ToSS-it: A Cloud-based Throwaway Spatial Index Structure for Dynamic Location Data

Afsin Akdogan, Cyrus Shahabi, Ugur Demiryurek
Integrated Media Systems Center
University of Southern California
Los Angeles, USA
[*aakdogan, shahabi, demiryur*]*@usc.edu*

***Abstract*— The widespread use of GPS-enabled devices have led to a number of emerging applications that require monitoring and querying a large number of moving objects, such as in location-based services, mobile phone social networking, UAV surveillance, and car navigation systems. In such applications, indexes for moving objects must support queries efficiently and also cope with frequent updates. In this paper, we propose a *cloud-based* throwaway index structure, dubbed ToSS-it, where we generate the index from scratch in a short period of time rather than updating it with every location change of the moving objects.**

ToSS-it employs *inter-node* and *intra-node* multi-core parallelism paradigm to rapidly construct a distributed Voronoi Diagram. ToSS-it *scales out* by using a voronoi partitioning technique that minimizes the network message exchanges between the nodes (i.e., the major overhead in parallel generation of Voronoi Diagrams), and *scales up* since it fully exploits the multi-core CPUs available on each server. As a comparison point, with the state-of-the-art cloud-based spatial index structure (RT-CAN), if at least 7% of the objects are moving and issue updates to the index, it is faster to recreate ToSS-it from scratch than updating RT-CAN.

Keywords—moving objects, parallelism, cloud computing

I. INTRODUCTION

The capabilities of collecting people's and cars' location data have been growing rapidly with the fast progress of mobile, satellite, sensor, and video technologies. These spatial moving object datasets have enabled numerous new applications. Social networking services such as Facebook and Foursquare discover other people nearby a user; retail stores and malls combine real-time customer location data with geofencing for targeted marketing; mobile taxi applications such as Uber enable anyone to request a ride and match the clients with nearby drivers immediately. Obviously, this ever-changing massive data and large query volume resulted from wide-spread use of these applications bring a scalability problem.

The Holy Grail is to design a *scalable* method that can handle *frequent updates* and efficiently *search* this massive ever-changing spatial data. However, the challenge is that there is a tradeoff between search and update in index structures; namely, an index can be made more search efficient by trading off update efficiency and vice versa.

Towards this end, one group of studies, focused on devising specialized index structure for moving objects [4, 13, 17] but due to their centralized design they are not easily scalable. Alternatively, another group of studies focused on devising distributed versions of successful spatial index structures [1, 7, 25], leveraging the power of cluster of servers to scale out. However, their focus has not been on moving objects that result in frequent index updates, which is even less efficient with a distributed index structure than its centralized version due to the high network overhead which dominates the total cost.

Finally, a few studies proposed hybrid approaches and devise elegant distributed moving-object index structures [8, 21]. Both these studies observed the update-heavy workload of the application and tried to alleviate the problem by delaying updates and keeping track of them in a separate in-memory data structure: a one-dimensional z-ordered buffer in [8] and a two dimensional grid in [21], consistent with their underlying spatial index structure. Both studies suggested to occasionally swapping the update-tracking data structure with the permanent index, due to the fact that most objects' locations would be updated such that the tracking buffer would end up with the new locations of almost all objects. Even though both studies discussed a distributed version of their index structures by hash-partitioning the objects on multiple nodes (based on object ids), the update-tracking buffer needs to be maintained in a centralized manner, becoming the system bottleneck limiting scalability. This is because the update-tracking buffer needs to maintain a globally sorted list of updates (in z-order with [8] and in 2d-grid with [21]), which is inefficient to maintain in a decentralized manner.

Towards this end, we propose a new *scalable distributed (cloud-based) throwaway index*, where we no longer need to keep track of updates, eliminating the *centralized* update-tracking buffer. The reason that we can afford this elimination is due to the recent technological advancements in cloud servers and the multi-core CPUs available at each server --by taking advantage of this *two-level parallelism* (*inter-node* and *intra-node*), we can construct the index so fast that it is more efficient to frequently generate the index from scratch than to update it (even if only a small portion of the objects are moving). This is analogous to the concept used in computer graphics where the content of the display memory is created entirely from scratch (e.g., 60 times a second) rather than

maintaining data structures that track the updates from one memory frame to the other. Note that, there may be a negligible staleness in our query results due to the fact that queries use the current index until a new one is constructed. However, under high update load any other approach will observe the same staleness since it takes the same (or more) time for the updates to take effect. Moreover, given that real-world mobile applications introduce an artificial delay between two consecutive GPS readings to save battery, there will be staleness in the query results anyway [26].

The fundamental construct of our index structure, dubbed ToSS-it (for Throwaway Spatial Index Structure), is a Voronoi diagram, which is an extremely efficient data structure to answer spatial queries. Hence, our goal is to construct a Voronoi diagram on all the point data (i.e., moving objects) rapidly and in a distributed fashion. The challenge is that this *global* Voronoi diagram should be broken and stored on multiple nodes of the cloud. The intuitive approach to accomplish this is to first build the global Voronoi over all the points (in parallel) and then partition it across the cloud servers. This build-first-distribute-later approach has been recently proposed by us [1] and others [3]. However, in this paper, we take a completely different approach of *distribute-first-build-later*.

That is, we first distribute all the points across cloud servers and then construct *local* Voronoi diagrams at each server. Obviously, these local Voronoi diagrams will include some inaccurate Voronoi cells because some of their neighboring points may be residing in a different server. Therefore, we need to exchange some information between servers to ensure that the cells are accurate and if not, fix them. The more the number of these inaccurate cells, the more information must be exchanged over the network which results in inefficiency. Thus, we propose a novel three-step approach to address this challenge. First, ToSS-it minimizes the number of network exchanges during the index creation by adapting an intelligent partitioning technique that quickly learns from the dataset and distributes the objects across the servers while preserving the spatial proximity among the objects. Second, we devise a technique to quickly identify whether a generated cell at a server is accurate or not, locally, without requiring any information about other servers, and refine inaccurate cells effectively. Such a mechanism provides *high scalability* because every server fixes the inaccuracies at the borders of the partitions in parallel with multi-way message passing (all servers communicate with each other in parallel) without requiring another server to merge the partial results which will become the system bottleneck. Third, we further reduce the number of network exchanges by replicating the border points so that the points with inaccurate cells can be fixed using only the local information without requiring any message exchanges.

Finally, to expedite spatial query processing at each node even further, we construct a hierarchical index structure on top of the local Voronoi cells. This is achieved by decoupling local Voronoi cell generation at each node and hence exploiting the multi-core CPUs available on each server. Consequently, ToSS-it not only can *scale out* because of our novel 3-step distribute-first-build-later technique, but also can scale up due to our *intra-node parallelization* during the local index construction. Although most of the current studies in Cloud Computing have focused on scaling out, it is crucial to utilize multi-core CPUs as chip vendors such as Intel have recently shifted to multi-core architecture rather than increasing processor speeds [12].

We performed several experiments by fully implementing ToSS-it on the Amazon cloud. ToSS-it yields a very high query throughput, scales *near-linearly* with the number of nodes in the cluster and the total execution time remains almost *constant* as the data size increases. For example, we can process about 30,000 circular range queries per second with a small cluster. Moreover, if at least 7% of the objects are moving and issue updates to the index, it is faster to recreate ToSS-it from scratch than updating state-of-the-art cloud-based index structure, RT-CAN [25]. In addition, with ToSS-it, a query can be forwarded to any node, for example in a round-robin fashion, and this *decentralization* provides *load-balancing* across the nodes.

II. RELATED WORK

Spatial indexing has been one of the active areas in recent database research. We separate spatial indexing techniques into three groups as centralized, cloud-based, and hybrid and discuss each group separately in this section.

A. Centralized Moving Object Index Structures

Most of the existing spatial indexes are designed for a centralized paradigm where all spatial operations are performed on a single server. Among these approaches R-tree [11] and Quadtree [9] are the most prominent index structures as they are also employed in commercial products such as Oracle. However, both of these methods suffer from the search/update tradeoff. In particular, R-tree, being an object-based index, is efficient for search but slow in updates while Quadtree, a space-based index, is popular because of its effectiveness in handling updates but performs poorly in search specifically when the dataset has a skewed distribution.

Clearly, handling highly dynamic datasets is not the strong suit of these conventional index structures; therefore, other techniques have been proposed [4, 13, 17] to efficiently handle moving object datasets. The main problems with these approaches are as follows. First, they make the simplifying assumption that extra information about the object movement behavior is known in advance, such as direction and speed, which might not be available in real-world applications. Second, given the limited computational capability and storage of a single machine, a centralized system will eventually suffer from performance deterioration as the size of the dataset or the number of queries increases.

B. Cloud-based Spatial Index Structures

To address the scalability issues, there are few recent studies that handle spatial data in the cloud computing context. RT-CAN [25] integrates a CAN-based [19] routing protocol with an R-tree based indexing scheme to support multi-

dimensional query processing in a cloud system. QT-Chord [7] is very similar to its preceding approach RT-CAN with the main difference that it combines Chord [23] routing protocol with Quadtree. In response to a data update, CAN finds the target node to send information with $O(\sqrt{N})$ number of network messages, where N is the number of nodes in the cluster. Considering the frequent updates from the (moving) objects and high network overhead, it is impossible to keep up with such index updates. Note that although Quadtree outperforms R-tree in terms of updates in a centralized system, in a distributed environment Quadtree is no longer more efficient than R-tree as they have almost the same network cost, which dominates the total cost; therefore, both these approaches suffer from update. A detailed cost analysis about RT-CAN can be found in Section 6.A.

Several other approaches that use distributed hierarchical index structures such as R-tree based SD-Rtree [14] and kd-tree based k-RP [15] have been proposed for parallel spatial query processing. The major problem with the tree-based approaches is that they do not scale due to the traditional top-down search that overloads the nodes near the tree root, and fail to provide full decentralization.

C. Cloud-based Moving Object Index Structures

The idea of short-lived throwaway index structures for indexing moving objects is not new [8, 21]. However, as discussed in Section 1, even though we adopt the same terminology of calling our index structure throwaway, the main difference is that Movies [8] and TwinGrid [21] maintain an update-tracking buffer for occasionally swapping this buffer with the underlying spatial index. Hence, it is not really the case that they ignore updates but just delaying them. Unfortunately, the centralized update-tracking buffer becomes the main hindrance for a truly decentralized approach since it limits the scalability considerably. Moreover, to keep that buffer sorted as new updates arrive, its data structure must be very simple: one-dimensional z-order in [8] and 2d-grid in [21]. Consequently, to be swappable, the same exact simple data structure must be used for the underlying spatial index, which results in approximate query answering in [8] and inefficiency for real-world skewed datasets in [21]. By eliminating the update-tracking buffer with ToSS-it, we not only can scale out (and up) but also our underlying data structure becomes independent and can become spatial-query and spatial-data friendly, i.e., Voronoi.

We consider a scenario where ToSS-it server polls all the registered mobile users through a mobile application at a fixed frequency and subsequently build a fresh index on all the reported locations from scratch. Therefore, ToSS-it has no notion of "updates" and treats the new locations of all the moving objects registered with the systems as "new". Hence, the need for an update-tracking data structure is eliminated.

III. PRELIMINARIES

Voronoi Diagram:

A Voronoi diagram decomposes a space into disjoint polygons (cells) based on the set of generators (i.e., data points). Given a set of generators S in the Euclidean space,

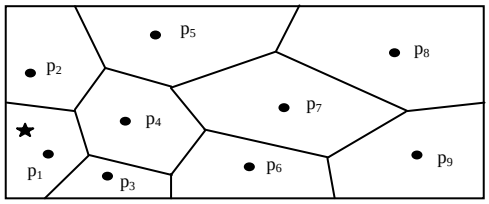

Fig. 1. A sample Voronoi Diagram

Voronoi diagram associates all locations in the plane to their closest generator. Each generator s has a Voronoi cell consisting of all points closer to s than to any other site.

DEFINITION 1. Voronoi Cell

Given a set of generators $P = \{p_1, p_2, \ldots, p_n\}$ where $2 < n < \infty$ and $p_i \neq p_j$ for $i \neq j$, $i, j = 1;\ldots, n$, the Voronoi Cell of p_i is $VC(p_i) = \{p \mid d(p, p_i) \leq d(p, p_j)\}$ for $i \neq j$ and $p \in VC(p_i)$ *where $d(p, p_i)$ specifies the minimum distance between p and p_i in Euclidean space.*

Figure 1 illustrates an example of a Voronoi diagram and its cells for nine generators. Voronoi diagrams have several geometric properties. Here we enlist their main properties that we use to establish our proposed solution. The proofs for these properties can be found in [18].

Property 1: The Voronoi diagram for a given a set of generators is unique.

Property 2: Let n and n_e be the number of generators and Voronoi edges, respectively, then $n_e \leq 3n\text{-}6$.The average number of Voronoi edges per Voronoi polygon is at most 6, i.e., $2(3n\text{-}6)/n = 6\text{-}12/n \leq 6$. This states that on average, each generator has 6 adjacent generators. This property is especially useful to limit the number of candidate generators during Nearest Neighbor search.

Property 3: The nearest generator point of p_i (e.g., p_j) is among the generator points whose Voronoi cells share similar Voronoi edges with $VC(p_i)$.

IV. TOSS-IT: A THROWAWAY SPATIAL INDEX STRUCTURE

In this section, we first explain our parallel Voronoi generation algorithm that uses multiple servers in the cloud. The main idea behind our approach is first distribute all point objects across multiple cloud servers, and then build local Voronoi diagrams at each server (i.e., *distribute-first-build-later*). We discuss our 3-step approach in order to minimize the number messages passed between the cloud severs during the Voronoi construction, i.e., 1) intelligently distribute the objects across the servers, 2) quickly identify the inaccurate Voronoi cells and 3) replicate the border cells. Next, we present our hierarchical Voronoi index structure that is built in each cloud server to expedite the query processing at each node. This index consists of multiple layers representing local Voronoi cells at different levels of granularity. Finally, we discuss how we use our proposed index structure ToSS-it to answer a variety of spatial queries.

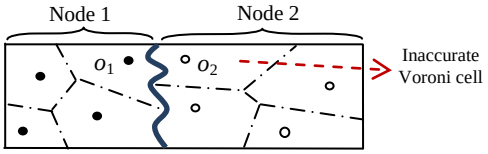

Fig. 2. Distributed Voronoi diagram generation for 8 objects with 2 nodes.

A. Parallel Voronoi Diagram Generation

Given a set of scattered point objects *O* in the plane and *N* servers (nodes) in cloud, our goal is to generate Voronoi diagram of *O* using *N* servers in parallel.

We have proposed a distributed Voronoi Diagram generation algorithm in [1]. In this work, we used a *build-first-distribute-later* approach and modeled distributed VD generation problem with MapReduce [5]. Specifically, we generate partial Voronoi diagrams (PVD) in each node in parallel and merge the (partial) results in a single node to build one global Voronoi Diagram. Subsequently, we partition and redistribute the global Voronoi Diagram to each node for parallel query processing. The main shortcoming of this build-first-distribute-later approach is its *limited scalability* as the single node that constructs the global Voronoi Diagram becomes the system bottleneck.

In this paper, we propose a completely different approach that addresses the scalability shortcoming of our previous method. The main idea of ToSS-it is to *distribute-first-build-later*. That is, we first effectively distribute the objects across the nodes to balance the load at each node. Next, we build local Voronoi diagrams at each node. The challenge is that even though the objects are distributed effectively, when *O* is divided into *disjoint* subsets, some of the generated local Voronoi diagrams will be inaccurate as their neighbors reside in different nodes. The nodes need to communicate between each other to validate the accuracy of Voronoi cells and fix them if needed, which yields significant network I/O.

For instance, Figure 2 illustrates an example of two computing nodes that are asked to generate Voronoi Diagram of eight data objects. The objects are divided into two equally sized partitions where each node stores four objects. As shown, Voronoi cells of objects o_1 and o_2 are inaccurate. This is because even though o_1 and o_2 are Voronoi neighbors, they are sent to different nodes in which local Voronoi diagrams are created independently. To generate the correct Voronoi diagram the nodes need to communicate between each other (i.e., two-ways message passing). Obviously, as the number of inaccurate cells grows, the communication overhead increases significantly resulting in performance problems.

ToSS-it addresses the above challenge by (i) minimizing the number of network messages passed between the nodes by using a Voronoi-based partitioning technique, (ii) identifying whether a cell is accurate or not, and fixes only those cells through network messages, and (iii) including a replication technique which replicates the border objects in each partition to further reduce the number of network messages.

We explain these three phases in turn.

1) Data Partitioning: Given *N* number of nodes and a set of data objects *O* which are randomly distributed across the nodes, the purpose of this phase is to assign each object *o*, where $\forall o \in O$, to one of the *N* nodes.

The ideal partitioning should preserve spatial locality among the objects in each partition to minimize the number of inaccurate Voronoi cells (and hence message passing between the nodes). Towards this end, we propose a Voronoi-based partitioning technique. The main idea is that we select *P* number of pivot data objects, where each pivot corresponds to a partition. Subsequently, we assign each object *o*, to its closest pivot. We use a two-step approach to select the pivots in a distributed manner. In the first step, each node identifies a set of randomly selected candidate objects and sends them to the master node. Upon receiving all candidates, the master node identifies *P* number of pivots and transmits *P* back to all nodes. We note that each pivot corresponds to a partition and each partition (that includes subset of *O)* is stored in a separate node. ToSS-it can store more than one partition in a single node since the neighboring partitions in the same node can still preserve the spatial locality. Finally, after the set of pivots (each representing a node) are received, the nodes start to shuffle the objects by assigning each object *o* to the closest pivot (and hence its corresponding node). We formally define object & node mapping function *f* as follows:

$$f(o.location) = P_i, \text{ where } d(o.loc, P_i) = \min\{d(o.loc, P_i)\}, \ \forall \ P_i \in P$$

where *o.location* is composed of a latitude and longitude value, and node N_i stores partition P_i.

Note that, we assume the data objects are initially randomly distributed among the servers. Our purpose here is to test the worst case performance of ToSS-it construction time by increasing the number of points needs to be transferred over the network in the shuffling phase.

Figure 3 illustrates an example of our data partitioning approach. As shown in Figure 3a and Figure 3b, the data objects are initially randomly distributed across five nodes. First, the master node gathers a set of candidate pivots from each node and computes five pivot objects, and broadcasts them to the nodes (see Figure 3c). At this point, each node is assigned a specific pivot and also stores the information about all the other pivots. For example, node N_5 is mapped to pivot P_5 and stores the information of the other pivots $\{P_1, P_2, P_3, P_4, P_5\}$ (see Figure 3d). Finally, the nodes redistribute the objects based on the pivot set, where each object is assigned to the node with its closest pivot. Once the redistribution phase is completed, as shown in Figure 3e spatially close objects are clustered in the same node.

Clearly, pivot selection is an important task to obtain balanced distribution among the nodes. Next, we discuss our distributed pivot selection algorithm. As we discussed, each node N_i, where $\forall N_i \in N$, initially stores O_i number of objects i.e., random subset of O, where $1 \le i \le N$ and $O = \bigcup_{1 \le i \le N} O_i$. With our approach, each node randomly selects candidate S_i objects among O_i and sends S_i to the master node. The master node generates S/P (where P is the number of pivots) number of random sets where each set has P objects. Then for each set,

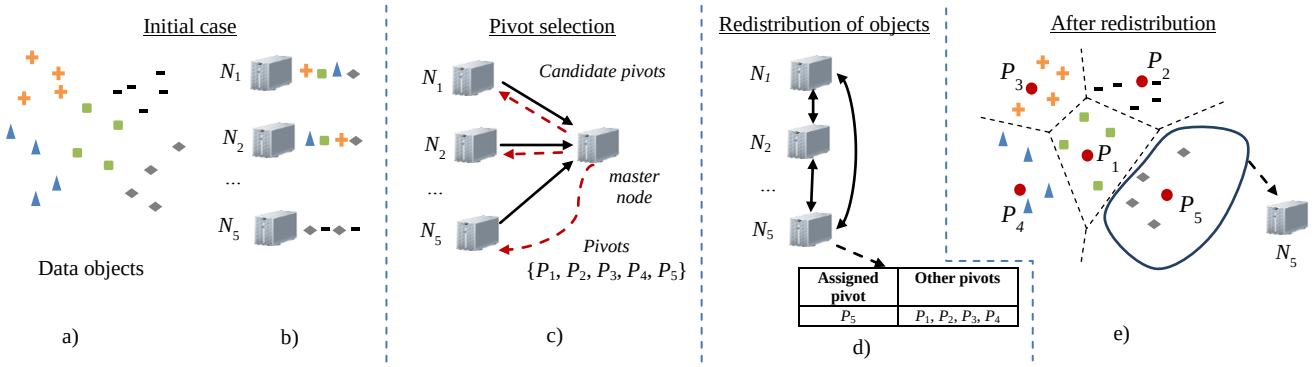

Fig. 3. Workflow of the data partitioning phase:
a) A set data objects O in the plane. b) Objects randomly distributed across 5 nodes. c) Pivot selection. d) Shuffling objects among the nodes based on elected pivots. Each node maintains the set of pivots. e) After redistribution objects nearby in space are stored together in the same node. Each pivot is mapped to a node.

we compute the total sum of the distances between every two objects and choose the set with the maximum total sum as pivots. Finally, the master node broadcasts the pivots to all the nodes.

2) Local Voronoi Diagram Generation: In this phase, each node N_i is asked to generate a local Voronoi diagram (LVD) for the corresponding objects o, where $\forall o \in O_i$. We have two specific goals in this phase: we aim to a) exploit all multi-core CPUs available at N_i to generate local VDs, and b) effectively identify and fix the Voronoi cells which might be inaccurate because of the partitioning.

A straightforward method to generate local Voronoi Diagram is to use sweepline algorithm [18]. However, out-of-the-box sweepline algorithm is sequential and cannot be executed in parallel. Towards this end, we modify sweepline algorithm to utilize multi-core CPUs. In particular, let C_i be the number of cores available at node N_i. We break the problem into smaller pieces where each core c, where $\forall c \in C_i$, handles O_i/C_i number of data objects. To distribute the objects across the cores we use the same partitioning technique used in the previous phase. First, c number of pivots is identified using random selection method. Then, each core c is assigned O_i/C_i number of data objects. Subsequently, the sweepline algorithm is executed by each core c and O_i number of Voronoi cells is generated in parallel as the final output at node N_i. As we discussed earlier, since the objects are divided into disjoint subsets, some of the generated Voronoi cells (VC) may be incorrect because their Voronoi neighbors reside in different nodes. We define these generated cells as Inaccurate Voronoi Cell (*IVC*). The formal definition of *IVC* is as follows:

DEFINITION 2. Inaccurate Voronoi Cell (IVC)

Let P be a set of data points, P^+ and P^- be subsets of P and $P^+ \cup P^- = P$. If we only consider the points in P^+, the generated Voronoi cells might not be accurate because there might be some points $p_j \in P^-$, which will decrease the area of the generated cells. Therefore, these cells might be inaccurate (IVC).

Now the challenge is how to identify whether a generated *IVC* is correct or not. To this extent, we need to identify a set of candidate objects that can possibly modify an *IVC*. If all these candidate objects reside in the same node, we know that the cell is accurate.

To identify inaccurate cells, we first compute, for each object o, $\forall o \in O_i$, an influence region *IR*(o) and then check *IR*(o) against the borders of the node N_i. If *IR*(o) falls inside the N_i, o does not need to be refined. Otherwise, o is marked as inaccurate. Note that, the inaccurate cells are the border objects of the partitions, and hence are a very small subset of O_i. Another important point is that we can identify such inaccuracies locally without requiring any information about other nodes and this mechanism provides **high scalability** because every server first detects and then fixes the inaccuracies in parallel (with multi-way message passing). We define Influence Region IR as follows.

Influence Region (IR) is computed based on current IVC. In particular, for each vertex v on the IVC of an object o, the locus of the points which can exclude v from the cell is inside a circle centered at v with a radius r = d(v, o). Figure 4 illustrates the influence region of object o. As shown, o′ is inside the circle IR_1. The bisector crossing the line |o, o′| intersects with the IVC causing the vertex v_1 being excluded from the cell. The set of circles IR_i specifies the minimum influence region IR including all points that their presence in P^+ changes the IVC. The radius of circle centered at IR is $r_o = \max(d(o, v_i))$.

3) Geospatial Replication: As we discussed local Voronoi diagrams in each node include inaccurate cells and we can identify them using *IR*. Once we identify the inaccurate Voronoi cells, we replicate their generator (data objects) across the partitions so that the potential inaccurate Voronoi cells are refined using local information. This enables us to

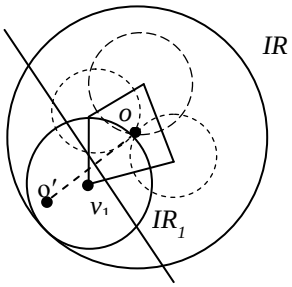

Fig. 4. Influence region of object o.

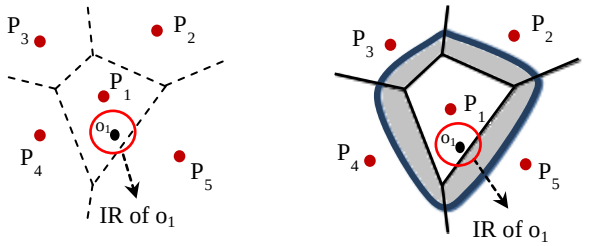

Fig. 5. a) An inaccurate Voronoi cell (o_1) whose IR is crossing another partition (P_5); therefore, o_1 needs to be refined. b) A sample geospatial replication. Object o_1 does no longer need refinement.

avoid significant amount of network communication. In particular, to refine *IVC* of *o*, we first identify the candidate partitions with which *IR*(*o*) overlaps. Specifically, let *CN*(*o*) be the set of nodes need to be considered to refine *o* and N_i be the node that stores *o*. N_i sends *IR*(*o*) to each of the nodes in *CN*(*o*) and asks for the objects within *IR*(*o*). Consequently, the *IVC* of *o* is refined based on the information passed to the node. Figure 5a illustrates an inaccurate Voronoi cell o_1 whose influence region is crossing another partition, i.e., P_5. To refine o_1, N_1 will communicate with N_5 and ask for the objects inside *IR*(o_1).

Figure 5b illustrates our geospatial replication technique where the shaded area around partition P_1 is replicated into node N_1. In this specific case, o_1 does not need to be refined anymore. We note that replicating small amount of data improves the accuracy significantly. Our *experiments* show that by replicating 5% of the data objects, we can reduce the number inaccurate cells by 70%. The question is that how much of the border objects should be replicated to accomplish maximum accuracy. Towards that end we use range selection technique where all the objects within a certain distance λ (see Figure 6 from the border of a partition are replicated. Note that, we replicate data objects at the partitioning phase by slightly modifying the *object-&-partition* mapping function *f* presented in Section 4.A.1. The formal definition of the modified mapping function is as follows:

$$f(o.location, \lambda) = P_i, \{P_{ne}\}$$

where $d(o.location, P_i) = min\{d(o.location, P_i)\}$, $\forall P_i \in P$. Let $VN(P_i)$ be the Voronoi neighbors of partition P_i, which originally contains *o*. Then $P_{ne} = \{P_j\}$, where $\forall P_j \in VN(P_i)$, $d(o, hp(P_i, P_j)) \leq \lambda$ and $hp(P_i, P_j)$ is the hyper plane (border line) between pivots P_i and P_j. The computation of distance is $d(o, hp(P_i, P_j))$ is presented in [16] using triangle inequality.

Figure 6 shows an example of the object-&-partition mapping with geospatial replication. Partition P_4 is the closest pivot to object o so it is directly mapped to P_4. Subsequently, since d(o, hp(P_4, P_1) is less than λ, it is mapped to P_1 as well. In this specific example, P_{ne} is composed of only P_4; however in some cases an object might be mapped to more than 2 partitions. These are the objects that are at the intersection of multiple partitions.

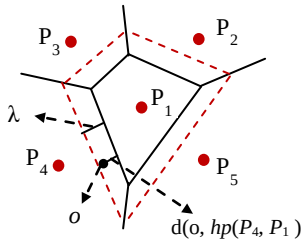

Fig. 6. All objects within λ threshold from the border are mapped to multiple partitions.

B. *Building Hierarchies on Local Voronoi Diagrams*

In this section, we briefly discuss how we further improve query processing by building local hierarchies on top of local Voronoi diagrams.

Several approaches have been proposed that construct hierarchies on top of Voronoi diagrams [20]. Rather than adapting such sophisticated techniques, we take a completely Voronoi-based approach where we organize the objects with hierarchies of Voronoi cells. A Voronoi hierarchy is defined as follows. A Voronoi cell of the higher level-i contains all Voronoi generators of the lower level-i-1 that are closer to that level-i generator than to any other [10]. The main advantage of this approach is that we can build Voronoi hierarchies by exploiting multi-core CPUs efficiently. Our bottom-up hierarchy generation algorithm is as follow. Let *f* be the fan-out of a bucket and *c* be the number of cores available at a particular node. First, *f* number of pivots are identified, using the same technique applied in the data partitioning phase. Then each core c_i is assigned *f*/*c* number of objects, and subsequently each c_i maps the objects to their closest pivot. Once all objects are mapped, the same algorithm is iteratively executed for the higher level. In the following section, we show how we use these hierarchies to speed up query processing.

V. QUERY PROCESSING

In this section, we show how we use ToSS-it to evaluate two major types of spatial queries: range and k nearest neighbor (k-NN)[1]. With ToSS-it, the queries can be submitted to any node in the system, e.g., in round-robin fashion. This decentralization provides **load balancing** across ToSS-it nodes. The receiving-node either processes the query or forwards it to another node with **one network message** using the pivots we generated in the data partitioning phase.

In particular, let N_{init} represent the server that receives query *q (that includes the query type and the location of q)*, and N_q represent the server that contains location of *q*. If *q* is contained in N_{init} (i.e., N_{init}=N_q), the query is processed in N_{init}. Otherwise, N_{init} finds N_q using a random-walk algorithm [10] on the pivot list, and then forwards *q* to N_q. Recall that each node maintains the list of pivots *P*, where each pivot corresponds to a node. The total cost of locating the node N_q with random-walk is $O(\sqrt{P})$, where *P* is the number of pivots, and we consume only one network message for forwarding *q*

[1] Several studies have shown the superiority of Voronoi-based index structures for various types of spatial queries (e.g., [25]).

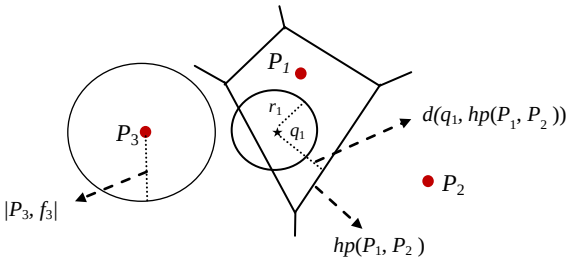

Fig. 7. Query q_1 discards both partitions P_2 and P_3.

from N_{init} to N_q. We note that in distributed environments, the network message exchange between the nodes is the dominating cost, and ToSS-it minimizes this cost by using only one message. Moreover, the space requirement of storing P pivots at each node is very small. We store 8 bytes for the pivot location (latitude, longitude), and 32 bytes for the IP address of the host of each pivot. For example, given 1,000 pivots, the total space requirement of the pivot list is only 40KB.

Once the node N_q is found, ToSS-it processes the query using the local Voronoi Diagram (LVD) of N_q. In the rest of this section, we explain how we use ToSS-it to answer Range and k-NN queries, respectively.

A. Range Query

Given a dataset O, query point q, and distance r, range query q finds the objects O' of O such that $\forall\, o \in O', |q, o| \leq r$.

Upon receiving query q, N_q finds the set of nodes (IN_q) with which q intersects and forwards q to the nodes in IN_q. Subsequently, each node N_i, where $\forall\, N_i \in IN_q$, executes the query in parallel. To find IN_q, we first include all neighbors of N_q into IN_q and then refine it by using r. In particular, let P_i be one of the neighboring partitions, if $d(q, hp(P_q, P_i)) > r$, then we remove P_i from IN_q. Otherwise, we keep P_i in IN_q and add P_i's neighbors to IN_q as potential intersecting partitions. This process terminates once all the nodes in IN_q are examined.

Figure 7 illustrates an example of range query based on LVDs. Given a query q_1 with range r_1, we first locate pivot P_1 with the closest distance to q_1 and then initialize IN_{q1} with P_1's neighbors i.e., P_2 and P_3. Since $d(q_1, hp(P_1, P_2)) > r_1$, we remove P_2 from IN_{q1}. This means that there is no use for the LVD in the node corresponding to P_2, and hence no message needs to be sent. On the other hand, $d(q_1, hp(P_1, P_3)) < r_1$, thus we add P_3 and P_3's neighbors to IN_{q1}. The algorithm runs until all the elements in IN_{q1} are examined.

We can further improve the partition search algorithm by pruning false-positive partitions using the location of the furthest object (from the pivot) in that partition. In particular, even if P_i satisfies the equation, $d(q, hp(P_q, P_i)) < r$, we can discard it using the following rule.

PRUNING RULE. Furthest Object

LEMMA-1: Let f_i be the location of the furthest object in the partition from the pivot P_i, where $f_i = max\{|o, P_i|\}$, $\forall\, o \in O_i$. Given a query q with a range r, even though $d(q, hp(P_q, P_i)) < r$, if $|P_q, P_i| > |P_i, f_i| + r$ then no object in P_i can be in q's range.

The proof is straightforward, and we explain it using Figure 7. As shown, q_1 intersects with partition P_3; however, by keeping the location of the furthest object f_3, we guarantee that q_1 does not cover any object of P_3.

Finally, once IN_q is computed, we forward q to each node in IN_q and execute the query using the local Voronoi diagrams. For the rest of the query processing, we iteratively traverse Voronoi hierarchies using the same pivot locating and pruning strategies described above. The cost of this top-down traversal is $\sqrt{f} \cdot h$, where h is the height of the hierarchy and f is the fan-out. Once local query processing is completed, each node sends its partial result to the query issuer.

B. k Nearest Neighbor Query

Given a query point q and a set of data objects O, k Nearest Neighbor (kNN) query finds the k closest data objects $o_i \in O$ to q where $d(q, o_i) \leq d(q, o)$.

The straightforward strategy to process a kNN query is to locate the Voronoi cell that contains q and then execute the sequential kNN algorithm as in centralized systems [20]. The problem with this approach is that when the value of k is large, the performance degrades, especially when q retrieves objects from multiple nodes. We propose an incremental algorithm, which is similar to the approach in [25], to answer kNN queries using multiple nodes in parallel. The main intuition is to start the search with an estimated range, and enlarge the search range until enough number of objects is found. Thus, we express a kNN query as a sequence of range queries. First, we locate the node N_q which contains q and set an initial radius r_{init} using the following equation [24] which gives distance estimation between q and its k nearest neighbor in 2 dimensional space:

$$r_{init} = ed(k) / k \tag{1}$$

$$ed(k) = \left(\frac{2\sqrt{1}}{\sqrt{\pi}}\left(1-\sqrt{1-\sqrt{\frac{k}{O}}}\right)\right) \tag{2}$$

where O is the total number of objects. Subsequently, N_q locates the nodes within the r_{init} range, and asks those nodes to execute a range query with r_{init}. Each node reports back the number of objects from their range query. If the number of objects is not enough (i.e., less than k), we increase r_{init} by $ed(k)/k$ and continue to increase until we find k objects. Once we have enough number of objects, the nodes send these candidate objects and N_q selects the closest k objects to q as results.

VI. EXPERIMENTS

We conducted our experiments on Amazon's EC2 cluster. We use varying number of computing nodes to evaluate inter-node parallelism (scale-out) of ToSS-it. The number of computing nodes in our test cluster varies from 32 to 128. We evaluated intra-node parallelism (scale-up) of our local index generation (LVD) using 8xLarge (32 CPU) nodes. These nodes are connected via 10Gbps network, and run on 64 bit Fedora 8 Linux Operating System with 60 GB memory, 32 virtual CPUs, and 4 disks with 840 GB storage. To evaluate

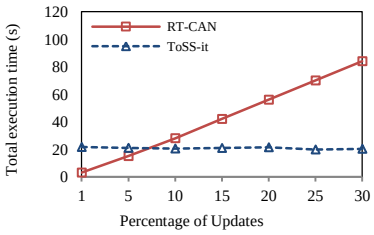

Fig. 8. ToSS-it construction versus RT-CAN update (uniform dataset, N=64,CPU=32)

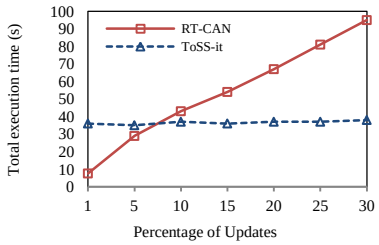

Fig. 9. ToSS-it construction versus RT-CAN update (taxidataset, N=64, CPU=32)

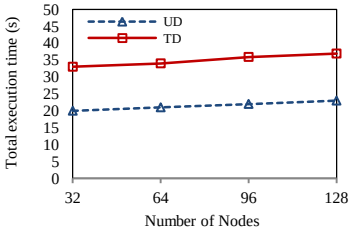

Fig. 10. Effect of node number

scale-up, we disable parallel LVD generation and first run the algorithm in single-thread mode. Subsequently, we enable multi-threading mode and exploit multiple CPUs to generate LVD.

As for our datasets, we used two different datasets that consists of a uniform dataset (UD) and real-world taxi dataset (TD). Specifically, each dataset consists of approximately 500,000*N* objects (as proposed in [7, 25]), where *N* is the number of nodes, and each object has an x and y value. For UD dataset, the objects are uniformly distributed in the range of $[0,10^9]$. Our real-world TD dataset includes Beijing taxi dataset which consists of trajectories of 13,000 taxis over 3 months. Each taxi reports its location approximately in every 10 seconds on average. Since the number of objects is only 13,000 (relatively small) we split each trajectory into 25 minute intervals and treat each of these 25 minute trajectories as a different object. As a result, we generate approximately 67 million objects. Initially, we randomly distribute the objects among the nodes.

We use throughput (processed requests per second) to evaluate query performance of our index structure. In real systems, there are concurrent requests, where multiple users run queires on the index stucture. To this extent, we send concurrent requests to ToSS-it using the technique discussed in [2], which benchmarks the performance by increasing the number of threads. Specifically, we first run the experiments using one thread (T=1), and then increase the value of T by one to run the experiment again. A thread generates a sequence of queries (range and *k*NN) and may not issue another until its pending request is served. We repeat this process as long as we observe a higher throughput than the previous experiment. The point where throughput cannot improve further is the maximum throughput that ToSS-it can achieve. We run the experiments for 2 minutes and then compute the total number of processed queries, scale the results to 1 second and report it as throughput.

As we discussed in Section 2, other throwaway index structures [8, 21] are intrinsically centralized because of the central buffer structure that globally tracks the moving objects. On the other hand, Toss-it is a fully decentralized distributed index structure. Therefore, we compare our approach with the state-of-the-art cloud-based decentralized spatial index structure RT-CAN [25]. In the rest of the experiments, we first compare one-time index creation performance of ToSS-it with update performance of RT-CAN. We argue that recreating ToSS-it is indeed faster than updating RT-CAN, even if only a small portion of the objects issue update to the index. Subsequently, we compare the query (i.e. *k*NN and range) throughput of ToSS-it with that of RT-CAN, and show that our approach scales better and outperforms RT-CAN for both query types.

A. ToSS-it Construction

In this experiment, we study the impact of varying number of nodes, CPUs, and datasets on the ToSS-it construction. Figure 8 compares the construction time of ToSS-it with the RT-CAN's update performance. In this experiment, we use 64 nodes and the uniform dataset. To update the location of the objects in the uniform dataset, we randomly generate two values x', y' within [∓100, ∓250] range and add them to the current position (x, y) in the coordinate system. The justification for [100, 250] range is as follows. According to Google maps, a human walks 80 meters/minute on average. The perimeter of the world is $40x10^6$ meters. When we convert the walking speed in real life into our coordinate system ($[0, 10^9]$), a human can walk 2,000 units per minute on average. Assuming a mobile device can report current location in several seconds, a number between 100 and 250 corresponds to how much a person can walk. We simulated human behavior in our experiments with the uniform dataset. As shown, even if more than 7% of the objects issue location updates it is faster to recreate ToSS-it from scratch. We observe the same pattern in Figure 9, where we run the same experiment with the taxi dataset. The only difference is that with the taxi dataset both RT-CAN and ToSS-it get slower and the lines cross around 8%.This is because the change in the location of a taxi between two consecutive GPS readings is larger compared to the distance a person can go. With RT-CAN larger movement leads more update propagation and increases the total update cost. With ToSS-it, larger movement does not have any impact; however, the distribution of the real data is skewed as taxis are more clustered in some regions and skewed datasets slightly reduce the effect of geospatial replication in ToSS-it.

To explain why a complete index construction is faster than updating 7% of objects, let us analyze each cost in details. We first analyze the update cost of RT-CAN. The performance of RT-CAN is severely affected by a 3-step index update approach. First, RT-CAN locates the node which contains the query object by redundant routing between the nodes. This routing cost is $log(N)$ messages, where N is the number of nodes in the cluster. Second, it updates the local R-tree with respect to new updates. Third, since the structure of

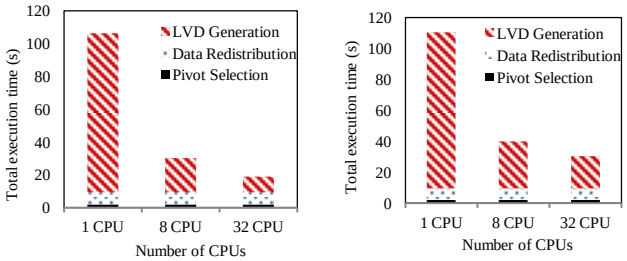

Fig. 11. Effect of CPU number on ToSS-it construction

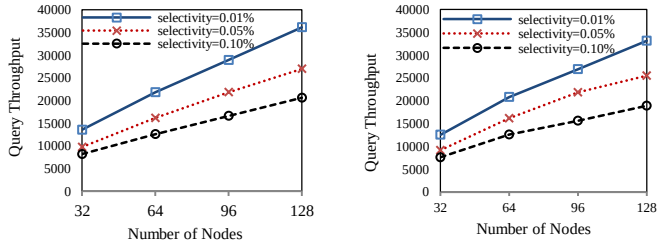

Fig. 12. Effect of selectivity on range query

the local R-tree in the node changes, the updates are propagated to other nodes with additional $3\log(N/4)$ messages. Let us explain the index update cost of RT-CAN using our UD dataset. In particular, 7% of UD dataset with 64 nodes corresponds to $2.2\text{x}10^6$ objects (i.e. 7% x 64 x 500,000). In the first step, RT-CAN passes 6 messages (i.e., $\log(64)$) to locate the node whose index needs an update. Those 6 messages are need to handle only one update, and hence the total number of messages for $2.2\text{x}10^6$ updates is $13.4\text{x}10^6$, which is already 42% of the entire dataset. Once the nodes are found, in the second phase, the local R-trees are updated, where even a single object might change the whole structure of an R-tree. In the third phase, since the local R-trees are different now, RT-CAN propagates the index changes to other nodes, which incurs additional $3\log(N/4)$ messages per each update propagation.

Now let us analyze the cost of creating ToSS-it from scratch. ToSS-it only incurs $\beta \cdot |O| + \alpha \cdot |O|$ number of network messages to re-create the index structure, where $\beta \cdot |O|$ is the total number of objects mapped to a node, and $\alpha \cdot |O|$ is the number of objects exchanged between the nodes to fix inaccurate Voronoi cells. Clearly, when the objects are randomly distributed, in the worst case every object o, $\forall o \in O$, is mapped to a different node; therefore $\beta=1$. Obviously, it is unlikely to have $\beta=1$ with large number of objects. In addition, once the pivots are identified, we take into account the data distribution of each node and try to assign a good pivot to each node so that more objects will be mapped to their correct node. As we discussed earlier, inaccurate cells are the border cells and there are very few of them. Therefore, α is a small decimal number.

Figure 10 shows the scale-out performance of ToSS-it. In this set of experiments, we vary the number of nodes in the cluster from 32 to 128, and evaluate the amount of time required for index creation using both uniform and taxi datasets. Note that as the number of nodes increases the total amount of data increases as well; however, as shown, the total execution time is **almost constant** (and always below 40 seconds!), which verifies the scalability of ToSS-it construction. We observe that there is a slight increase in the total execution time as we employ more nodes. This is because the pivot selection phase is performed at the master node, and hence, it takes a little longer for the master node to identify pivots.

Figures 11a and 11b illustrate the effect of varying the number of CPUs at each node when constructing ToSS-it for uniform and taxi datasets, respectively. With ToSS-it, since the number of network messages are minimized, the total index generation time is dominated by the LVD generation phase. Therefore, we enable multi-threading mode and generate LVDs using intra-node parallelism. As shown in both figures, as the number of CPUs increases, the LVD construction time improves significantly. We observe that the uniform dataset scales-up better than the taxi dataset. The main reason behind this difference is as follows. The LVD generation consists of two steps: Voronoi cell generation and refining inaccurate cells. With the uniform dataset, geospatial replication eliminates almost all of the inaccurate cells. Therefore, once the Voronoi cells are generated, the algorithm terminates. As there is no inter-node communication to refine Voronoi cells, the impact of increasing the number of CPUs is more significant. On the other hand, with the taxi dataset, by exploiting multiple CPUs we generate Voronoi cells faster; however, the inaccurate cells need to be refined in the second step. The inter-node communication becomes the dominant cost as we reduce the construction time of the Voronoi cells by exploiting multiple CPUs.

B. Range Query

In this set of experiments, we evaluate the performance of range queries. We define range (with respect to a query point) as the percentage of the entire space, and as proposed in [25] we consider three separate selectivity ranges: *small* (0.01%), *medium* (0.05%) and *large* (0.10%). We randomly generate the query points and send each query to a ToSS-it node in a round-robin fashion.

Figures 12a and 12b illustrate the effect of data distribution on range query for varying selectivity. We observe that as we use more nodes, the throughput increases linearly, which verifies the scalability of ToSS-it. The reason behind linear scalability is that the number of nodes has no impact on the number of network messages we send to process a query. With our approach adding more nodes only increases the cost of locating the node N_q (the node that contains q) and the partition search step executed at the beginning of query processing at each node, which are insignificant. As expected, as the selectivity increases, the query throughput decreases. This is because even though larger ranges are processed by multiple nodes in parallel, we need to retrieve more objects which degrades the performance. We also observe that

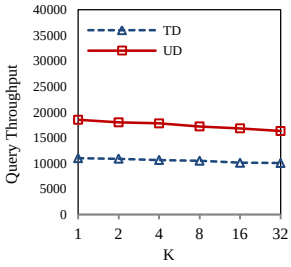

Figure 13. Effect of k on kNN Query (both datasets, N=64)

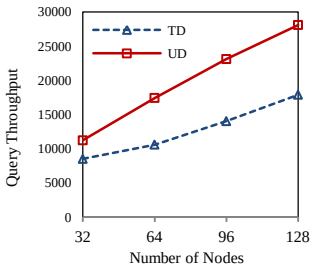

Figure 14. Effect of number of nodes on kNN query (both datasets, k=16)

increasing the selectivity has marginally more impact on the query throughput with the taxi dataset. This is because with the taxi dataset, there are more partitions in the areas where the data is clustered. The queries searching for the clustered regions need to examine more partitions which results in extra message exchanges.

C. *kNN Query*

In this set of experiments, we evaluate the performance of *k*NN queries. We randomly generate the query points and send each query to a ToSS-it node in a round-robin fashion. Figure 13 illustrates the impact of the value of k on the query throughput. In this specific experiment, we used 64 nodes, and varied the value of k from 1 to 32 and compare the query throughput of ToSS-it with skewed and uniform datasets. As shown, the performance is less for the taxi dataset; on the other hand, the effect of k is also less compared to the uniform dataset. In Figure 14, we keep the k value constant (k=16) and vary the number of nodes. As the number of nodes increases, query throughput increases almost linearly for both datasets.

VII. CONCLUSION

In this paper, we introduced a novel approach, ToSS-it, to index highly dynamic moving object data. The main idea behind ToSS-it is to build short-lived Voronoi based index structures instead of updating indexes with each location update from the moving objects. ToSS-it creates the throwaway indexes in a very short time by employing a parallel and distributed architecture that uses multiple nodes in a cluster and exploits multi-core CPUs in each node. Moreover, we proposed an efficient parallel search algorithm with which queries can be submitted to any node in the cluster without looking for the relevant node that may include the result-sets which provides decentralization. ToSS-it yields a very high query throughput and scales near-linearly with the number nodes in the cluster. In particular, we can generate ToSS-it for 64 million objects in less than 40 seconds using 128 nodes and process around 30,000 circular range queries per second. The total amount of time to construct ToSS-it from scratch remains almost **constant** even if the data size increases, which verifies the scalability of ToSS-it.

VIII. ACKNOWLEDGMENTS

This research has been funded in part by Award No. 2011-IJCXK054 from National Institute of Justice, Office of Justice Programs, U.S. Department of Justice, as well as NSF grant IIS-1115153, the USC Integrated Media Systems Center (IMSC), and unrestricted cash gifts from Google, Northrop Grumman, Microsoft, and Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the sponsors such as the National Science Foundation or the Department of Justice.

IX. REFERENCES

[1] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with MapReduce. In *CloudCom*, pages 9-16, 2010.

[2] S. Barahmand, S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*, 2013.

[3] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on Processing Spatial Data with MapReduce. In *SSDBM*, pages 302-319, 2009.

[4] S. Chen, B.C. Ooi, K. L. Tan, M.A Nascimento. A self-tunable spatio-temporal B+-tree index for moving objects. In *SIGMOD*, 2008.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 107-113, 2004.

[6] U. Demiryurek and C. Shahabi. Indexing Network Voronoi Diagrams. In *DASFAA*, pages 526-543, 2012.

[7] L. Ding, B. Qiao, G. Wang, and Chen Chen. An Efficient Quad-Tree Based Index Structure for Cloud Data Management. In *WAIM*, 2011.

[8] J. Dittrich, L. Blunschi, M. A. V. Salles. Indexing Moving Objects Using Short-Lived Throwaway Indexes. In *SSTD*, pages 189-207, 2009.

[9] R. A. Finkel, J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica* 4, pages 1–9, 1974.

[10] C. Gold and P. Angel. Voronoi Hierarchies. In *GISScience*, pages 99-111, *2006.*

[11] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47-57, 1984.

[12] http://software.intel.com/sites/default/files/m/1/5/f/Tera_Era.pdf

[13] C.S. Jensen, D. Lin, B.C. Ooi,: Query and update efficient B+-tree based indexing of moving objects. In *VLDB*, pages 768-779, 2004.

[14] C. Mouza, W. Litwin, and P Rigaux. SD-Rtree: A Scalable Distributed Rtree. In *ICDE*, pages 296-305, 2007.

[15] W. Litwin and M. A. Neimat. K-RP*S: A Scalable Distributed Data Structure for High-Performance Multi Attribute Access. In *PDIS*.

[16] W. Lu, Y. Shen, S. Chen, and B. C. OOi. Efficient Processing of k Nearest Neighbor Joins Using MapReduce. In *VLDB*, 2012.

[17] T. Nguyen, Z. He, R. Zhang, P. Ward. Boosting moving object indexing through velocity partitioning. *PVLDB* 5(9), pages 860–871, 2012.

[18] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. Spatial Tessellations, Concepts and Applications of Voronoi Diagrams. 2nd edition, 2000.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[20] M. Sharifzadeh, C. Shahabi. VoR-Tree: R-trees with Voronoi Diagrams for Efficient Processing of Nearest Neighbor Queries. In *PVLDB*, 2010.

[21] D. Sidlauskas, K. A. Ross, C. S. Jensen, and S. Saltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *SSTD, 2011.*

[22] I. Stanoi, M. Riedewald, D. Agrawal, and A. E.Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *VLDB*, 2001.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, pages 146-160, 2001.

[24] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. In *ICDE*, pages 1169-1184, 2004.

[25] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. 2010. Indexing multi-dimensional data in a cloud system. In *SIGMOD, pages 591-602, 2010.*

[26] http://developer.android.com/reference/android/location/LocationManager.html