Large Scale Indexing of Geofences

Flavio Cirillo, Tobias Jacobs, Miquel Martin, and Piotr Szczytowski
NEC Laboratories Europe
Kurfürsten-Anlage 36, 69115 Heidelberg, Germany
firstname.lastname@neclab.eu

Abstract—The modern smartphone and car concepts provide a fertile ground for new location-aware applications, ranging from traffic management to social services. While the functionality is partly implemented at the mobile terminal, there is a rising need for efficient backend processing of high-volume, high update rate location streams. It is in this environment that geofencing, the detection of objects traversing virtual fences, is becoming a universal primitive required by an ever-growing number of applications.

To satisfy the functionality and performance requirements of large-scale geofencing applications, we present in this work a backend system for indexing massive quantities of mobile objects and geofences. Our system runs on a cluster of servers, achieving a throughput of location updates that scales linearly with number of machines. The key ingredients to achieve a high performance are a specialized spatial index, a dynamic caching mechanism, and a load-sharing principle that reduces communication overhead to a minimum and enables a shared-nothing architecture. The throughput of the spatial index as well as the performance of the overall system are demonstrated by experiments using simulations of large-scale geofencing applications.

I. INTRODUCTION

Geofencing is becoming a standard feature of modern geographical information systems and increasingly receives attention from the research community. As GPS receivers are available at low prices and research on indoor location systems is currently taking off, there is a clear trend that more and more everyday devices are aware of their geographical position. This data is used by a multitude of applications including mobile advertisement, fleet management, traffic control, social networks, and security. Many of the use cases require permanent awareness of the set of objects within certain geofence zones; see [3, 8, 10, 13, 14, 15] for a number of recent application papers.

A geofencing system is a service which keeps track of the coordinates of mobile objects and continuously matches them against geofences, which are user-defined areas of interest. Whenever a mobile object enters or leaves a geofence, the system issues a notification. A special focus of this work is on mobile geofences, which are areas defined in relation to the coordinates of some mobile object. Whenever the object updates its location, the geofence is implicitly updated as well. Such geofences typically represent the immediate surrounding of an object, which is needed for example in applications where users wish to receive notifications on nearby friends.

One of the major challenges in those kind of scenarios is that the geofences change position as frequently as the mobile objects do, and thus the underlying spatial index of the backend system experiences write operations as frequently as read operations. This is because whenever an object changes location, a write operation is required to update the index, while read operations are needed to determine the objects that have entered or left geofences as a result of the movement. The spatial index therefore has to be designed such that the processing time of read/update pairs is minimized, while at the same time concurrent operations are supported to make the throughput scale.

In this work we present a spatial index tailored to meet the above requirements. We also describe how the index is used within a full-fledged geofencing service implementation that runs on a cluster of servers. The efficiency and scalability of our approach is evaluated in a number of experiments.

Related work. Geofencing has been discussed from an application point of view for about a decade, where applications range over a broad variety of different domains, see [3, 8, 10, 13, 14, 15]. Despite this substantial interest, the topic has only recently started to receive attention in database and algorithms research. The 2013 SIGSPATIAL Cup [1] called for algorithms that efficiently evaluate masses of point objects against a small number of geofence polygons. Here the main challenge lay on the complexity of the polygons, consisting of hundreds of edges and having holes. Impressing results have been achieved by a number of approaches [12, 18, 19]. We remark that our work addresses a different challenge, as we focus on the problem of designing a scalable index to maintain the state of large quantities of both mobile objects and geofences over time, triggering notifications whenever a geofence is crossed. In other words, we address geofencing as a data indexing problem, while the above work considers it as an algorithmic problem. The only work in our direction we are aware of is [4]. Here the indexing system is designed under the assumption that the geofences are not moving, we see no straightforward way to adopt it to the dynamic case.

As pointed out above, a key requirement of spatial indexes for geofencing is to efficiently support the same high frequency of read and write operations. Despite a long history of research on spatial data structures, we are not aware of previous work which explicitly addresses this requirement. What has nevertheless been studied in the recent years are spatial indexes with high performance in scenarios with frequent updates. Various techniques have been proposed to deal with the problem of possible inconsistencies due to concurrent access. In [6] the authors present a system where read-only index structures are created as snapshots and therefore queries and updates do not interfere. This has the side effect that query results show slightly outdated data. The snapshotting approach is also taken...
in [16] combining a grid index for updates with a snapshot grid for queries. Most recently, a grid index for point objects which completely avoids taking snapshots has been presented in [17].

The solutions of [6, 16, 17] are well suited for tracking systems for moving objects. The objects are frequently updating their location, and from time to time spatial queries are issued. Here the main challenge is to keep the query response time low while being able to process a high update throughput. For example, the query/update rate in the experimental setup of [17] is between 1:250 and 1:16.000. This differs from the requirements of our geofencing scenario not only in terms of the update/query ratio, but we also need to index spatial objects with extent (at least rectangles) as opposed to only point objects. We believe that grid indexes are not suited for this purpose, and we therefore take a different road.

Our contribution. We present an indexing system that is tailored to the requirements of large-scale mobile geofencing applications. Our target is to maximize the throughput of object location updates, where each object has an associated geofence and therefore multiple geofence crossings can be caused by each update. We design the underlying spatial index as an optimized variant of the R-tree and solve the concurrency issues by splitting the data among several instances of the tree (Section II). The performance of this index is evaluated in a dedicated experimental setup.

Furthermore, we present a backend geofencing system based on this index (Section III). The system runs on a cluster of servers and achieves a throughput of location updates that scales linearly with the number of machines. This is achieved by using dynamic caching mechanisms and a load-sharing principle that eliminates any need for communication between the different index servers and thus enables a shared-nothing architecture. Experiments with millions of mobile objects demonstrate the capability of our geofencing system.

II. SPATIAL INDEX

In this section we describe the spatial index employed by our geofencing system. The index needs to support functionality for the storage of objects with extent, where the two important operations are position updates of objects and window queries returning the set of all objects overlapping a given query window. Also insertion and deletion need to be supported, but as the frequency of these operations is comparably low we do take them into account for the basic design decisions. The main non-functional requirement is to maximize the throughput of read/update pairs, as motivated in the preceding section.

In general, to obtain a high throughput of operations on data structures two conditions have to be met. The first condition is that the computation time required for individual operations is low. The second condition is that the operations can be performed in parallel by several independent execution threads. The access control mechanisms that are necessary to prevent inconsistent states need to be implemented so as to avoid computational overhead and extensive wait cycles. For spatial data structures, meeting both conditions is a highly non-trivial challenge, and substantial research has been conducted in order to achieve satisfying solutions. For indexing point objects under high update rates, the series of articles mentioned in the preceding section [6, 16, 17] has proposed grid index variants. This is not suitable for our requirements, as objects with extent would have to be stored by multiple grid cells, and also for window queries one needs to access all grid cells overlapping with the window. On the other hand, locking mechanisms for the R-tree that are more fine grained than simply blocking access to the whole tree have been proposed; see e.g. [9]. The problems with the latter methods is that, for achieving consistency guarantees and avoiding deadlocks, one has to restrict the tree traversal directions and apply all changes only locally. Therefore, the possibilities to speed up the tree access methods are quite restricted. As also the locking mechanisms themselves incur a substantial computational overhead, we do not take the fine-grained locking approach either.

Instead, we split the data to be indexed among several instances of the index, where each instance only has a single working thread accessing it and therefore consistency problems do not occur. The splitting is done based on the geographical coordinates, such that each update and each window query has to be handled by only one index instance most of the time; details are given below.

The splitting approach has the advantage that the computational overhead due to locking mechanisms is negligible. Furthermore, this principle is completely orthogonal to any other speed-up technique, i.e., optimizations of the tree structure and tree traversal algorithms do not need to take concurrency into account. This holds in particular for the dedicated update operation we describe below. Finally, as we will see in Section III, our full backend geofencing system splits the data among different servers, so splitting among the execution threads on the individual servers is a straightforward application of the same principle and vastly simplifies the overall system architecture.

The main drawback of geographical splitting is that there is not guarantee that the objects will be evenly distributed among the different index instances, so some of the execution threads can remain idle while others are overloaded by requests. We believe however that for geofencing scenarios the distribution of the objects in the geographic area under observation is predictable, so the splitting can be done accordingly.

In the subsequent paragraphs we present improvements of the original R-tree that increase the throughput of read/update pairs. After that, we describe in more details how the geographical splitting is implemented. The section is concluded by an experimental study evaluating the performance of the index in a multi-processor environment, before in Section III we finally present our full-fledged geofencing system.

Specializing the R-tree. Due to space constraints in this extended abstract we do not give a detailed introduction of the R-tree. For understanding the explanations it suffices to know that the R-tree stores pointers to the spatial objects (points or objects with extent) in its leaves. Every internal R-tree node is associated with the minimum bounding rectangle (MBR) of all objects stored in the subtree under it, so descending the R-tree top-down corresponds to a step-wise refinement of the search area. The number of children each tree node can have is parameterized by a minimum and a maximum; as soon as the number of children leaves this given range, tree restructuring mechanisms are applied. An introduction to the R-tree can be
found in [7] and in introductory textbooks on spatial databases.

What our improvements address is the lack of a dedicated update operation. The authors of the original R-tree [7] did not envisage an update operation at all; updates were to be implemented simply by a delete operation followed by an insertion. Due to efficiency reasons, this can only be tolerated when update operations are executed sporadically. The inefficiency comes from the fact that the R-tree has to be traversed twice, each time triggering possible modifications of the tree structure.

A number of solutions to this issue have been proposed in literature, and we adopt and extend concepts described in [11]. Our update mechanism uses an auxiliary index for the R-Tree. This index, denoted the parent index, is a hash map which provides direct access to the leaf node which holds the object to be updated. The index additionally holds a pointer to the parent for each tree node, so that the R-tree can be traversed bottom-up from leaves to the root.

Upon the arrival of an update request, we use the parent index to look up the leaf node containing the corresponding object. The algorithm then checks whether that leaf node can accommodate the new location of the object without growing its own MBR. If this is the case, then only the location of the object within the leaf is updated. Otherwise, the algorithm checks whether the removal of the object from the leaf will lead to an underflow (i.e., the node will contain less objects than permissible). If the underflow condition is met, then the update is performed as a pair of delete and insert operations. The original procedure described in [11], before checking the underflow condition, still tries to update the object within the leaf node whenever the growth of the leaf node’s MBR is not greater than a threshold defined by some predefined factor ϵ. We decided not to implement this additional check, as continuous movement of objects at low speed can otherwise lead to overgrowth of the leaf node’s MBR.

If the object cannot stay inside its previous leaf’s MBR and there is no underflow, then our algorithm performs a check whether the object’s new location is contained within the MBR of the leaf’s parent. If this is the case, then the algorithm assigns the object to some sibling of the original leaf. Among all siblings that have capacity for another object, the one is selected whose MBR has to grow the least in absolute number.

This implicitly gives preference to siblings into whose MBR the object already fits without growth, and we remark that also the previous leaf of the object can happen to receive the object again. The latter kind of local modification is where our algorithm goes beyond the method in [11]. Note that also top-down re-insertion would eventually enlarge the MBR of some sibling leaf node, so our method achieves the same result more efficiently.

We also make use of the parent index when re-inserting the object top-down, as the leaf node from which the object is to be removed does not have to be searched in the tree. The additional index is further used for traversing the tree from the leaf to the root in case the MBRs have to be adjusted.

Splitting the data. We describe how we split the data among several R-tree instances in order to make use of multiple processors (or hardware execution threads on one processor) without consistency issues. The explanation below relates to the pure index structure, whereas in the subsequent section we describe how a complete geofencing index is realized using an extension of the same principle. As we use a one-to-one mapping of working threads to instances of the index, we use the terms worker and index instance interchangeably.

The assignment of the data to workers is done based on the geographic location. The universe is split into regions and each region is represented by a worker performing operations on a local R-tree instance. Each worker only services query and update requests for the region of the universe that it represents.

Special attention needs to be paid to the border zones between the regions, as here the objects and the query windows can overlap regions of multiple workers. The index is designed such that any query can always be answered by a single worker; therefore, objects inside the border zones have to be redundantly stored. The replication mechanism in our solution is parameterized by three assumptions about the geofencing scenario. These assumptions relate to the maximum object radius (i.e. distance between object center and boundary) o, maximum query window radius q, and maximum distance d an object is expected to move between two consecutive location updates.

Each worker w indexes all objects which can overlap with a query window that is centered within the worker’s area of responsibility Uw, so that w can answer all those queries. In addition to all objects being centered inside Uw, this includes all objects in distance at most q + o from the border of Uw.

The geographic splitting approach additionally necessitates the implementation of a dispatcher component which assigns query and update requests to the proper worker. As described above, queries are always assigned to the unique worker responsible for the region that contains the center of the query window. Update operations, in contrast, are possibly assigned to multiple workers. This happens when the updated object position (i.e., the center of the object’s MBR) is in the border zone. More specifically, the update operation is assigned to each worker whose region R has a distance of at most o + q + d from the new object position. This way each worker w will also be made aware of the situation when an object moves far enough away from Uw to be removed from its index instance.

Experimental setup. The objectives of the subsequent experiments are to determine the optimal configuration of our index, and to evaluate the throughput in scenarios where both mobile objects and geofences frequently change location. Unless otherwise stated, the throughput is evaluated in terms of how many location updates can be processed per second, where each such location update is handled by an update operation on the index and a query to find out new geofence crossings.

We use three different data sets representing various distributions and movement patterns of mobile objects. In all of them, each mobile object has a geofence attached to it. This means that the geofence location needs to be updated together with the object movement, and a spatial query reveals which other geofences the object has entered or left.

The first data set, denoted UNIF, is generated by the most simple random model. The initial object positions are chosen uniformly at random, and the movement is simulated by a
random walk. The default values of the scenario are: a universe that comprises the city of Heidelberg in Germany, with an area of roughly 1.4 km x 3.5 km; the geofences around the objects are 40x40 m rectangles (0.032% of the universe size). Each latitudinal movement is chosen as a uniform random value in the range 0.015-0.035% of the latitude span of the universe, and analogously for longitudinal movements.

The second data set follows the same random walk model, but here the initial object positions are generated using Zipf distribution. More specifically, the middle of the universe is chosen as the center point and the object positions are then generated such that 70% of all objects are within 30% of the maximal possible distance from the center. As the simulation only runs until each object has updated its location once, the effect of the random walk evening out the object distribution after some time is negligible. This data set is denoted ZIPF.

The final data set employs a model which simulates a specific application scenario. We used SUMO [5], a traffic simulation, running TAPAS Cologne [2], a simulation of the city of Cologne. This setup provides simulations of the movements, speeds and relative positions of different vehicle types on a week day between 6 and 8am. The simulation includes cars queuing at traffic lights and making decisions on unmarked intersections, which provides for varying speeds and the creation of congestion along the roads. In order to assign geofences to the objects, we generate a rectangle around each point starting from an apotheme (i.e. distance from center of the rectangle to the middle of the top side of the rectangle). A speed factor scales the value of the apotheme proportional to the speed of the vehicle, and the aspect ratio is used to compute the longitudinal extent of the geofence. The resulting rectangle is centered on the location of the vehicle. In the concrete setup we use an apotheme of 65m, an aspect ratio of 2 (i.e the longitudinal side is twice the latitudinal one), and a speed factor of 0.1. This third data set is denoted SUMO.

For the execution of the above described experiments we use a commodity PC with an Intel i5-2400 CPU running at 3.1GHz. The CPU features 4 hardware CPU cores, which we can turn on and off to measure scalability. The experimental machine is equipped with 8GB of RAM and is running Java Virtual Machine, which serves as an execution environment for the Java implemented benchmark.

Computational results. In a first set of experiments we target to optimize the parameters of our load sharing mechanism for the optimized R-trees. The influence of the number of worker threads (and thus the number of regions the universe is split into) on the throughput is evaluated in Figure 1 for different hardware configurations, using the UNIF data set with 100,000 objects. A larger number of workers on the one hand tends to increases the probability that the available processor cores are fully utilized, while on the other hand effectuates a higher probability that objects are indexed by multiple workers. Also the overhead due to context switching increases as soon as the number of workers exceeds the number of available processor cores. Figure 1 shows that the optimal throughput is achieved when the number of workers is about the same as the number of processor cores. As predicted, the throughput starts to decrease as soon as the workers start to substantially outnumber the processor cores. In the remaining experiments, the number of workers will be fixed to the optimum value for the respective number of processor cores.

In Figure 2 the number of workers is fixed to the optimum and we vary instead the number k of clients making requests in parallel. In each experiment k clients are executing a loop where they make a request and wait for the response before issuing the next request. The higher the number of clients, the higher tends to be the number of different worker threads that concurrently perform operations, and thus the higher is the throughput we expect. Figure 2 reveals that the available hardware is fully utilized when the number of concurrent clients is about twice the number of processor cores.

In the remaining experiments we fix the number of worker threads to the optimum for the respective hardware configuration and use a sufficient number of parallel clients so that the maximum throughput is reached. In doing so we simulate the situation where the number of incoming location updates the geofencing system receives is at least as high as the throughput it can achieve. Table 1 examines all three data sets. The numbers show that the throughput scales with the number of processor cores, although the scaling is not perfectly linear due to effects like redundant storage and multi-threading overhead. The observation is noticeably consistent for all three data sets, so the distribution of locations does not seem to play a specific
role for the scalability.

In the experimental setup up to now, we have measured the throughput in terms of location updates per second, where each location update includes both an update operation and a query operation on the spatial index. We now deviate from this scheme and investigate in Figure 3 and 4 the influence of the relative frequency of queries on the throughput. The figures show that for both UNIF and ZIPF distributed data the throughput is higher by a factor of up to three when only update operations are involved. Interestingly, when using more processors the relative difference becomes less pronounced. Also note that most of the difference is already explained by the fact that the number of operations per location update is two on the right-hand side of the plot and one on the left-hand side.

In Figure 5 we study the influence of the geofence selectivity on the throughput that our index can achieve. Larger geofences lead to larger result sets and therefore tend to decrease the throughput. However, as it can be seen on the figure, the effect on the overall system performance is only moderate, and in particular the scalability is not negatively impacted.

In a final experiment, the influence of the population size (e.g. the number of objects stored in the spatial index) on the throughput is studied. The results are shown in Table II, showing again that the population size has a moderate influence on the throughput, despite the fact that more objects also lead to larger result sets from the queries. In particular, the throughput does not seem to decrease further as soon as more than 150,000 geofences are indexed.

III. GEOFENCING INDEX

In this section we present our full geofencing system. In terms of functionality, the difference to the spatial index presented in the previous section is the way users interact with it. On a spatial index users issue query and update operations, whereas with a geofencing system users interact by sending location updates and subscribing to geofences. In other words, a spatial index can be considered as data base, whereas a geofencing system is a specific type of event processing system. Although the principle architecture of the geofencing system is similar to that of the spatial index, a number of additional mechanisms are in use here. As the overall system is slightly more complex, we use a more formal notation in this section to avoid ambiguities.

From an architecture point of view, the system consists of several indexing machines and at least one machine serving as the event dispatcher. The event dispatcher is stateless, so it can be easily replicated in order not to become a bottleneck. For simplicity of description we assume that there is only a single event dispatcher. Each of the indexing machines hosts a number or workers, where each worker will be responsible for a certain area of the universe; details will follow later. The number of workers a machine hosts depends on the number of hardware threads available. As this number can differ

<table>
<thead>
<tr>
<th>number of processor cores</th>
<th>UNIF</th>
<th>ZIPF</th>
<th>SUMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75.197</td>
<td>100.180</td>
<td>101.358</td>
</tr>
<tr>
<td>2</td>
<td>147.782</td>
<td>200.231</td>
<td>203.329</td>
</tr>
<tr>
<td></td>
<td>259.454</td>
<td>352.294</td>
<td>347.208</td>
</tr>
</tbody>
</table>

TABLE I. THROUGHPUT ON THE DIFFERENT DATA SETS FOR VARYING NUMBERS OF PROCESSING CORES.

<table>
<thead>
<tr>
<th>population size</th>
<th>throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>50,000</td>
<td>223,001</td>
</tr>
<tr>
<td>100,000</td>
<td>186,466</td>
</tr>
<tr>
<td>150,000</td>
<td>155,276</td>
</tr>
<tr>
<td>200,000</td>
<td>159,263</td>
</tr>
<tr>
<td>250,000</td>
<td>152,253</td>
</tr>
</tbody>
</table>

TABLE II. INFLUENCE OF THE POPULATION SIZE ON THE THROUGHPUT (UNIF DATA SET).
from machine to machine, our system supports heterogeneous hardware environments.

In our data model, each object has an associated geofence defined around it, where geofences have polygonal shape. As we address the challenge of maintaining many dynamic geofences rather than the challenge of complex polygons (see [1]), we restrict the number of polygons edges to eight. The geofencing system processes events, where each event represents the movement of one object, including the movement of its associated geofence. More specifically, the information represented by any event \( e \) consists of the object identifier \( o_e \), the previous location \( l_e \) of object \( o_e \), and the new location \( l'_e \) of the object. Additionally, \( e \) contains information about the previous geofence \( f_e \) and the new geofence \( f'_e \).

Whenever any object \( o \) enters or leaves a geofence, a notification is expected to be triggered by the geofencing system. Therefore, an event \( e \) can cause notifications for four different reasons: \( o_e \) enters new geofences, \( o_e \) leaves geofences, objects enter the geofence defined by \( f_e \) due to its movement to \( f'_e \), or objects leave the geofence due to the movement. The indexing system needs to make sure that each expected notification is triggered once and only once, despite the partly redundant storage of geofences.

Similar to the preceding section, our index makes use of two assumptions about objects and geofences, which we believe are satisfied by all typical mobile geofencing application scenarios. The first assumption is that, whenever an objects sends a location update, the distance between the new location and the previous one is bounded by some number \( d \), where \( d \) is small compared to the size of the area each worker is responsible for. The second assumption relates to the maximum size of mobile geofences. We assume that the distance between an object and the border of its geofence is upper bounded by a second constant \( q \), where also \( q \) is small compared to the size of the area each worker is responsible for. Even though in real application scenarios there might not be a guarantee that these parameters hold in 100% of the cases, exceptions can still be handled by special mechanisms that are computationally expensive but need to be executed only rarely.

The universe \( U \) is split among the different workers, where \( U_w \) denotes the universe part for which worker \( w \) is responsible. Denoting by \( W \) the set of workers, the splitting is done such that the areas \( U_w \), \( w \in W \) are pairwise disjoint and \( \bigcup_{w \in W} U_w = U \). For each event \( e \) there is a unique worker \( w \) into whose area of responsibility the object \( o_e \) moves; we denote this worker as \( w_e \). Note that \( w_e \) is a derived value and not contained in the information encoded in the event \( e \). Formally speaking, \( w_e \) is the unique worker \( w \in W \) with \( l'_e \in U_w \). The underlying principle of our distributed index is given as follows:

**Design Principle.** For each event \( e \) the worker \( w_e \) is responsible for issuing all notifications triggered by the object and geofence movement represented by \( e \).

By this principle it is straightforwardly guaranteed that each expected notification will be triggered by exactly one worker, without any further need for coordination. This requires, however, that each worker \( w \) is aware of any notification it has to trigger, even if some of the corresponding objects are outside its area \( U_w \).

In order to satisfy this requirement, each worker \( w \) indexes an object set slightly larger than the set of objects within \( U_w \). More specifically, worker \( w \) indexes all objects (and their corresponding geofences) which are in distance of at most \( d+q \) from some point of \( U_w \). Intuitively, \( w \) is aware not only of all objects in \( U_w \), but also in a border zone of width \( d+q \) around \( U_w \). We denote this extended area by \( V_w \). Formally, \( V_w := \{ x \in U \mid \exists y \in U_w : \text{dist}(x,y) \leq d+q \} \).

This definition of \( V_w \) effectuates that for any event \( e \) with \( l'_e \in U_w \) the worker \( w \) will be aware of any geofence that contains the previous object location \( l_e \). This is because, by the assumption about the maximal movement distance, \( l_e \) has a distance of no more than \( d \) from \( U_w \), and thus, by assumption about maximal geofence size and the triangle inequality, any geofence containing \( l_e \) corresponds to an object whose distance from \( U_w \) is no more than \( d+q \) and thus is contained by \( V_w \). The same kind of reasoning shows that any object inside the previous geofence \( f_e \) is as well contained by \( V_w \).

To support this design, the event dispatcher forwards each event \( e \) to the set of workers \( \{ w \in V_e \mid l_e \in V_w \wedge l'_e \in V_w \} \) whose extended area contains the previous or new object location. By the assumption about the size of \( d \) and \( q \) in comparison to the universe size, events are forwarded to only one worker in the majority of cases. Only when the movement takes place near the border between the areas of several workers more than one worker needs to be made aware, and even then the number of such workers is limited to a small constant.

It remains to describe the implementation of the individual workers. Each worker maintains four indexes. In the **polygon index**, the exact geofence definition is stored for each object. This index is a simple key-value store like a hash table. In the **spatial index** the minimum bounding rectangles (MBR) of the geofence polygons are stored. This index supports window queries, and we use the optimized R-tree described in the preceding section. Third and fourth, each worker maintains an index storing for each geofence the set of objects contained by it, and another index reversely storing for each object the set of geofences it is contained by. These two indexes are denoted **geofence cache** and **inverted geofence cache**, respectively.

Assume that a worker \( w \) receives an event \( e \), where both \( l_e \) and \( l'_e \) are contained by \( V_w \), so the object \( o_e \) and its geofence have been indexed by \( w \) before. Upon arrival of \( e \), the worker makes a single window query to the spatial index to retrieve the set of geofences whose bounding boxes overlap with the bounding box of \( f'_e \). We denote the result set of this query as \( O_q \). Let further \( O_f \) be the set of objects that were previously contained in the geofence of the object \( o_e \). The set \( O_f \) is retrieved by the worker from its geofence cache. Finally, let \( F_l \) be the set of geofences that contain the previous location \( l_e \) of object \( o_e \). This set is retrieved from the inverted geofence cache. In order to detect the necessary notifications (which are only triggered when \( l'_e \in U_w \)) the worker performs the following operations:

I. For each object \( o \) in \( O_q \), if \( o \) is contained by the geofence \( f'_e \) and is not contained by \( O_f \), then a notification is detected that \( o \) has entered the geofence of \( o_e \).

II. For each geofence belonging to an object \( o \) in \( O_q \), if \( o_e \)
is inside the geofence but the geofence does not appear in $F_l$, then a notification is detected that $o_e$ has entered the geofence belonging to $o$.

**III.** For each object $o$ in $O_f$, if the object is not a member of $O_q$, or if the object is a member of $O_q$ but $o$ is not contained by the new geofence $f'_e$, then a notification is detected that $o$ has left the geofence of object $o_w$.

**IV.** For each geofence $g$ in $F_l$, if the object $o$ of geofence $g$ does not appear in $O_q$, or if $o$ does appear in $O_q$ but $l_e$ is not contained by the geofence $g$, then a notification is detected that $o_e$ has left the geofence $g$ of object $o$.

Due to the fact that $O_q$ not only contains all objects inside the new geofence $f'_e$, but also all objects whose geofences contain the new object position $l'_e$, the above procedure guarantees that the worker $w_e$ (which is the one to trigger notifications) detects all necessary notifications.

Whenever a notification is detected, then the geofence cache and the inverted geofence cache are updated accordingly by the worker. The updates are also done even when the worker is not the one to trigger notifications. The worker finally also updates the polygon index with the new geofence polygon $f'_e$, and the update operation on the spatial index is called in order to store the new bounding box of the geofence.

In case of arrival of an event $e$ with $l'_e$ outside $V_{w_e}$, we know that the new object location $l'_e$ must be outside of $U_{w_e}$, so no notifications need to be triggered but only the index structures need to be updated. The same procedure as above is applied with $O_f = \emptyset$ and $F_l = \emptyset$. Also upon arrival of an event $e$ with $l'_e$ outside $V_w$, no notifications need to be triggered by worker $w$. The above procedure is then applied with $O_q = \emptyset$.

**Experimental setup.** For the experimental evaluation of the geofence indexing system we use a data set of uniformly distributed objects over a universe of 20km x 10km. The fences around the objects cover 0.000625% of the universe on average; hence their average area is 1250 $m^2$. Object movement is simulated by a random walk whose speed is a random value between 0 and some upper bound of 36m/s. As compared to the experiments in the previous section, the number of objects in this experimental setup is by an order of magnitude larger.

The simulation is executed on varying sets of Virtual Machines (VMs) deployed using the Cloud management software Openstack. For simulation purposes we vary the number of VMs as well their configuration in terms of how many CPUs (and thus workers) each VM has available. Independently of the CPU configuration, each machine has 5 GB of RAM and uses a Java Virtual Machine as the execution environment.

**Computational results.** We evaluate the performance of the geofence indexing system presented in this section. To verify the scalability claim with respect to the throughput, we investigate a variety of system configurations regarding the number of servers and the number of workers on each server.

In a first series of experiments we keep the number of objects and geofences constant while varying the number of indexing servers in the setup. This test has been conducted twice; in one case the total number of objects is one million and each indexing server hosts two workers, and in the other case the number of objects is two million and each server runs four workers. The results of both experiments are depicted in Figure 6.

The figure plots the average throughput per server and the overall throughput against the number of servers. The plots show that the throughput of the individual servers does not suffer when a larger cluster is instantiated, and therefore the total throughput scales linearly with the number of indexing servers. This observation holds both for the case of two workers and four workers per server.

In terms of absolute numbers, the performance of the four-worker servers is not better than the servers hosting two workers, taking into account that the further have to maintain twice the number of objects. It therefore seems that the number of objects significantly influences the throughput.

When comparing Figure 6 to the experimental results of the preceding section, it becomes apparent that the throughput is by an order of magnitude less than the throughput of the bare index structure. This observation can be explained by the larger number of objects used in this experiment, and by the fact that a considerable number of additional computational steps are required here in addition to the R-tree operations.

To further quantify the influence of the number of objects on the throughput we conduct a second series of experiments. This time we keep the number of objects proportional to the number of workers, so that the average number of objects each worker has to handle remains roughly constant (recall however the redundant storage of objects in the border regions). The results are visualized in Figure 7.

One can observe that the average throughput of the individual servers is decreasing with the total number of working threads. Our explanation is that with more mobile objects the number of geofence crossings per location update is substantially higher, and thus more notifications need to be issued. In other words, while the number of objects per worker is constant, the objects that an individual worker has to index are concentrated on a smaller area, and thus they cross each other’s geofences more often. The overall system throughput first slightly increases when the number of servers is doubled from two to four (and the number of objects is doubled accordingly), but when further increasing the number of servers and objects...
the total throughput begins to degrade slightly. That decrease can be explained by the negative effects of the need to store some of the objects redundantly on several servers.

IV. SUMMARY AND CONCLUSION

In this work, we have investigated the indexing problem in dynamic geofencing for large-scale application scenarios. We have developed a spatial index tailored to the functionality and performance requirements of geofencing, and we have presented a backend geofencing system that is based on the same design principles. In an experimental study we have shown that the throughput of the spatial index scales with the number of hardware threads it runs on. Although the scaling of the index is not perfectly linear, the experiments with the full geofencing system show that the negative effect on the overall scalability is negligible.

What has become apparent from the experimental results is that, in order to maintain the same throughput for an increasing population of objects, the number of servers needs to be kept proportional to the number of objects. We believe that this is the best we can hope for, as already the output size, i.e. the number of geofence crossings, is proportional to that number.

A topic we leave to future work is to combine our system with efficient methods for evaluating more complex geofence shapes, so that the overall system response time is kept within acceptable bounds. Although the geofencing application requirements admit a certain tolerance regarding the response time, it is still useful to be able to guarantee some upper bound.

REFERENCES