion focuses on the spatial aspects of a coverage window query, but coverage window queries can also be performed over time windows. When the application provides a time based window query, the set of points considered is reduced to those that fall within the specified start and end times. To compute a time-based coverage window, we simply walk the linked list of spatiotemporal objects to find the start time, and then continue until we find the end time. The

PoK is then computed from the set of data items that fall in between. Of course, the most interesting queries are those that combine spatial and temporal aspects. This general form of the algorithm is basically a two pass filter: before the algorithm builds the reference k-d+ tree from the candidate points, it filters the candidate points to be those that fall within the time region (or, more specifically, those points that can influence the time region, which may also contain observations whose genesis is just before the start time of the time region). The remainder of the algorithm in Fig. 6 remains the same.

In summary, coverage window queries operate over two basic parameters: the rectangular spatial region and the time region that, together, define the "window." The parameters described previously (TIMEDECAY SPACEDECAY, SPACETRIM, and TRIMTHRESH), allow the data structure client to manipulate each individual coverage window query.

Smart Insert. Maintaining a coverage map of constantly moving objects generates large structures that take a long time to query. Many data items are redundant in space, time, or both, and maintaining them often does not add much information. Recall that the commuter in our motivating example stays in a similar location throughout the entire day at work. Many mobile location services only update observations if the mobile object has moved a significant distance. This does not account for the temporal relevance of data objects, nor does it account for motion paths where an actively mobile user is constantly looping back on already "covered" areas.

We introduce *smart insert*, which uses application tunable guidance to ask "how well is this information already represented?" before inserting a data item. Upon calling smart insert, the application provides an INSERTTHRESH parameter that accounts for both space and time by providing a threshold on the PoK that the data item must meet in order to be inserted. For example, an INSERTTHRESH value of 80 means that, if the PoK for the point in the current data structure is 0.8 or higher, then the new data point should not be inserted. Lower thresholds result in a greater reduction in the number of data items inserted, trading some degree of accuracy for storage and computational efficiency. Clearly, this behavior also relies on SPACEDECAY and TEMPDECAY, which can be defined for individual calls to smart insert.

Find Path. A primary motivation behind the LST-Structure is to enable queries about a mobile user's *trajectories* through space and time. For example, our commuter encounters another pedestrian who has recently visited a coffee shop advertising a promotional discount. One option would be to collect the coupon and then search the Internet to find directions. This approach requires communication and data resources, and it also reflects a statically computed path as opposed to the current "best" path as observed by other *in situ* users, which may account for traffic, weather, construction, and other dynamic aspects. Instead, our commuter can query the other pedestrian's mobility data, requesting the trajectory through space between the current location and the coffee shop. The find path operation takes two pairs of x, y coordinates. Under the hood, the operation first finds the nearest neighbor of each point in the LST-Structure, then uses the k-d+ tree's get sequence method. As a second example, when our commuter steps on the bus on the rainy afternoon, the device can query the bus's prior "loop," match the spatial information of the loop with the commuter's home address, and compute the expected route time in today's current rainy conditions, giving the commuter an accurate estimate of his arrival time. This scenario uses the temporal version of find path, which is simply a wrapper around the k-d+ tree's get sequence method.

A challenge in many of the operations on an LST-Structure is setting the parameters to achieve desired application behavior. We next report extensive benchmarking of the parameters, homing in on a well-suited set of defaults; future work will provide better information to the data structure's users about the semantic relationships between these parameters and the state of the physical environment and its mobile objects.

IV. EVALUATION

We performed a series of *benchmarks* on the LST-Structure, comparing it both to itself with different settings and to other less expressive structures. We used two datasets of real mobile data from CRAWDAD: (1) the NSCU-KAIST collection of 92 traces of 500-2000 GPS data points each, collected at a university campus in South Korea [17] ("Mobi"); and (2) a set of vehicular traces of taxicabs in the San Francisco area with 500 taxicabs over 30 days [16] ("Cabs").

We used two computational environments: a desktop (Intel Zeon 3.0 Ghz Quad-core, 3GB RAM, Ubuntu 12.04) and a mobile device (Samsung Galaxy S4 mini (Qualcomm Snapdragon 400 chipset, Dual-core 1.7 Ghz, 1.5GB RAM), Android OS, v4.2.2). Determining execution times in Java can incur unreliability due to the nature of the JVM's JIT compiler and variations in system behavior. We mitigate these concerns by following industry guidelines[5]. We performed all benchmarks on quiescent machines and ran warmup tests on the JIT compiler until iterations of the same computation were within 3% of each other. We performed tests multiple times and over a large number of traces to reduce noise. Despite these measures, execution times should not be taken as absolute performance measures but rather as reasonable estimates of performance and as comparative exercises. Unless we specify otherwise, we refer to desktop tests. In the Mobi dataset, when the data items are generated by and representative of pedestrians, we set the default spatial radius of each data item to 30m; in the Cabs dataset, we set the default spatial radius of influence to 1km. The default setting for TIMEDECAY in Mobi is 8X, representing data that is temporally relevant longer (i.e., changes more slowly); in Cabs we used a default TIMEDECAY of 3X since the dataset represents more rapidly changing (vehicular) data. Our default TRIMTHRESH is 10 data points and INSERTTHRESH is 80 (requiring a PoK less than 0.8 before a new element is inserted).

The following evaluation is framed under the assumption that reducing execution time will reduce device energy consumption. To that end, we demonstrate the tunability of the LST-Filter to provide execution time improvements by varying query parameters as well as limiting the size of the structure.

Coverage Window Queries. We examine several configurations of the coverage window query parameters to assess performance and accuracy tradeoffs. We quantify the loss in accuracy as the change in reported probability for the queried

[5]http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html

window in comparison to a baseline, ground truth query. We set the window bounds to include the entire observed space ($\sim$1000 km^2 for Mobi and $\sim$2000 km^2 for Cabs). These coverage windows are very large and serve to intensively test the implementations for comparison. In application settings, window query sizes will likely be much smaller.

We first quantified the performance improvement from the *CoverageOpt* calculation, which utilized a second, smaller, balanced k-d+ tree for internal processing. The optimized algorithm improved performance by 50X on average while losing only 0.34% accuracy. This optimization proved so beneficial that it is included in all following evaluations.

We then evaluated the impact of TRIMTHRESH and SPACETRIM on coverage window queries. In addition to the default settings, we report results for a *Safe* version (SPACETRIM = 0.1; TRIMTHRESH = 15) and an *Aggressive* one (SPACETRIM = 0.4; TRIMTHRESH = 5). Smaller values of TRIMTHRESH curtail the set of candidate points that influence a PoK computation, while a smaller SPACETRIM broadens the spatial area in which candidate points can reside.

Increasing SPACETRIM and decreasing TRIMTHRESH improves performance (Fig. 7). What was surprising was the magnitude of the performance gain of the small change between *Safe* and the default settings, with only a small loss in quality (i.e., a loss of 4.3%). The loss in quality from the default settings to *Aggressive* was more substantial and at a much lower relative benefit in terms of execution time gains.

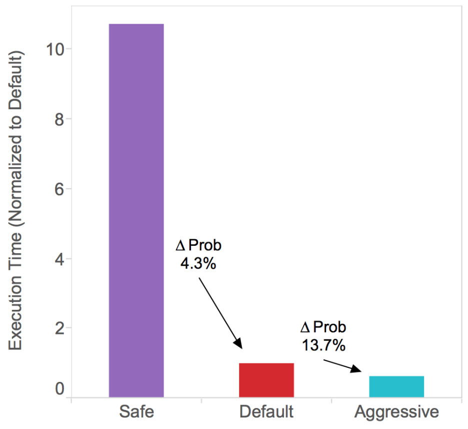


Fig. 7: Window coverage execution vs. settings

Another parameter we investigated for window coverage is the size of the x, y grid. We used four settings for each data set (Table I). As shown in Fig. 8, coarser grained grids drastically improve execution time; since the calculation for each grid space can be computationally expensive when many data points are concerned, limiting the number of grid spaces greatly reduces execution time. These speedups come at a very small degradation in the quality of the result; this bolsters our use of the grid-based approximation heuristic for computing aggregate coverage.It is also supportive of the resource constrained mobile devices that we target, where managing computational efficiency of common operations is crucial for limiting the energy footprint.

Data	**Name**	x x y (m^2)
Cabs	Accurate	44 x 55
	Default	88 x 110
	Optimized	440 x 550
	Speed	1110 x 880
Mobi	Accurate	1 x 1
	Default	5 x 5
	Optimized	25 x 25
	Speed	50 x 50

TABLE I: Grid square sizes

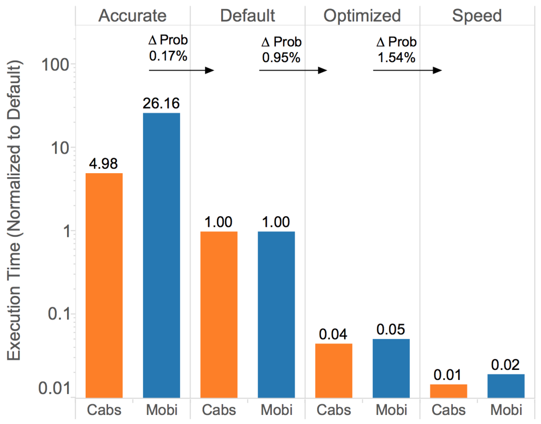


Fig. 8: Average Execution For Varying XY Resolution Configurations. Note the log scale on the y-axis.

Our final coverage window benchmark analyzes an LST-Structure with a naïve internal structure implemented as an array of points. While the array implementation is not common in (large) moving object databases, it is seen in simpler trajectory traces common to on-device mobile applications. Fig. 9 shows that the LST-Structure with a k-d+ internal structure was was 7X to 10X more efficient than the array internal structure (again, note the logarithmic scale).

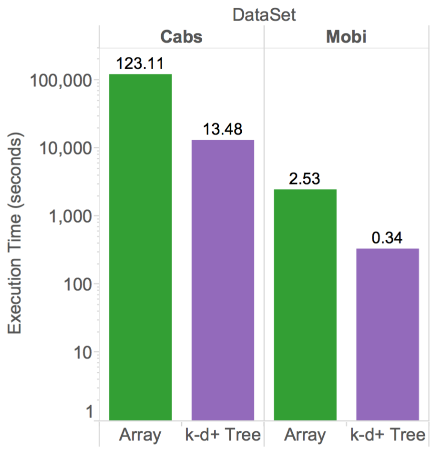
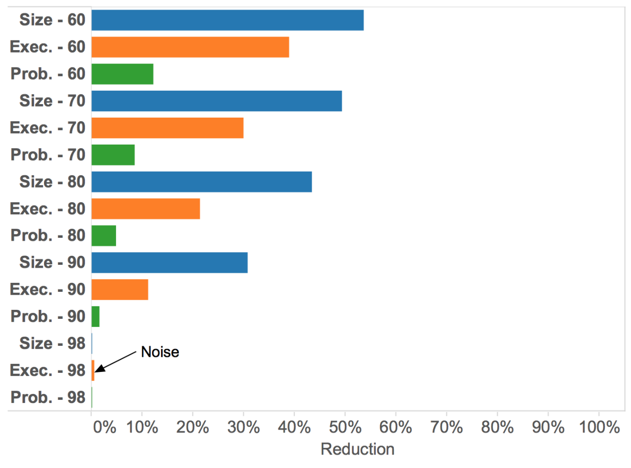


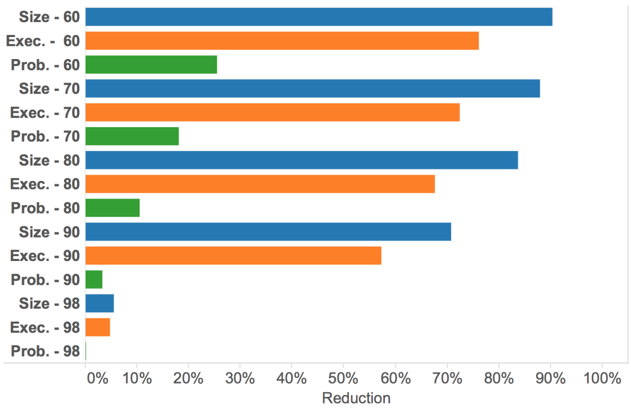
Fig. 9: Average execution time for array vs. k-d+ tree backed LST-Structures. Note log scale on y-axis.

Smart Insert. The LST-Structure's smart insert allows clients to control the degree of data(and resulting query accuracy) loss. We performed a sweep across many INSERTTHRESH values to characterize the quality of coverage window probability computations, the size of the LST-Structure, and execution time tradeoffs. Fig. 10 shows the results. Lower values of INSERTTHRESH are more aggressive, but they trade the performance increases and size decreases for sometimes significant losses in quality. While limiting the probability loss to within 10% of the ground truth, smart insert reduced Mobi by ~50% in size and Cabs by ~85% in size, which resulted in ~20% and ~70% reductions in window query execution time, respectively. It can be seen from the characterization of the two real-world data traces that the tunable parameters of smart insert are clearly application-dependent (e.g., "mission critical" applications may not be able to sacrifice quality, while purely "social" applications may favor device lifetime).

The LST-Filter's tunable smart insert is very useful for reducing the size of the spatiotemporal data structure without substantial losses in expressiveness (i.e., the quality of the data) and resulting query accuracy. Reducing the size of the data structure is crucial for decreasing storage requirements since we intend on using comprehensive data on storage-constrained mobile devices. Additionally, query execution times will be improved, which is important for responsive user feedback and for maintaining low energy consumption.



(a) Mobi



(b) Cabs

Fig. 10: Average reduction in structure size, execution time, and coverage window quality with smart insert

Find Path/Trajectory. While the execution time analyses for find path were not particularly interesting, find path was able to successfully determine paths when provided either two time ranges or two points. Simply querying a path between two spatial points was ambiguous in cases where similar spatial regions were visited at varying times, e.g., in a loop. Combining temporal and spatial components of find path into a single form i.e., requiring all points on the path to not only be between two points in space but also between the start and end time) can finely control this behavior.

Mobile Platform. We next turn our attention to experiments on an actual mobile platform to ascertain whether our LST-Structure is suitable for running "on-device." We used the same defaults, with the exception of setting the grid sizes to be those listed for *Speed* in Table I, since the performance boost was so significant and the quality loss quite small. It bears repeating that our coverage window queries are for the entire space; in real applications, coverage windows are likely to be much more focused; we maintain large windows for consistency with the previous results. Our goals are twofold: (1) ascertain whether smart insert's added overhead is reasonable and (2) measure how well coverage window queries perform on the LST-Structure with smart insert.

The overhead for smart insert in the Mobi data set is shown in Fig. 11 for four different INSERTTHRESH values; we compare against an LST-Structure without smart insert. This "standard" insert is much more efficient since smart insert must perform a coverage window query before inserting. The execution time for insertion increases with INSERTTHRESH because higher values of INSERTTHRESH cause more data elements to be inserted into the structure, making the later calls to coverage window query from smart insert more expensive. Despite these trends, smart insert with INSERTTHRESH of 90 is still a very reasonable 8.709 ms per insert on average.

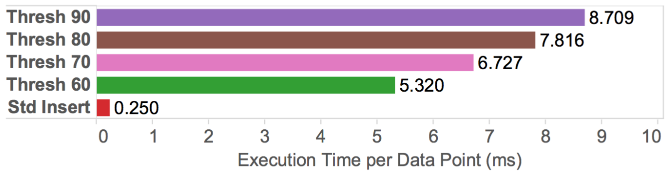


Fig. 11: Android execution for varying INSERTTHRESH values. Averages across all Mobi inserts

We also performed coverage window queries and compared the execution time for the array LST-Structure and the k-d+ LST-Structure with and without smart insert. Fig. 12 shows the results. For Mobi, the k-d+ with smart insert demonstrated a nearly 3X speedup over the array and a 35 ms speedup (per coverage window query) over the standard k-d+. For Cabs, the k-d+ with smart insert demonstrated a 3.4X speedup over the array and a 2.9 second speedup over the standard k-d+. This is consistent with the smart insert performance evaluation above, and these numbers are convincing evidence that, with real data, the LST-Structure is a viable structure for storing and accessing spatiotemporal data on a mobile device.

V. APPLICATIONS

We next demonstrate how LST-Structures can be incorporated into applications, using the scenario in Section I. Our

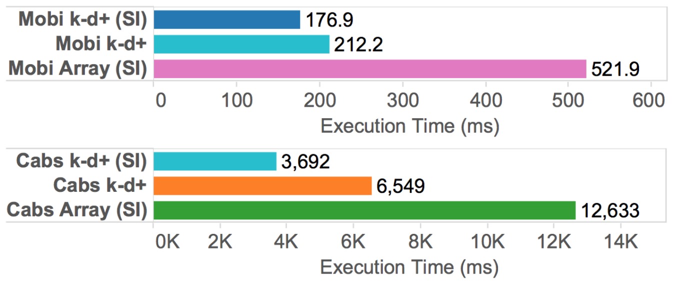


Fig. 12: Android coverage window execution.

implementation is publicly available[6].

Bus Path. Recall our commuter whose device, upon determining that it was raining, performed a sequence of operations to discover an alternative method of transportation home. The implementation of this `checkBus` method (Fig. 13) uses the commuter's work and home locations to find reasonably sized bounding boxes around them. We use a 100 square meter box; since it is raining, our commuter does not want to walk too far. Using the LST-Structure that contains the bus's trajectory information, `checkBus` determines the bus's "coverage" of the work and home locations. If the coverage is sufficient, the method computes the expected travel time using the bus's trajectory information, alerts the commuter of his expected arrival time, and returns *true*, indicating that an alternative method of travel has been discovered.

```
public boolean checkBus (LSTStruct busStruct) {
  Location work = User.getLocation();
  Location home = User.getHome();
  //create 100m x 100m rectangle around workplace
  Window workArea = getBoundingBox(work, 100, 100);
  //create 100m x 100m rectangle around home
  Window homeArea = getBoundingBox(home, 100, 100);
  //compute coverage window with default settings
  if (busStruct.coverageWindow(workArea) >
          Constants.SUFFICIENT_COVERAGE &&
      busStruct.coverageWindow(homeArea) >
          Constants.SUFFICENT_COVERAGE ) {
    Time t2 = getCurrentTime();
    Time t1 = t2 - (60*60);
    //get the path bus followed for the past hour
    Path path = busStruct.findPath(t1, t2);
    //linear search on temporal aspect of path points
    LSTItem closestW = findClosest(path, work);
    LSTItem closestH = findClosest(path, home);
    Time estRouteT = closestH.time - closestW.time;
    alert("Take the bus!");
    alert("Est. arrival at: " + (t2 + estRouteT));
    return true;
  }
  return false;
}
```

Fig. 13: Sketch of `checkBus` implementation

Heading Coverage Window. Another of our scenarios entailed the commuter finding a spatiotemporally relevant promotional offer. Such an application might be implemented in a event handler invoked when an opportunistic connection to a nearby device appears (Fig. 14). If the commuter is headed in the direction the encountered device is coming from, then the commuter's device may query the encountered device for promotional offers. The calls "`d.LSTStruct.coverageWindow(win)`" and "`d.getObservations(win, User.Interests)`" call methods on the remote object `d`. The user who owns the device controls whether and how his information is released; the LST-Structure API can release statistical information about location without releasing the potentially sensitive raw location data.

[6]https://github.com/nathanielwendt/LST-Structure

```
//event handler triggered by device discovery
public void onConnected(Device d){
  headingQuery(d);
}
public LSTItem[] headingQuery(Device d) {
  LSTItem[] items = null;
  //get the user's current heading
  Heading heading = System.GetHeading();
  //compute rectangle encompassing current position
  //and likely positions within the next 10 minutes
  Window win = calculateWindow(heading, (10*60));
  double coverage = d.LSTStruct.coverageWindow(win);
  //determine whether device d is coming from where
  //the user is going
  if(coverage > Constants.DISCOVERY_COVERAGE){
    //get d's semantic data associated with window
    items = d.getObservations(win, User.Interests);
  }
  return items;
}
public Path getDirections(Device d, LSTItem item) {
    Location l1 = item.location;
    Location l2 = User.getLocation();
    Path path = d.LSTStruct.findPath(p1, p2);
    return path;
}
```

Fig. 14: Sketch of `headingQuery` implementation

Once the commuter has sifted through the observations, his device may request a specific path to reach, for example, the coffee shop offering the discount; as shown in `getDirections` in Fig. 14. As described previously, using the *in situ* routing information available on collaborating nearby devices may be preferable to using a static map server because the *in situ* information may better reflect current conditions.

This application requires devices to share their spatiotemporal data structures with one another. In our strawman approach, we release this data through an object abstraction (e.g., the `Device d` in Fig. 14), which allows the owner of the spatiotemporal data to control how and to whom his data is released. Our future work (described in more detail in the next section) will investigate protocols that explicitly allow users to *tune* the release of their spatiotemporal data for metrics of privacy, directly controlling the amount of personal information that is released when sharing spatiotemporal data.

VI. Discussion and Future Work

While our evaluation in Section IV provides detailed guidance on settings to use to achieve various quality and performance goals, tinkering with these parameters is non-trivial. An application interface could easily provide higher-level abstractions of these parameters that simply allow applications to provide quality and performance constraints, adjusting the settings for each type of observation, each context, or each user as appropriate. Additionally, it might be useful to facilitate controls for setting desired error levels in data accuracy and using some predictive modeling to determine the necessary parameters to match those levels. Such a façade is left for future work.

We would also like to investigate more deeply integrating the temporal dimension into the internal structure. One possibility is to use an R-Tree with time as a third dimension. This will support queries over regions at specific time intervals and allow for more efficient detection and removal of stale data. By expanding the functionality of the internal structure, we may be able to provide even more interesting application operations in the LST-Filters, providing a richer overall LST-Structure. It would be interesting to then compare the computational efficiency of different internal data structures.

More generally, we will need to replace our strawman open sharing model with a more intelligent privacy-centric model. This will include investigating protocols for data exchange between devices including methods for obtaining permission to access private data. We may include additional parameters allowing users to tune the privacy of their data and the openness of discovery. Users could then make tradeoffs for sharing data of their own for more interesting aggregates with other devices. We will connect with recent work on privacy-preserving aggregates [24] to allow users and applications to use trust relationships to navigate the tradeoffs between privacy and data release. Lastly, we will need efficient ways to summarize and exchange portions of a device's LST structure to facilitate efficient exchange; all of these aspects are avenues for future research.

VII. Summary

Motivated by real-world applications, we introduced the LST-Filter, a set of tunable application layers that operate over an enhanced model of spatiotemporal data. We defined a set of requirements for an internal data structure to interface with LST-Filters to create an LST-Structure. We augmented a k-d tree with a time-ordered linked list to create a k-d+ tree, an exemplar of this internal structure. We evaluated, over a wide variety of configuration parameters, the structure's ability to answer queries about coverage over regions, intelligently limiting the amount of data in the structure, and providing path information. We demonstrated the LST-Filter's ability to trade execution expense for quality of information on a *per data item* or *per operation* basis. The LST-Structure enables on-loading location storage, giving users direct control over potentially private information and enabling low-latency responses to spatiotemporal queries for a wide variety of applications.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider B., and Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of SIGMOD*, pages 322–331, 1990.

[2] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.

[3] J. Biagioni, T. Gerlich, T. Merrifield, and J. Eriksson. EasyTracker: Automatic transit tracking, mapping, and arrival time prediction using smartphones. In *Proc. of SenSys*, 2011.

[4] U. Blanke, T. Franke, G. Tröster, and P. Lukowicz. Capturing crowd dynamics at large scale events using participatory gps-localization. In *Proc. of ISSNIP*, 2014. (to appear).

[5] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *VLDB J.*, 15(3):211–228, 2006.

[6] P. Cudre-Mauroux, E. Wu, and S. Madden. TrajStore: An adaptive storage system for very large trajectory data sets. In *Proc. of ICDE*, pages 109–120, 2010.

[7] M. Erwig, R.H. Gu, M. Schneider, and M. Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

[8] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[9] S. Guha, M. Jain, and V. Padmanabhan. Koi: A location-privacy platform for smartphone apps. In *Proc. of NSDI*, 2012.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of SIGMOD*, 1984.

[11] S. Han and M. Philipose. The case for onloading continuous high-datarate perception to the phone. In *Proc. of HotOS*, 2013.

[12] Q. Jones, S. Grandhi, S. Karam, S. Whittaker, C. Zhou, and L. Terveen. Geographic place and communication information preferences. *CSCW*, 17(2-3):137–167, April 2008.

[13] M.B. Kjaergaard, S. Bhattacharya, H. Blunck, and P. Nurmi. Energy-efficient trajectory tracking for mobile devices. In *Proc. of MobiSys*, pages 307–320, June 2011.

[14] R. Lange, F. Dürr, and K. Rothermel. Online trajectory data reduction using connection preserving dead reckoning. In *Mobiquitous*, 2008.

[15] M.F. Mokbel, T.M. Ghanem, and W.G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.

[16] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser. Epfl mobility dataset. http://www.crawdad.org/epfl/mobility/.

[17] I. Rhee, M Shin, S. Hong, K. Lee, S. Kim, and S. Chong. NCSU - KAIST mobility models. http://www.crawdad.org/ncsu/mobilitymodels/.

[18] N. Sadeh, J. Hong, L. Cranor, I. Fette, P. Kelley, M. Prabaker, and J. Rao. Understanding and capturing people's privacy policies in a mobile social networking application. *Personal and Ubiquitous Computing J.*, 13(9):401–412, 2009.

[19] M. Satyanarayanan. Mobile computing: The next decade. *ACM SIGMOBILE Mobile Computing and Comm. Rev.*, 15(2):2–10, 2011.

[20] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present, and the future in spatio-temporal databases. In *Proc. of ICDE*, 2004.

[21] A.U. Tansel. Temporal databases. In *Wiley Encyclopedia of Computer Science and Engineering*, pages 1–7. 2008.

[22] E. Toch, J. Cranshaw, P. Hankes-Drielsma, J. Springfield, P. Kelley, L. Cranor, J. Hong, and N. Sadeh. Locaccino: A privacy-centric location sharing application. In *Proc. of Ubicomp*, pages 381–382, 2010.

[23] N. Vallina-Rodriguez, V. Erramilli, Y. Grunenberger, L. Gyarmati, N. Laoutaris, R. Stanojevic, and K. Papagiannaki. When David helps Goliath: The case for 3G onloading. In *Proc. of HotNets*, 2012.

[24] M. Xing and C. Julien. Trust-based, privacy-preserving context aggregation and sharing in mobile ubiquitous computing. In *Proc. of Mobiquitous*, December 2013.

[25] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *Proc. of MobiSys*, pages 113–126, 2012.