Journal of Information Processing Vol.22 No.3 418–424 (July 2014)

[DOI: 10.2197/ipsjjip.22.418]

Invited Paper

Geocookie: A Space-Efficient Representation of Geographic Location Sets

Peter Ruppel[1,a) Axel Küpper[1,b)

Received: November 1, 2013, Accepted: March 13, 2014

Abstract: A new approach is presented that allows a user agent to compute a compact browser cookie-like character string (called Geocookie) for a third party, whereby for any given geographic location the third party can either infer that the location is definitely not covered by the Geocookie, or it can infer that the location is probably covered by the Geocookie, depending on whether the user agent has or has not included the location beforehand. The approach extends the concept of a Bloom filter and combines it with Geohashes, thus making it possible to store information about visited geographic locations in the filter. Geocookies can be used in many different scenarios for location-based queries and location-based services, whenever a user agent wants to inform a third party about a set of visited locations such that the third party can compile a result that either favors or excludes these locations. In contrast to existing approaches such as session cookies that are mapped to server-sided stored location trajectories, Geocookies provide a compact and privacy-preserving structure which does not reveal the actual set of all visited locations, but provides a one-way check function which can be used by the third party to evaluate given locations against the Geocookie. In addition, Geocookies provide plausible deniability in case of location matches. This paper introduces a formal definition for Geocookies together with a discussion on practical applications and embedding into HTTP headers.

Keywords: data structures, spatial data, data efficiency, location privacy, location-based services

1. Introduction

Information retrieval for location-based applications often involves the filtering and processing of datasets on the basis of user-dependent trajectories and sets of visited places, respectively. For example, a system that recommends geographic points of interest (POIs) to users may exclude suggestions for places the users have already been to, but at the same time favors new POIs that are near to the ones that have been visited before. In another scenario a service omits location search results that are near to locations seen before by the user, or it limits the results to the places that are well known by the user. Both scenarios usually require the application to be aware about the set of places visited by the user, i.e. that the set is either stored within the application context and linked with a user identifier, or that the set is forwarded to the application with every request. Either way the full disclosure of a set of visited places is unfavorable, mainly because of the following two reasons: on the one hand the set can grow very large in case the user visits a lot of places. Especially when the set is provided on a per-request-basis, this can render the queries impracticable. On the other hand an even more challenging issue is the possible traceability of trajectories which concerns the location privacy of users.

The general problem statement for such application scenarios can be formulated as follows: given a set $L = \{l_1, l_2, \ldots, l_n\}$ of geographic location elements and a location element p, determine whether there is at least one element in L that covers p. Additionally, the problem can be extended to determining whether there is at least one element in L which is apart from p no more than a given distance d, where d corresponds to e.g., the Euclidean distance. If both L and p are stored in the same system, such queries can be easily carried out by any current spatial database. However, the complexity rises when there is a large number of location sets, e.g., one set L per user in a many-user system, together with a high rate of checks to be performed. Such a scenario is the typical challenge for any location-dependent messaging and geofencing application. Location-dependent messaging systems deliver messages to recipients only at certain locations, possibly considering additional contextual constraints that have been defined when the message was sent. Geofencing applications trigger events based on the entering, leaving, staying or the recurrence of mobile targets in certain geographical regions, called geofences.

Besides performance there is a second challenge for future location-based services: the characterization and portability of location sets. Given that more and more location-dependent tasks are fulfilled on mobile devices, it is required to process e.g., location trajectories not only in central databases but also on the device. At the same time there is a lack of approaches to port a location set from one application to another, for example when a mobile user wants to use her personal set of visited places as input in different location-based applications. Thereby the issue is not merely the recording of locations in a standardized format such

[1] Technische Universität Berlin, Department of Telecommunication Systems, Service-centric Networking, Berlin, Germany, http://www.snet.tu-berlin.de

[a] peter.ruppel@tu-berlin.de

[b] axel.kuepper@tu-berlin.de

as the XML-based *GPS Exchange Format*. The point is rather to provide a *compact* data structure that holds a *scalable* amount of geographic location elements, especially when the location set is to be a) frequently exchanged between two parties, b) frequently matched against other location elements, and c) frequently extended by additional location elements.

Efficiently determining whether a given item is a member of a set has been extensively researched in the past. One of the most prominent approaches is the hash-coding method originally introduced by Bloom [2] (known as *Bloom filter*), which has been applied to many fields of applications, e.g., for recognizing known malicious Web URLs or for testing whether a given file is already present on a disk before performing the actual seek operation. A Bloom filter allows to efficiently identify nonmembership of an item in a set S with a certain (adjustable) false-positive rate for the membership detection. It relies on an m-bit vector V (initially set to 0) and k independent hash functions $h_1, \ldots, h_k$. An item x is added to the filter by setting the bits at $h_1(x), \ldots, h_k(x)$ in V to 1. The same way membership of an item y can be tested: if any of the bits at $h_1(y), \ldots, h_k(y)$ in V is 0 then y is identified as a nonmember, and a probable member otherwise. The false positive rate of the Bloom filter depends on the size of V, the number k of hash functions and the number $|S|$ of elements, and can be adjusted to the specific application requirements.

However, when comparing and matching *geographic location elements* it is not only required to identify exact matches, for example comparing two pairs of latitude/longitude coordinates, but also to take into account matches as a result of being *covered* by other location elements or because of their *spatial proximity* to the element under consideration.

The goal of this paper is to introduce a novel combination of existing Bloom filter and location hashing approaches that allows to efficiently evaluate spatial containment of a geographic location in a set of previously collected locations. Generally speaking the anticipated application domains are both native and web-based LBS applications that rely on filtering geographic locations in order to compile their responses.

The remainder of this paper is structured as follows: the following section introduces the fundamentals of the data structure and describes how location elements can be added and tested. Also the false positive rate of the proposed approach is discussed. Section 3 highlights different use cases and exemplifies numbers for the utilization of the approach within HTTP, followed by a discussion in Section 4. Section 5 concludes the paper and highlights future work.

2. Geocookies

In the following a new data structure, called *Geocookie*, is introduced. The approach is based on the well-known concept of a *Bloom filter* [2] and combines it with *Geohashes*, which have been introduced by Niemeyer [10]. The main idea behind the Geocookie is to provide a space-efficient method for representing a set of geographic locations, and to allow for fast membership tests on the set, i.e., to check whether a certain geographic location is covered by any location element in the Geocookie. Once a location element has been added to the Geocookie, it cannot be removed, and an increasing number of elements in the Geocookie will increase the false positive rate of membership tests (but the false positive rate can be adjusted as described below and non-existing members are always identified correctly). This is why Geocookies are not intended for perfect checks on geographic object, but rather for various optimization purposes in the domain of location data queries.

The Geocookie is a space-efficient data structure that can be used to test whether a geographic location is spatially covered by at least one member in a set $L = \{l_1, l_2, \ldots, l_n\}$ of geographic locations that have been added to the Geocookie beforehand, where every $l_i \in L, 1 \leq i \leq n$ is described by a polygon of arbitrary size, thus making it possible to also represent single points.

A Bloom filter is used to store the elements and to test for membership: initially, a Geocookie is an m-bit vector V, whose bits are all set to 0. In addition there need to be k independent hash functions defined. For a given geographic location element l the Geocookie provides two operations:

- *add*(l): adds the geographic location l to the set L. Adding an element to L will result in some of the bits in V to be set to 1, which depends on the hash functions and the mapping of 2D geographic areas onto one-dimensional identifiers, which is discussed below in Section 2.1.
- *isCovered*(l): returns *false* if $\nexists l_i \in L : l_i \supseteq l$ (i.e., there is no element in L that fully covers l), or *true*, if the geocookie already contains an element (area) that *probably* covers the given location l. The false positive rate is a function of the Geocookie's size and the chosen mapping function, which is discussed below in Section 2.2.

Furthermore, we assume that the set L of geographic locations in the Geocookie is

- *monotonically increasing*, i.e., the number of elements in L may increase over time, but elements are not removed,
- *diverse*, i.e., the polygons stored in L significantly vary in size and geographic distribution,
- *possibly overlapping*, i.e., any pair of elements in L might have a non-empty intersection for the corresponding areas, and
- *approximative*, meaning that the elements in L may be subject to minor inaccuracies when being mapped onto the real world.

2.1 Adding Locations to the Geocookie

Existing Bloom filters already provide methods to add and test e.g., URLs or other textual elements in an efficient manner. However, when storing and testing geographic locations it is not only required to recognize an *exact* area, but also to test whether a given location is *inside* a previously stored area. Geographic locations are typically described e.g., by a coordinate, a circle, or by a polygon that represents an area. In either way they can be approximated by some geometric object such as the minimum bounding rectangle or minimum bounding circle. Our approach is to rely on such approximations when performing the comparison between locations, thus trading some accuracy for efficiency.

We utilize the existing concept of a *Geohash* to approximate and process geographic locations within the geocookie.

The Geohash is a simple and brilliant method invented by Niemeyer[10] for geocoding a pair of latitude/longitude coordinates into a shorter hash. For example, the WGS84 coordinate 52.513061, 13.320048 can be encoded into the string *u336xpeqg85d*. The encoding is achieved by recursively dividing the latitude (and longitude respectively) into two intervals. Starting from the intervals −90° to 0° and 0° to 90° a binary 0 is noted if the latitude falls into the lower interval and binary 1 in case of the upper interval. In the second round the according interval 0° to 90° is further subdivided into the intervals 0° to 45° and 45° to 90° and so on. The two resulting bit sequences are then alternately interleaved and the result is encoded to Base32[1]. Geohash encoding brings several advantages: the accuracy of a geohash can be coarsened by just removing characters from the end of the hash. Furthermore, two hashes with the same prefix usually point to the same region, i.e., geohashes can be used for rough estimate proximity searches. However, in certain cases two locations that are in close proximity can result in totally different hashes, e.g., if one is directly west of the prime meridian and the other location is directly east of it.

The accuracy of a position information given in geohash notation depends on the length of the geohash. For example, a geohash of length 4 is accurate ±0.087° in the latitude and ±0.18° in the longitude, which is equivalent to ±20 km. Extending the length to 6 will increase the latitude accuracy to ±0.0027°, longitude accuracy to ±0.0055° and ±0.61 km overall.

A first and straightforward approach for realizing the *add()* operation of the Geocookie could be as follows: for a given location l find its minimum bounding rectangle l_{mbr}. Then compute a geohash $cp(l)$, called *common prefix* of l, which is defined as the geohash that has the same prefix for all points inside l_{mbr}. Then add $cp(l)$ to the internal Bloom filter of the geocookie.

Obviously this naive approach has several disadvantages: first of all it does not scale well in case of many single-point locations that are all next to each other. Consider e.g., locations originating from a Global Positioning System (GPS) receiver that rapidly reports many locations which are all apart from each other by only a few meters. Such data points will unnecessaryly fill the geocookie and lead to an undesirable false positive rate later on. Also this naive approach does not properly handle all kinds of polygons. For example a diagonal polygon that represents a larger coast area will result in a large minimum bounding rectangle and thus in a short geohash, which does not properly reflect the actual area. A third drawback is that redundant information might be added in case some (smaller) location is added, but some other (larger) area, which contains the former, is already present in the filter.

The issue of multiple nearby locations can be solved by introducing a maximum length $cp_{\max}$ for the common prefixes. Depending on the application requirements, $cp_{\max}$ could be set e.g., to 8, which is equal to an accuracy of ±0.000085° for the latitude and ±0.00017° for the longitude, or ±19 m in total.

Large areas can be handled by introducing a minimum length $cp_{\min}$ for the common prefixes. In case $|cp(l)| < cp_{\min}$, then multiple geohashes with longer prefixes need to be computed for l as follows: identify the set $F = \{f_1(l), \ldots, f_q(l)\}$ of geohashes, such that

$$\forall f(l) \in F : |f(l)| = cp_{\min} \tag{1}$$

and

$$\forall f(l) \in F : f(l) \cap l \neq \emptyset \tag{2}$$

and $\forall p \in l$:

$$\exists f(l) \in F : f(l) \text{ is a prefix of geohash}(p) \tag{3}$$

whereas $f(l) \cap l$ denotes the intersection of all points that are contained in the rectangle represented by the geohash $f(l)$ and all points contained in l. The basic idea behind this approach is to raster the overlarge location element l into many smaller rectangular fragments whose size equals the size of the rectangles that are represented by Geohashes with the minimum allowable length $cp_{\min}$.

After the common prefix $cp(l)$ for l has been identified (or multiple $f_1(l), \ldots, f_q(l)$ respectively), the corresponding identifier is added to the Bloom filter. Given that a location area element might be covered by other location elements that exist in the Bloom filter, we perform an additional check before adding the element in order to reduce overall redundancy. Let $pre_i(cp(l))$, $1 \leq i < |cp(l)|$, be the prefix of $cp(l)$ with length i. To decide whether or not to add $cp(l)$, check for every prefix $pre_i(cp(l))$, $1 \leq i < |cp(l)|$ of $cp(l)$, whether $isCovered(pre_i(cp(l)))$ returns *true* or *false*. If all checks for the prefixes return *false* then add $cp(l)$ to the Bloom filter, otherwise discard it. Addition is achieved by applying the k independent hash functions onto the Geohash and by setting the resulting bit positions in V to 1.

Choosing the right values for $cp_{\max}$ and $cp_{\min}$ depends on the specific application scenario, but for obvious reasons very low values for $cp_{\min}$, e.g., 1 to 3, might quickly lead to many false positives when elements are tested for membership.

2.2 Testing for Coverage

Once location elements have been added to the Geocookie, the purpose of the *isCovered(l)* method is to check, whether a given location element l is covered by any of the elements already stored in the Geocookie.

In the first step a common prefix $cp(l)$ is computed the same way as before by taking into account the minimum bounding rectangle l_{mbr}, $cp_{\min}$ and $cp_{\max}$. In case the resulting Geohash is too short, multiple longer Geohashes $f_1, \ldots, f_m$ are computed as before.

In the second step classical Bloom filter checking is performed on each Geohash. Therefore the k hash functions are applied on each Geohash. The resulting values of each of the k hash functions point to positions in the bit vector V. If at least one of these bit positions in V is set to 0, then the Geohash element is definitely not contained in the Geocookie. In case all of the resulting k bits in V are set to 1, then the Geohash element is *probably* contained in the Geocookie. The probability of false positives depends on the length of V and the number k of hash functions, and is discussed below.

However, testing only for the Geohash of l will not reveal any coverage of existing (larger) regions. In case the initial check returns *false* (i.e., the Geocookie does not contain the Geohash value), additional checks need to be performed. Again, all prefixes $pre_i(cp(l))$ are tested too and only if none of these checks return *true* the location is identified as a non-member of the Geocookie.

2.3 False Positives and Location Coverage

The probability P_{fp} for a false positive when testing an element in a Bloom filter is

$$P_{fp} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{4}$$

where m is the number of bits in V, k is the number of independent hash functions and n the number of locations already stored in the Bloom filter [9]. For example, if a 25 kB bit array (200.000 Bits) is already filled with 10.000 elements using 10 hash functions, the false positive rate will be 0.00009. Recent work [4] suggests that this classic formula is wrong for small values of m, but the error can be neglected if m is large enough (> 1000).

For fixed values of m and n the optimum value for k, with respect to a minimum false positive rate, is $k = \ln 2 \frac{m}{n}$ [7].

Some of the location elements in the Geocookie might lead to multiple Geohashes because they exceed the minimum allowable prefix length and need to be fragmented. Thus the number of location items l may be lower than the actual number n of items that are added to the Bloom filter. Therefore the proportion f of location fragmentation needs to be considered, which is defined as the ratio between the number of Geohashes and the number of distinct location elements.

Thus the false positive rate for the Geocookie becomes

$$P_{fp} = \left(1 - \left(1 - \frac{1}{m}\right)^{knf}\right)^{k} \tag{5}$$

Furthermore, the *coverage ratio* $c \geq 1$ of a location element and its resulting Geohash bounding boxes is employed, it is defined as the fraction between the area covered by the Geohash approximation and the area covered by the location. The value of c for a location cannot be deduced once the location has been added to the Bloom filter. It is therefore important to calculate it during the addition, and utilize an increased Geohash raster resolution in case the desired coverage ratio cannot be met with the current prefix length.

Figure 1 shows an example for varying bit vector lengths and the resulting false positive rates for optimal k and $f = 1$.

Figure 2 shows the same scenario, but for $f = 2$, which equals two fragments per location element.

By keeping track of the number of previously stored elements, it is possible to assess the current statistical false positive rate for every Geocookie instance. Once it reaches a certain threshold, e.g., 0.001, it can be declared as full, thus inhibiting more additions. For some application scenarios it could be reasonable to focus on a single resolution in terms of the geo raster approximation. In that case $cp_{\min}$ will be equal to $cp_{\max}$, which will remove the necessity to perform additional checks on different prefixes.

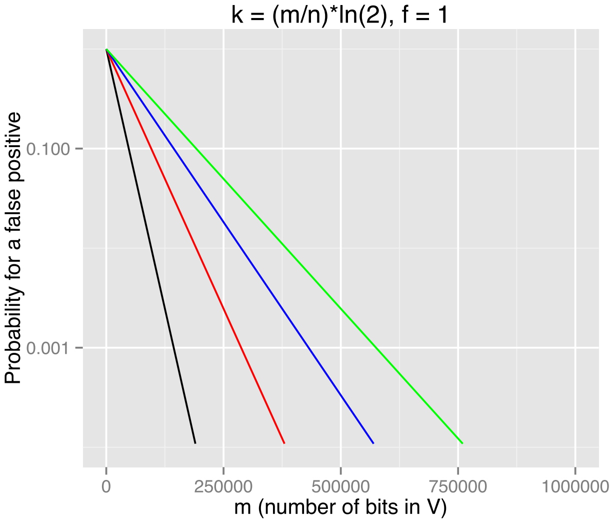

Fig. 1 False positive rates for different numbers of $|L|$: 10,000 (black), 20,000 (red), 30,000 (blue), 40,000 (green).

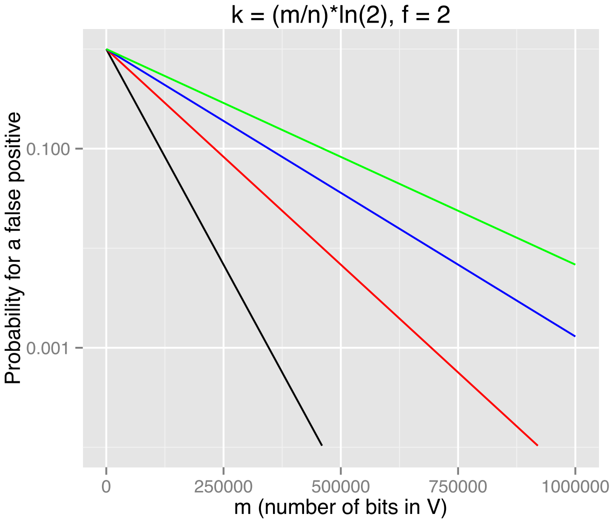

Fig. 2 False positive rates for different numbers of $|L|$: 10,000 (black), 20,000 (red), 30,000 (blue), 40,000 (green).

While the false positive rate of a Geocookie is affected by both the size of V and the number k of hash functions, the complexity of adding a location element or testing for coverage is $O(k)$.

The question which hash functions to use is the same as for any Bloom filter. Most importantly the hash functions need to be independent. In addition they should be as fast as possible with a uniform distribution. Good candidates are MurmurHashes, mainly because of their speed. But also SHA hashes can be applied, e.g., by taking fixed-length snippets of SHA-512 as values for $h_i(x)$.

2.4 Related Work

There exist several extensions to the classical Bloom filter. An extensive overview on network applications is given by Broder and Mitzenmacher [3]. Counting Bloom filters have been introduced by Fan et al. [6], they allow to delete elements from the filter by applying counters for each individual bit in the filter. Compressed Bloom filters [8] reduce the size of the data that is actually transmitted over the network by using larger but sparser bit vectors. Spectral Bloom filters [5] allows to filter elements based on their multiplicities in a multiset. However, existing approaches do not apply to geographical coordinates and they do not provide the means to test for cover of location elements. Geohashing on its own is widely used today e.g., in spatial databases and web map services for efficient storage and indexing. According to our present knowledge the proposed Geocookie approach is the first of its kind to combine location hashing with Bloom filters.

3. Geocookies for Location Data Processing

The general data structure of a Geocookie can be utilized in various different ways. In the following we classify different use cases and discuss their adaptability.

3.1 Types of Geocookies

Browser Geocookies can be used to append location sets to usual HTTP requests. Therefore the bit vector V is either send in the HTTP header or as an HTTP POST parameter. That way a client (browser) can send the Geocookie on a per-request basis to servers, where the location set acts as a filter on the returned response elements. The embedding of Geocookies into HTTP is discussed below in Section 3.4.

Local Geocookies can be applied both locally in mobile devices' browser storage and native applications' contexts, or also on the server side. The advantage of local geocookies is that they provide very fast means to assess whether location data, such as newly received GPS coordinates, are relevant in the current application context. On server-sided applications local Geocookies facilitate location-dependent event processing. For example, a Geocookie may present a set of certain cities or regions. Incoming location events can then be matched against the Geocookie and in case the event location is covered can trigger the according event. Essentially such a procedure is similar to other means such as basic Bloom filters, hash maps or bit maps, except for validating the coverage of location objects. If an application processes solely 2D point coordinates, then the Geocookie will not add significant performance, except for space savings when all coordinates are coarsened by limiting the length of Geohashes – which basically trades accuracy for performance. But in case the application has to deal with areas, polygons and coverage of objects, then the combination of Bloom filters and Geohashes is a convenient approach.

Hierarchical Geocookies provide a facility to separate multiple layers of location granularity in an application context. Therefore several Geocookies are maintained, which are configured with increasing values for $cp_{\max}$. The coarser ones will require less space and can be used to reveal rough information about contained locations. In parallel, more fine-grained Geocookies are created, possibly focussed on certain Geohash prefixes.

Public Geocookies represent a fixed set of locations that are not specific to a certain person or trajectory, but rather represent a group of locations in popular queries. For example, a public Geocookie may represent all pedestrian zones in a specific city or all points that are within a 1 kilometer radius of a train station. Assuming that no elements are added after its creation, the parameters for m and k can be fine tuned in order to minimize the size of V and achieve an acceptable false positive rate.

Concealed Geocookies contain either additional artificial location elements or are configured with a high false positive rate, which both disguises the actual set of locations of interest. It can be applied e.g., as a browser Geocookie and will hide the genuine elements from a server, thus making it possible to cloak e.g., trajectories or queries around visited places. That way concealed Geocookies also warrant plausible deniability in terms of queried locations, which increases the location privacy in some application scenarios.

3.2 General Privacy Considerations

Location privacy approaches can be grouped generally into three categories: privacy policies, data abstraction, and identifier abstraction. Privacy policies let users specify how location data about them should be processed. Data abstraction is achieved by artificially obfuscating location data, i.e., reducing the spatial and/or temporal accuracy, whereas identifier abstraction utilize pseudonyms that are linked to location data. In addition, there exists concepts such as k-anonymity, which describes the property of a target of being indistinguishable from at least k other targets.

The privacy properties of a geocookie belong to the class of data abstraction. However, it induces a different kind of privacy that is not directly related to k-anonymity, but to plausible deniability. By passing on a geocookie and making a statement such as "These are the places I've been to in the last year, please compile some travel recommendations for me," the sending party can plausibly deny that a certain location element is contained in the geocookie. The receiving party could only learn about the actual false positive rate P_{fp} (and thus about the probability that the denial is false), if and only if it knows the number n of existing elements in the geocookie. But given that n is only known by the sender, plausible deniability is guaranteed.

3.3 Distance-based Testing

So far, we described a combination of Bloom filters and Geohashes that allow to examine whether a location element is covered by any element in a location set. With a small modification it is possible to also allow a distance parameter d, so that the function *isCovered(l)* not only returns true if l is covered by an elements in L, but also if an element in L is apart from l not more than d. Therefore we need to recursively lookup the eight neighboring Geohash cells of $cp(l)$ and perform a regular Bloom filter test on their Geohashes.

3.4 HTTP Embedding

Geocookies can be embedded in HTTP request either as ordinary browser cookies or as POST parameters. Current Web browsers limit the size of header cookies to 4,096 Bytes, which directly limits the size of V. **Figure 3** gives an overview on varying numbers of $|L|$ and the resulting false positive rates for $m = 32{,}000$. A second scenario for $m = 64{,}000$ is depicted in **Fig. 4**.

Depending on the application requirements, such a size can already serve a couple of thousands of location elements. In case more elements (or a lower false positive rate respectively) are re-

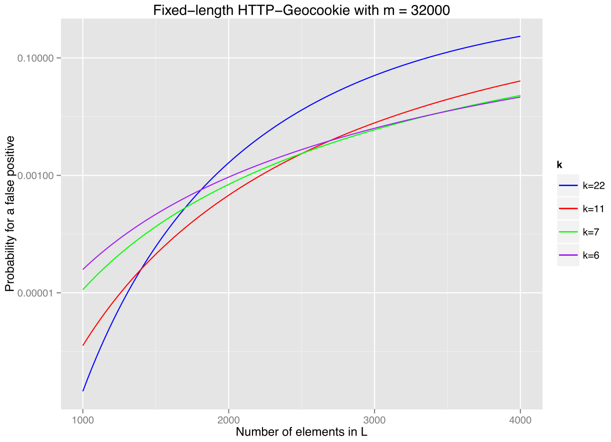

Fig. 3 False positive rates for a fixed-length HTTP-Geocookie with m = 32,000 for a varying number of elements in L. The four different values for k are the optimal values for $n = 1{,}000, 2{,}000, 3{,}000$ and 4,000 respectively. For example, the red curve ($k = 11$) is the optimal solution for 2,000 elements.

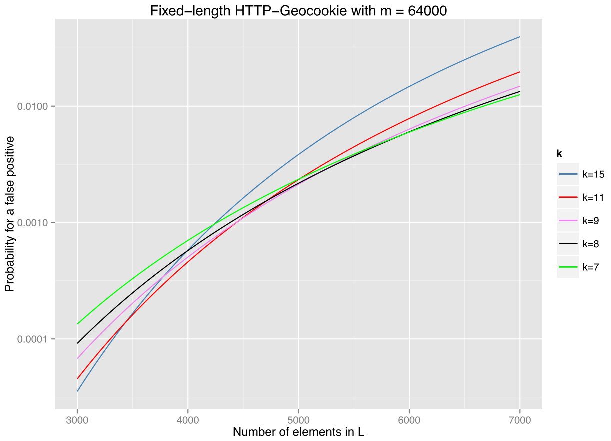

Fig. 4 False positive rates for a fixed-length HTTP-Geocookie with m = 64,000 for a varying number of elements in L. The five different values for k are the optimal values for $n = 3{,}000, 4{,}000, 5{,}000, 6{,}000$ and 7,000 respectively. For example, the red curve ($k = 11$) is the optimal solution for 4,000 elements.

quired, Geocookies can also be saved in the local Browser storage. The exact maximum allowable space varies from browser to browser, but currently is in the order of magnitude of a few MB per domain. At the same time these larger Geocookies can be transmitted as POST parameters, their size is not limited by the sender but only by the receiving application.

If Geocookies are exchanged between two parties it is required that not only the bit vector, but also the numbers for $|L|$ and k, as well as the exact hash functions and the values for $cp_{\min}$ and $cp_{\max}$ are named. So far they can be appended as additional HTTP header fields. The design of a standardized protocol for Geocookie exchange is outside the scope of this paper and object of future work.

4. Discussion

The presented approach aims at application scenarios that benefit from fast coverage checks and which at the same time are tolerant towards a certain false-positive rate. This compromise cannot be made globally, but rather needs to be adjusted to the applications' needs. One fundamental decision is the determination of the minimum and maximum length for the Geohash prefixes. Some levels, e.g., sub-meter accuracy, will not add any value to most applications. At the same time the difference between the minimum and maximum prefix length defines how coarse-grained large location elements will be mapped.

In case the application is less focused on varying polygons, but more on the geographical distribution of homogeneous location elements, then it can perfectly make sense to set $cp_{\min} = cp_{\max}$. This can significantly reduce the overall number of objects in the Bloom filter, but only if there are no outliers, i.e., large polygons that will cause a high number of fragments to be inserted.

An open issue is the removal of previously inserted location elements. There exist approaches to remove elements from a classic Bloom filter by basically counting the number of times a certain bit has been set to 1 and decrementing the corresponding numbers upon removal of an element. However, these approaches apply to single-identifier items only. For the Geocookie, one location element can cause multiple Geohashes to be added to the Bloom filter. In that case the relation between these rectangular regions (Geohashes) cannot be recovered later on. And given that different location elements may spatially overlap, their resulting Geohash fragments cannot be removed without possibly destroying another element.

A second challenge for future work is to keep track of the time at which a location element has been added. This would require to extend the filter by time stamps and could support applications that rely on time-based ordering of locations.

5. Conclusions

In this paper we have proposed a compact representation of geographic location sets, which is based on the well-known concepts of Bloom filters and Geohashes. Compared to existing approaches the advantage is that polygonal location elements can be maintained and tested for coverage. The granularity of stored location elements can be adjusted by defining upper and lower bounds for the Geohashes, and the false positive rate for coverage checks can be computed and adjusted in advanced. Geocookies support various use cases such as passing location sets in browser requests, speeding up local lookups on location elements or sharing fixed location sets that act as filters for location-based queries.

References

[1] RFC 3548 - The Base16, Base32, and Base64 Data Encodings (2003).

[2] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors, *Comm. ACM*, Vol.13, No.7, pp.422–426 (July 1970).

[3] Broder, A. and Mitzenmacher, M.: Network applications of bloom filters: A survey, *Internet Mathematics*, Vol.1, No.4, pp.485–509 (2004).

[4] Christensen, K., Roginsky, A. and Jimeno, M.: A new analysis of the false positive rate of a bloom filter, *Inf. Process. Lett.*, Vol.110, No.21, pp.944–949 (Oct. 2010).

[5] Cohen, S. and Matias, Y.: Spectral bloom filters, *Proc. 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03*, pp.241–252, ACM (2003).

[6] Fan, L., Cao, P., Almeida, J. and Broder, A.Z.: Summary cache: A scalable wide-area web cache sharing protocol, *IEEE/ACM Trans. Netw.*, Vol.8, No.3, pp.281–293 (June 2000).

[7] Kirsch, A. and Mitzenmacher, M.: Less hashing, same performance: Building a better bloom filter, *Random Structures & Algorithms*, Vol.33, No.2, pp.187–218 (2008).

[8] Mitzenmacher, M.: Compressed bloom filters, *IEEE/ACM Trans. Netw.*, Vol.10, No.5, pp.604–612 (Oct. 2002).

[9] Mullin, J.K.: A second look at bloom filters, *Comm. ACM*, Vol.26, No.8, pp.570–571 (Aug. 1983).

[10] Niemeyer, G.: Geohash, available from ⟨http://geohash.org⟩.

Peter Ruppel is a senior researcher and lecturer at Telekom Innovation Laboratories, a private-public partnership of Deutsche Telekom AG and Technische Universität Berlin. He holds a doctoral degree in computer science from LMU München. His current research interests include Location-based Services, Mobile Applications, Location Data Analysis and Positioning Systems.

Axel Küpper is a professor for Service-centric Networking at Telekom Innovation Laboratories, which is a private-public partnership of Deutsche Telekom AG and Technische Universität Berlin, and a co-founder of Bitplaces GmbH, a Berlin-based start-up offering solutions for the next generation of Location-based Services. Prior to his current tenure, he researched and lectured at RWTH Aachen and LMU München, where he received his post-doctoral lecture qualification while working as an assistant professor and research assistant at the institute for computer science. His current research topics include Location-based and Context-aware Services, Cloud Computing, Semantic Web, Online Social Networks, and Business Process Management.