Reducing the Cloud Cost of Mobile Reverse-Geocoding

Thomas Phan, Albert Baek, Zheng Guo
Samsung Research America - Silicon Valley
San Jose, CA
{thomas.phan, albert.b, z.guo}@samsung.com

ABSTRACT

Reverse-geocoding performs an important function for many mobile applications, converting geographic latitude & longitude coordinates into real-world physical locations. While the resulting reverse-geocoded locations can be invaluable for many mobile apps, the process comes at a high cost: either battery power must be expended to invoke a cloud server, or local storage must be used to keep detailed cartographic data to run the process on the phone. In our work we reduce these costs by exploiting the user's geolocality and perform on-smartphone caching of reverse-geocoded locations obtained from calls to the cloud. To that end, we configured three different geospatial region-definition schemes (convex hulls, radial boundaries, and our own cartographic sparse hashes), implemented Android software to perform this caching, and explored cache propagation via preemptive pushing. We evaluated our system using a data set of 1.1 million geotagged photos taken with smartphones and show that our caching: (1) reduces the number of cloud server calls by over 70% for neighborhood granularity and by over 85% for city granularity; and (2) consumes less than 1MB of hash-encoded data even for a complete precomputation of the San Francisco Bay Area.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Communications Applications

Keywords

smartphone; cloud; reverse-geocoding; caching; GIS

1. INTRODUCTION

Mobile applications can leverage cloud computing to access services that are not bounded by the storage, computation, and battery limits of smartphones. One such service is reverse-geocoding, the process of converting a geographic coordinate, commonly expressed as a latitude & longitude pair, into a real-world physical location. A smartphone can obtain its geocoordinate either through GPS or cellular/Wi-Fi trilateration, and then reverse-geocoding will perform the additional step of providing a human-readable name for that geocoordinate, often at multiple resolutions. For example, the geocoordinate of 37.79522 latitude & -122.40296 longitude, both in units of decimal degrees, can be reverse-geocoded to the physical location of "600 Montgomery Street, Financial District, San Francisco, California, United States."

This type of named location metadata is invaluable for mobile applications that provide services based on the user's real-world position. For example, on-phone photo albums can group geotagged photos by city [9], assistive apps can vocalize to a blind user that he is moving through specific neighborhoods [2], and digital guidebooks can inform a tourist that he is standing near historic landmarks [4].

While real-world locations can be a great benefit, performing reverse-geocoding comes at high cost on smartphone platforms, where applications must send a geocoordinate to a cloud server to get the reverse-geocoded result. On some cellular networks, making the invocation can consume much battery power. For example, using a Samsung Galaxy S IV smartphone (released in April 2013) on AT&T's 4G LTE network outdoors in the San Francisco Bay Area, we measured a reverse-geocoding call to consume 710 mW, while using GPS to get a geocoordinate – a battery-hungry process in itself – consumed less at 391 mW. As a result, for scenarios where knowing the physical location many times during the day would be useful, high battery usage makes repeated reverse-geocoding server calls prohibitively expensive.

An alternative would be to run the same reverse-geocoding process on the smartphone, an approach that requires sufficient cartographic data to be stored in order to perform accurate lookup [21]. Proprietary commercial applications such as those from Garmin and TomTom allow the user to purchase and install maps for offline usage, consuming 100s of MBs to several GBs of storage. We seek a generalized, non-proprietary, OS-agnostic, and low-storage solution.

In this paper we propose an approach for reverse-geocoding down to the resolution of neighborhoods and cities/towns that reduces both the battery cost of invoking cloud servers and the local storage and monetary costs of using commercial mapping apps. First, we exploit user geolocality by *caching* reverse-geocoded locations obtained from server calls. The key challenge is then to define and manage reverse-geocoded geospatial regions on the phone in a manner that enables high accurate hit rates. In our work we evaluated three schemes to achieve that purpose: convex hulls, radial bounds, and cartographic hashing. Second, our system can

MCS'14, June 16, 2014, Bretton Woods, New Hampshire, USA.

http://dx.doi.org/10.1145/2609908.2609947.

propagate this data by *preemptively filling the user's cache* with precomputed regions that are pushed to phones.

We evaluated our system in the context of mobile photography, where reverse-geocoding can help organize photos. We used Flickr.com's public API and downloaded metadata of over 1.1 million geotagged and timestamped photos that were taken specifically with smartphones and conducted experimentation to determine the accurate cache hit rate, a measure that estimates the joint probability of hits in the cache that are also ground-truth correct. Our results show that for the photography-oriented user traces obtained, our system reduces the number of cloud server calls by over 70% for neighborhood granularity and by over 85% for city granularity. Finally, we show that the scheme offers substantial smartphone local storage as well, with our system occupying less than 1 MB of storage using our hash encoding even for a complete precomputation of the San Francisco Bay Area.

The rest of this paper is organized in the following manner. In Section 2 we describe related work, and in Section 3 we describe the reverse-geocoding process. We discuss our novel caching technique in Section 4 and show experimental results in Section 5. We conclude the paper in Section 6.

2. RELATED WORK

Our work is motivated by the need to reduce the battery and on-device storage costs of performing repeated reverse-geocoding for location-based applications, a problem we explored earlier with a data set of 5000 geolocation points [20]. In this paper we provide caching and propagation of locations acquired via cloud-assisted lookups on a data set of over 1.1M photos. To the best of our knowledge, ours is the first work that addresses this problem and offers a general and open solution.

Reducing power used by location-based applications on constrained mobile devices is an important topic that has been discussed in other work. GPS is the usual culprit, and other researchers have looked for solutions, such as inferring user behavior from lower-power sensors like the accelerometer to intelligently trigger GPS (e.g. [5, 23]), observing user speed (e.g. [13, 7]), and offloading functionality to servers (e.g. [14]). Our work is complementary; instead of addressing GPS, we identify a higher battery-cost operation, reverse-geocoding, that can be called repeatedly by applications, and we apply our effort at reducing this consumption.

Other work has looked at data-mining interesting locations obtained from GPS traces (e.g. [22, 3]) using offline desktop computation. Here, we use the user's geolocality to perform caching performed entirely on the smartphone.

We look to partition a metric space into regions, a goal similar to machine learning classifiers such as SVM or Decision Trees. Our feature space is very small (only two dimensions), reducing the need for a sophisticated classifier on a battery- and memory-constrained phone.

3. REVERSE-GEOCODING

In this section we describe reverse-geocoding, give a brief overview of the relevant terminology and applications, and explain why current solutions face many challenges.

3.1 Applications

Geocoding converts a human-readable query for a street address or landmark into a latitude & longitude coordinate. The process involves canonicalization of the query, searching through an index of ground-truth locations, and producing the associated geocoordinate. In the absence of an exact match to any known location, interpolation is performed to estimate the geocoordinate.

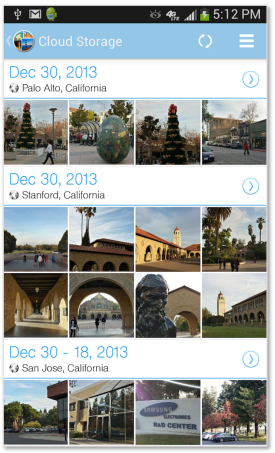

Figure 1: Our album groups photos by place and time.

Reverse-geocoding naturally performs the opposite function. Given a geocoordinate as input, its output is a named location at multiple resolutions. For example, the result can describe a location by its name, street address, neighborhood, county, city, state, and country.

In this paper we focus only on reverse-geocoding, which we distinguish from mapping, the process of projecting a geocoordinate onto a graphical map. We further concentrate on reverse-geocoding to neighborhoods and cities/towns (although we will extend our work to street-granularity in the future) because it already enables many application scenarios that do not have a need for mapping or routing at a street level. Instead, such applications rely only on named locations and thus may make repeated reverse-geocoding requests throughout the day, such as:

- Audible guidance for the blind: Assistive applications can vocalize human-understandable location names for visually-impaired users living in a city (e.g. [2]).
- Tourism: Visitors who are within the vicinity of a landmark can be informed what they are near (e.g. [4]).
- Life-logging: Applications can perform continuous life-logging into a diary so users can review what neighborhoods and towns they have been through (e.g. [8]).

To make our work concrete, we ground our discussion for the remainder of this paper in the context of mobile smartphone photography. Our own mobile app, shown in Figure 1, is in this domain. Mobile smartphone photography is important because in the last several years smartphone camera usage has increased significantly, with estimates on the order of millions of photos being taken by smartphones on a daily basis, out-pacing traditional point-and-shoot digital cameras [6]. Further, we can take advantage of data from photo-sharing websites, such as Flickr, in order to get geotagged and timestamped photos for testing our algorithms.

Grouping photos into disjoint albums is a popular use case for smartphone photography, where the groupings are based on the city or neighborhood where the user took the photo [9, 19]. The location can be determined from geocoordinates usually embedded into photo metadata, such as

JPEG EXIF headers. Our mobile app groups photos by locations acquired through reverse-geocoding lookups.

3.2 On-device reverse-geocoding challenges

To perform reverse-geocoding effectively, ground-truth spatial data must be collected accurately and exhaustively to describe physical locations. Spatial indexes such as R-Trees [11, 21] are then built to enable logarithmic search time. Companies providing mapping data, such as Navteq (owned by Nokia), Tele Atlas (owned by TomTom), and Google, invest heavily in street-level mapping, and because mapping is considered a killer feature for modern smartphones, collected data is kept proprietary for competitive advantage.

The nature of the data and the spatial index provides insight into why offline reverse-geocoding is problematic. First, and obviously, accuracy is impacted by the amount of data stored. If a location is missing from the index, then a search must settle for either nearby objects or a coarser-grained description. Further, unlike geocoding where the resulting answer is a geocoordinate with continuous numbers, missing real-world locations cannot be as easily interpolated. As a result, providing accurate reverse-geocoding results improves with more data. Since phones can be space-constrained, keeping a finely-detailed index is challenging.

We note that at the time of this writing, some commercial (non-free) offline mapping applications are available for iOS, Android, and Windows Phone, including those from Nokia, Garmin, and TomTom. The offline map data can span from 100s of MB to several GB depending on the region. It is important to note that while some mapping applications listed in the app stores may state that the software is free-of-cost and occupies space on the order of 25 MB, the user must purchase and download the map data separately. This approach does not exploit geolocality of a user who very rarely leaves his metropolitan region, and as a result may unnecessarily consume phone storage. Also, some applications sell the map data as a subscription, incurring ongoing charges. These storage and monetary costs inform our decision to find a non-proprietary, OS-agnostic, and low-storage solution.

Finally, updating the index may produce inefficient trees. Because cartographic data may change often due to urban construction, the index structure must change as well, leading to poorer performance compared to indexes built from scratch. For this reason and others, updates to offline mapping applications almost always require that the user download entirely new data, which incurs a non-negligible delay.

3.3 Cloud-assisted reverse-geocoding challenges

Given the space requirements for detailed cartographic data and the need to perform updating of the spatial indexes, mapping and reverse-geocoding services are most commonly accessed through online cloud services that can leverage the availability of TB-sized data stores (for example, Earth data provided by the open-source OpenStreetMap is about 330 GB in raw, unindexed form [15, 17]). On Android and iOS, there exist native API for performing reverse-geocoding by calling cloud servers, and in the absence of such dedicated API, programmers can still perform reverse-geocoding through RESTful HTTP calls to Google[10], OpenStreetMap Nominatim [16], and other services.

The resulting problem is that these online calls are costly in terms of battery power. For example, in our photography context, suppose the user is out taking photos, such as on a weekend or vacation trip, and wants to view his grouped albums several times during the day; invoking a server-side process for reverse-geocoding lookup of one or a few photos at a time would then incur increased power consumption. We quantified this expenditure using a Samsung Galaxy S IV smartphone, where we measured reverse-geocoding invocations to Nominatim over 4G LTE to consume, on average, twice the power of GPS. While battery consumption due to wireless data usage is an issue for any cloud-connected mobile app, it is especially problematic for the types of applications we consider since by their nature, the user is often out of the range of Wi-Fi and still needs to make repeated reverse-geocoding requests during the day.

In addition to battery use, other potential problems are:

- Calling the server incurs a wireless hop delay and a server processing delay.
- Using cellular data will incur wireless provider monetary charges, depending on the user's data plan.
- The reverse-geocoding service provider may enforce a quota limit that repeated requests may exceed.
- Finally, from the viewpoint of the reverse-geocoding service provider, fulfilling requests from potentially thousands or millions of mobile app users each day can be burdensome in terms of CPU load.

4. CACHING AND PROPAGATING REVERSE-GEOCODED LOCATIONS

4.1 Overview

In the previous section, we discussed mobile apps that leverage named physical locations and described the current limitations of performing repeated lookups: on-device, offline commercial mapping apps consume storage and monetary expenses, while server invocation (the most common practice) consumes battery due to data transmission.

Our approach looks to exploit geolocality in order to minimize both on-smartphone storage and server-assisted calls. For many application domains, user mobility is often limited in reality to intra-metropolitan regions. For example, from our 1.1M photo metadata set, we observe that users often take hundreds of photos (presumably at a special event, on vacation, or on a weekend trip), all within a relatively small region spanning a few neighborhoods.

We take advantage of geolocality in two specific ways:

- We perform caching of geospatial regions whose ground-truth location names are obtained from reverse-geocoding servers (running OpenStreetMap Nominatim). We implemented three schemes and ran our caching on Android. Our system is described in subsection 4.2.
- We also explore the opposite end of the communication versus storage tradeoff: instead of lazily storing visited regions, we can precompute hashed boundaries and preemptively push them to the phone. This preemptive propagation is described in subsection 4.3.

4.2 Caching

4.2.1 Cache operation

In our Android implementation, the cache is maintained mostly in memory with a SQLite backing store. The cache

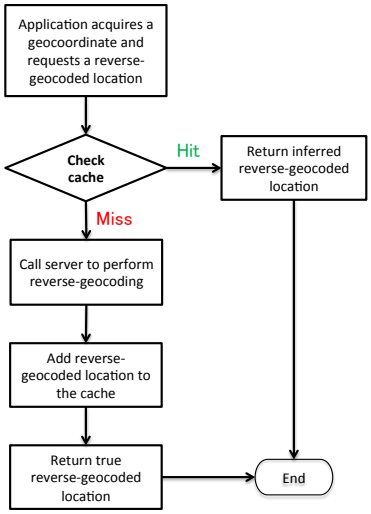

Figure 2: Caching algorithm decision flow.

operates using the logical flow in Figure 2. After an app acquires a geocoordinate (e.g. via GPS), it can call our component to get the location name. Our system then checks to see if the geocoordinate is in a cached region that has already been ground-truth labelled with a name. If there is a hit, then the location name is returned to the application, but note that this location is *inferred* from previously-cached data, where the accuracy of the inference is affected by the region definition scheme, as discussed below.

If there is instead a cache miss, then the system makes a call to a reverse-geocoding server to get a ground-truth location name. In our work we use OpenStreetMap data and the Nominatim reverse-geocoding service that we installed on our own Linux servers. After reverse-geocoding completes, the geocoordinate and its labelled name are added to the cache. A separate table maintains the name strings; with 1000 unique location names taking on average 40 2-byte characters, the lookup table will consume 80 kB. The name is then finally returned to the application.

Our caching system is characterized by the following three asymptotic probabilities:

$$Pr(hit) = \frac{\text{Requests answered with cached geolocation}}{\text{Requests made to cache}} \quad (1)$$

$$Pr(correct|hit) = \frac{\text{Requests answered with correct geolocation}}{\text{Requests answered with cached geolocation}} \quad (2)$$

$$Pr(correct, hit) = Pr(correct|hit) \times Pr(hit) \quad (3)$$

Equation (1) is the cache hit rate, a traditional measurement for caching strategies. Here, it is the proportion of reverse-geocoding requests that can be answered (correctly or incorrectly) by the cache without having to call a server.

In our case, although a geocoordinate may be in a cached region, that matching region may be labelled with a different name than where the geocoordinate actually is. For example, suppose the neighborhood of Chelsea, New York City, is cached but with a region defined to be the Earth. A subsequent request for a California geocoordinate would result in a cache hit, but the inferred Chelsea location would be inaccurate. We use Equation (2) to define this accuracy.

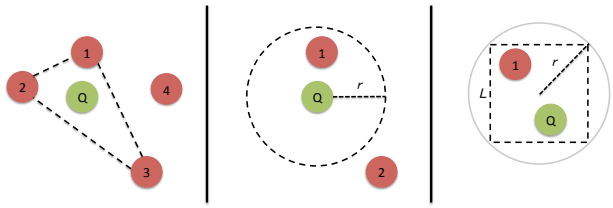

Figure 3: The three geolocation caching schemes we evaluate: convex hull (left); radial bounds (center); and Cartographic Sparse Hashing (right).

Inaccuracy can be assuaged by two means. First, the user or developer can set the cached region to be finer, but this adjustment may decrease both the cache hit rate and the accurate cache hit rate, discussed below. Second, if a user can manually identify a mislabeled location, then the system can take the same steps as it would for a cache miss, retrieving a ground-truth name from the cloud, re-labelling the cached geocoordinate, and returning the name to the application. Importantly, we *always* assume the worst case where a server call is made for all inaccurately inferred results.

Equation (3) is the **accurate cache hit rate**, the joint probability that there is a cache hit *and* an accurate result. This metric is extremely informative because it describes the proportion of all reverse-geocoding requests that can be accurately resolved in the cache without having to call a server. If we assume that in the absence of the cache that N reverse-geocoding server calls need to be performed, then we can state that a caching system with an accurate cache hit rate P will invoke the server $N \times (1.0 - P)$ times.

The key challenge to performing caching of location names is defining and managing the cached regions in order to enable a high accurate cache hit rate. We next discuss three schemes that implement this region management.

4.2.2 *Convex Hull*

Consider an application that asks for reverse-geocoding for three geocoordinates, shown as points 1, 2, and 3 on the left-hand side of Figure 3. All three produce a cache miss, resulting in a cloud-assisted reverse-geocoding call. If all three points share the same location, then we can form a bounding convex hull around them and assign the hull that location name. Suppose the app then requests a name for the query point Q that lies within the hull; the query is thus a cache hit, with the point being labelled with hull's inferred location. Note that since Q is inside the hull, it is not added to the cache. Further, suppose that the application acquires another geocoordinate, shown as point 4. Since this point lies on the outside of the hull, it is a cache miss and thus requires a cloud-assisted lookup. Now, if this point happens to have the same location name as the other points 1, 2, and 3, then the convex hull can be extended to cover point 4.

To efficiently find covering hulls, we place hull points in an R-Tree index. For H hulls with M points on average, this approach runs in $O(lg(HM) + M)$. An advantage is that potentially few points are needed to define a large region.

4.2.3 *Radial Bounds*

Our next region definition and management scheme, shown in the center of Figure 3, is based on a query point, a fixed ra-

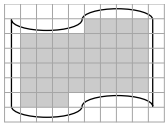

Figure 4: Preemptive precomputed hashes.

dius r, and the resulting bounding circle that can be formed around the point. Suppose initially the application has requested points 1 and 2, found them to be cache misses, and added them and their ground-truth locations into the cache. Now, for a new query geocoordinate at query point Q, the cache is searched to find the nearest point within distance r to Q. We use an R-tree to store the points, and to compute distances, we use a Haversine estimation for two geocoordinates upon a spherical surface (where the sphere has been modeled with the dimensions of Earth).

In this example, the result is point 1, and so point 1's location name is assigned to the query point Q. Note that Q is not added to the cache since its city location was inferred, not obtained through a ground-truth reverse geolocation lookup. If there are N points kept in an index, then this nearest-neighbor query can be answered in $O(lg\ N)$ on average. The disadvantage is that in order to achieve a high accurate hit rate, many points must be kept in the cache.

4.2.4 Cartographic Sparse Hashing (CASH)

Our third scheme uses the implicit formation of square boundaries formed throughout the geocoordinate space, as shown on the right of Figure 3. Here, the query point Q is hashed to the same implicit square as point 1, which was already in the cache, and so Q is given the same location label. Note that Q is not added to the cache.

Each square is formed through Cartographic Sparse Hashing (CASH), our hash algorithm that takes as input (i) latitude and longitude as 64-bit floats and (ii) a resolution in meters. It then outputs a 64-bit integer, where the hashed components for the longitude and latitude end up in the high and low bits, respectively. (Due to space constraints, we defer detailed code for another paper.) This final hash key is then used in lieu of geocoordinates in the cache. To better compare this scheme with radial bounds, we define a square boundary based on a radius r that describes a circumscribed circle around a square whose side is $L = \sqrt{2} \cdot r$. We use this value of L as the resolution given to the hashing function.

The advantage of this scheme is that cache search is $O(1)$, but like the radial bounds scheme, it suffers from needing many cached points to achieve a high accurate hit rate.

4.3 Preemptive cache propagation through precomputed hashing

One of our goals was to reduce on-device storage by keeping a small set of cached regions. Here, we consider what advantages could be gained by relaxing this requirement by first offline precomputing regions for an entire metropolitan area and then pushing the results to pre-fill location caches. Such an approach could improve the hit rate $Pr(hit)$ while not adversely affecting the accuracy $Pr(correct|hit)$.

Our approach is illustrated in Figure 4. Consider an arbitrary geometry, shown by the dark outline, that represents a boundary. We subdivided the region into squares with $r =$ 500 m, took points every 100 meters per linear side, and performed ground-truth reverse-geocoding at each point. Homogeneous squares have all points with the same name and are thus labelled with that name; these are the grey squares in the figure. Heterogeneous squares occur on the borders.

We then packaged the homogeneous regions (comprising their hashed value using CASH and their geolocation name) and pushed them to phones to pre-fill the caches. Because heterogeneous regions are not included, geocoordinates falling into such regions would be a cache miss. For the entire San Francisco Bay Area covering 9043 km^2, this approach produced 32,458 homogeneous squares for city granularity. At 16 bytes per entry, including the name lookup table discussed earlier, *the pre-filled cache occupies 587 kBytes.*

Activity	Ave. power
GPS	391 mW
Reverse-geocoding, 4G LTE	710 mW
Reverse-geocoding, 4G HSPA	381 mW

Table 1: Measured power consumption. GPS was averaged over 300 seconds, while reverse-geocoding was averaged over 30 calls.

5. EXPERIMENTS

We implemented and tested the reverse-geocoded location caching system on Android smartphones. Lab members took photos and used our album app to group together photos by city and neighborhood. The algorithm worked as expected, but to fully evaluate it, we conducted offline experiments using the same Java codebase but on a much larger data set: 1.1M publicly-available smartphone photos from Flickr. This section describes the experimental results.

5.1 Power consumption

We first evaluate our fundamental assumption that repeated reverse-geocoding operations can be a significant battery drain. As mentioned in Section 3, smartphones commonly perform reverse-coding by calling a cloud server. Currently in the USA, commercial wireless ISPs such as AT&T and Verizon offer 3G and 4G, with connectivity over 4G LTE and 4G HSPA providing the highest bandwidth.

We thus looked to determine the power consumption of making a RESTful invocation to the public Nominatim server [16] to perform reverse-geocoding. In our tests we used a commodity Galaxy S IV smartphone and measured its power use outdoors with a Monsoon Solutions Power Monitor hardware power meter. We were very careful to deactivate irrelevant background processes, turn off the screen, and keep the CPU awake with an Android wakelock.

Our results are shown in Table 1. Using GPS to acquire a geocoordinate took about 391 mW, which includes idle CPU power. The power needed for reverse-geocoding over 4G LTE was almost twice as much at 710 mW, while the slower 4G HSPA consumed on the same order as GPS.

We can also estimate the consumed energy (in Watt-hours) following a model where power is integrated over time. Similar to [18], we found that our test phone exhibited a tail power state where the LTE radio remains consuming power even after a network invocation ends; as a result, each call kept LTE on for 10.12 seconds on average.

In our work, we found mobile photographers that took over 250 photos in one day, and for life-logging applications that take geolocations every 2 minutes, 576 reverse-geocoding calls would be needed. If we evaluate our power

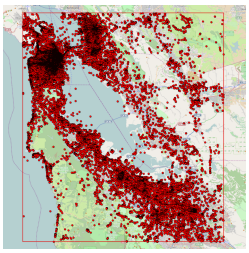

Figure 5: Some photos from the San Francisco Bay Area comprising 194K photos and 68 cities/towns.

model with 400 such calls per day, we estimate an energy expenditure of 0.79 Wh, or over 8% of our phone's 9.88 Wh battery. We measured our fully-charged phone to last between 12 and 18 hours on average with active use, so our estimation roughly equates to be between 1 and 1.4 hours of battery life. These observations suggest that having mobile apps make repeated calls to online servers for reverse-geocoding can cause a substantial drain, especially since each reverse-geocoding call is preceded by a GPS invocation.

5.2 Data set

To fully test our system, we focused on the context of mobile photography and the application-driven need for resolving photos to city and neighborhood locations in order to form photo album groupings while the user is out taking photos. We posit that this behavior may be representative of other location-based applications that need repeated reverse-geocoding during the day, such as those discussed in Section 3.1. We will explore other apps in future work.

Our data comprised over 1.1 million photos taken in the USA from Flickr.com. We queried for photos taken between September 1, 2009, and September 1, 2013, that were *specifically taken with smartphones* in order to obtain the closest trace possible to mobile photo-taking behavior.

Each retrieved photo was timestamped and geotagged with latitude & longitude coordinates. We assumed that the coordinates were produced by the phone's geolocation service when the photo was taken. We then obtained a ground-truth city and neighborhood for each photo by performing a reverse-geocoding lookup against OpenStreetMap data and the Nominatim service, both of which ran on a local server.

Attributes of this data were: 1,109,311 photos; 37,237 users; 34 states; 2062 cities/towns; and 3759 neighborhoods. We show plots for the San Francisco Bay Area in Figure 5.

On average, each Flickr user had 29.8 photos, but we note four things. First, Flickr albums are curated, so while a user may take many photos in a day, only some photos may be uploaded. Second, the distribution of photos per user is long-tailed where the head comprises users with thousands of photos, with an observed maximum of 5646. Third, while Flickr is popular, people are increasingly using Facebook, so Flickr may not be fully representative of mobile photography users . Finally, the 2062 cities/towns varied widely, including those with dense centers (e.g. Chicago, IL), sprawls (e.g. Los Angeles, CA), and low tourism (e.g. Oklahoma City, OK).

5.3 Schemes

We evaluated our system using the three region-management schemes from Section 4.2 and a fourth that serves as a baseline. The cache is always started completely cold (empty).

- **Convex Hull**: The convex hull containment scheme.
- **Radial**: The radial bounds scheme parameterized at 500, 1000, and 2000 meters.
- **CASH**: The cartographic sparse hashing scheme parameterized at 500, 1000, and 2000 meters.
- **Time**: This scheme serves as a baseline comparison with expiration times of 5 and 60 minutes. We assume that the caching system keeps exactly one previous geolocation name. If a request is made within the expiration time, then a cache hit occurs, returning the one geolocation name. If not, then it is a cache miss.

5.4 Case study for single user

We first evaluate our system in the context of a single user case study in order to gain insight at a microscopic scale. We will then evaluate the system at a macroscopic scale for all 37,237 users in the next section.

We selected a random user from our data set and found a subset of 519 photos that were taken over the 3-day span of August 12-14, 2011. We sorted the photos by ascending timestamp and iterated over them to request reverse-geocoded neighborhood and city/town locations from our caching system, simulating user behavior as if he were requesting reverse-geocoding lookup while taking his photos.

The first data row in Table 2 shows the observed relative frequency for each candidate geocoordinate's request achieving a hit in the cache using the granularity of neighborhood; this relative frequency approximates the underlying asymptotic probability $Pr(hit)$. Note that for both the Radial and CASH strategies, a larger radius intuitively produces more hits. Convex Hull produces a low hit rate for this user; when we traced him on a map, we found that he was moving along the perimeter of what would have been an enclosing hull.

The second row of Table 2 shows the relative frequency for accurately-inferred neighborhoods given that a hit occurred, approximating the asymptotic probability $Pr(correct|hit)$. Note for Radial and CASH, a small radius has higher accuracy because it is less likely to contain mixed geolocations.

The third row of Table 2 then shows the joint relative frequency of both a cache hit *and* an accurate inference at the neighborhood granularity, approximating the probability $Pr(correct, hit) = Pr(correct|hit) \cdot Pr(hit)$. Both Radial and CASH evaluate well here for this user, producing neighborhood accurate hits in excess of 80%. The Convex Hull scheme overall scores poorly because its low coverage of regions significantly reduce its accuracy for contained regions. Both Time schemes perform better than Convex Hull.

The last row of Table 2 shows the accurate cache hit rate at the granularity of city/town. As expected, the coarser regions of cities produces a higher accurate hit rate versus finer-grained neighborhoods due to improved accuracy.

We can explore the caching mechanism behavior in more detail by looking at the accurate cache hit rate as this user takes one photo after another. In Figure 6 we show the cumulative accurate cache hit rate for CASH parameterized at 1000m with city granularity. The rate is computed after each of the user's photos is taken, where the cache starts cold but quickly ramps up as the user takes more photos.

Figure 7 shows an estimate of the number of server calls at city and neighborhood granularity. For example, CASH at 1000 meters achieves an accurate cache hit rate of 0.802 at neighborhood granularity, as was shown in the third row

Probability	Convex Hull	Radial (2000m)	Radial (1000m)	Radial (500m)	CASH (2000m)	CASH (1000m)	CASH (500 m)	Time (60 min)	Time (5 min)
$Pr(hit)$, neighborhood	0.553	0.946	0.927	0.906	0.925	0.915	0.884	0.811	0.665
$Pr(correct\|hit)$, neighborhood	0.962	0.802	0.865	0.887	0.869	0.876	0.946	0.886	0.930
$Pr(correct, hit)$, neighborhood	0.532	0.759	0.802	0.803	0.803	0.802	0.836	0.719	0.618
$Pr(correct, hit)$, city	0.617	0.900	0.884	0.867	0.882	0.879	0.846	0.763	0.638

Table 2: Caching results for sample single user.

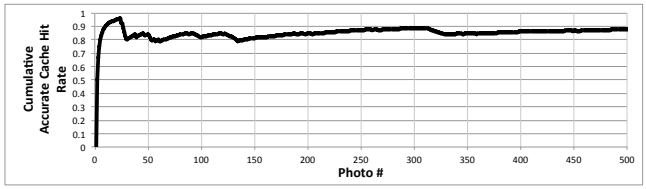

Figure 6: Cumulative accurate cache hit rate for one user, city granularity, using CASH (1000m) as photos are being taken. The cache starts cold.

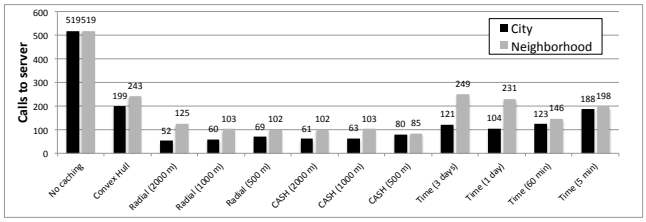

Figure 7: Number of calls to servers for one user.

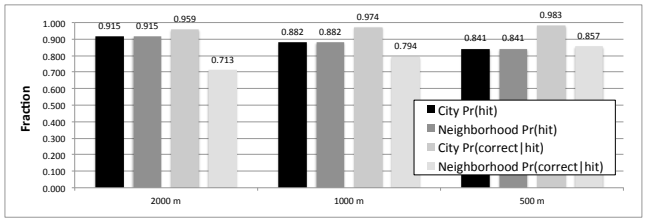

Figure 8: Hit rate & accuracy for CASH, varying radii.

of Table 2. *The result is that only 0.198 of the user's photos will result in a cloud invocation, or 103 out of the 519 photos.*

5.5 Evaluation for 37K users

We can apply insight gained from our previous single-user case study toward the entire 37,237-user data set. As we saw, the accurate cache hit rate $Pr(correct, hit)$ is a meaningful metric that characterizes how often the cache can fulfill an app's reverse-geocoding request without having to call the cloud. To evaluate our schemes with this metric over all the users, we conducted similar experiments.

In Table 3, we show the average accurate cache hit rate at neighborhood and city granularity, revealing two trends. First, Radial and CASH produce approximately the same results, and both are superior to Convex Hull and Time. Second, for both Radial and CASH, a finer radius produces a higher average accurate hit rate for neighborhood, but the opposite is true for city. (This behavior was also visible in the one-user study but is very evident over the entire data set.) The reason is that while both neighborhood and city produce the same hit rate reduction with decreasing radius, neighborhood's accuracy increases substantially more

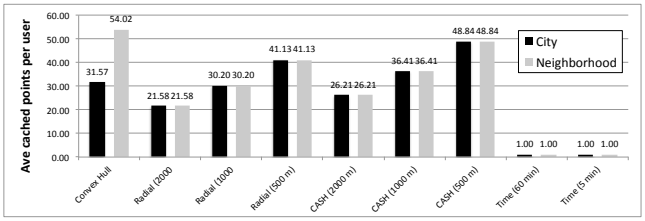

Figure 9: Average number of cached points per user across all users.

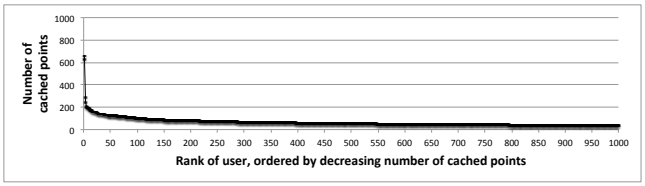

Figure 10: Distribution of cached points per user, top-1000, after running CASH (1000m).

than city's accuracy. Figure 8 illustrates this phenomenon by drilling down into CASH behavior at 2000m, 1000m, and 500m. While both neighborhood and city suffer the same 8.1% decrease in hit rate $Pr(hit)$ over that span, neighborhood's accuracy $Pr(correct|hit)$ increases by 20.2% whereas city's accuracy increases by only 2.5%.

Radial and CASH exhibit similar accurate cache hit rate performance, but CASH provides a faster $O(1)$ cache lookup time. Furthermore, CASH configured with a parameter of 1000m demonstrates good accurate cache hit rate of over 70% for neighborhood granularity and over 85% for city granularity, *both of which represent the reduction in the number of server calls that would have been needed to perform reverse-geocoding.* We thus use CASH parameterized at 1000m as the base configuration of our system, but we allow the developer or user to configure the scheme and parameters depending on application requirements.

Figure 9 shows the average number of points in each user's cache after iterating over all photos. These averages are misleadingly low because the distribution of photos per user is long-tailed, and thus users at the head of that distribution will have substantially more cached points. Figure 10 shows the distribution of cached points for each of the top-1000 users, ranked by the number of points, for CASH parameterized at 1000m. The head user has only 659 points in his cache, which is approximately 90 kBytes in encoded form.

5.6 Precomputed hash propagation

In Section 4.3 we considered hashes that were precomputed offline and then pushed to the user, thereby pre-filling the cache. In this separate experiment we created precomputed hashes of the San Francisco Bay Area at city gran-

Probability	Convex Hull	Radial (2000m)	Radial (1000m)	Radial (500m)	CASH (2000m)	CASH (1000m)	CASH (500m)	Time (60 min)	Time (5 min)
$Pr(correct, hit)$, neighborhood	0.520	0.642	0.705	0.731	0.652	0.700	0.721	0.553	0.475
$Pr(correct, hit)$, city	0.703	0.889	0.876	0.850	0.877	0.859	0.827	0.644	0.508

Table 3: Caching results for all 37,237 users.

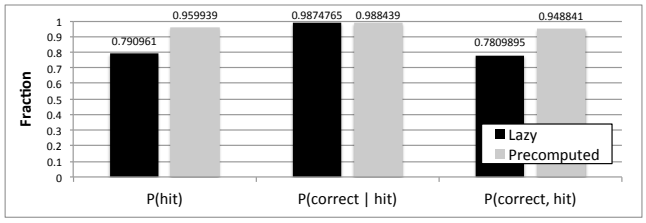

Figure 11: Improvements due to propagation of precomputed CASH hashing at 500m.

ularity with the smallest radius of 500 meters to maximize the number of needed points. We then pre-filled the caches for the 8190 users who took photos in this region.

Figure 11 shows the results for the three key probabilities $Pr(hit)$, $Pr(correct|hit)$, and $Pr(correct, hit)$ using the precomputed hashes and the "lazy" on-demand caching that we have been using throughout the paper. As expected, pre-filling the caches greatly improves the cache hit rate because there are more entries in the cache. For each of those hits, the accuracy remains the same because the size of the region entry does not change. The overall result is the accurate cache hit rate increases by 21%.

6. CONCLUSION

Reverse-geocoding plays an important role for many mobile pervasive applications by converting latitude & longitude coordinates into physical locations in the real world. However, commercial proprietary on-device mapping applications incur storage and monetary costs, while cloud-assisted lookup incurs battery discharge costs, the latter being the current best practice for reverse-geocoding. In this paper we reduced these costs by exploiting user geolocality and caching reverse-geocoded locations that were obtained from server calls. Using a 1.1M photo dataset, we showed that our cartographic sparse hashing scheme reduces the number of cloud server calls by over 70% for neighborhood granularity and by over 85% for city granularity. We additionally explored pre-filling user caches with hash results, which improved the caching hit rate. Finally, we showed that the system occupies relatively little space, with less than 1 MB of data being used by our hash encoding even for a complete precomputation of the San Francisco Bay Area.

In the future, we will look to expand our work by caching at street-level resolution, using other application scenarios, and measuring energy usage for entire days.

7. REFERENCES

[1] Apple iOS: Using Reminders. `support.apple.com/kb/HT4970?viewlocale=en_US&locale=en_US`

[2] Ariadne GPS application. `www.ariadnegps.eu`

[3] X. Cao, G. Cong, and C. Jensen. "Mining Significant Semantic Locations from GPS Data," In *Proceedings of VLDB*, 2010.

[4] K. Cheverst, N. Davies, K. Mitchell, and A. Friday. "Experiences of Developing and Deploying a Context-Aware Tourist Guide: The GUIDE Project," In *Proceedings of ACM MobiCom*, 2000.

[5] I. Constandache, S. Gaonkar, M. Sayler, R. Choudhury, and L. Cox. "EnLoc: Energy-Efficient Localization for Mobile Phones," In *Proceedings of IEEE Infocom mini-conference*, 2009.

[6] T. Donegan. "Smartphone cameras are taking over," *USA Today*, June 6, 2013.

[7] S. Fang and R. Zimmerman. "EnAcq: Energy-efficient GPS Trajectory Data Acquisition Based on Improved Map Matching," In *Proc. of ACM SIGSPATIAL*, 2011.

[8] J. Gemmell, G. Bell, and R. Lueder. "MyLifeBits: a personal database for everything," *Communications of the ACM*, 49(1), Jan. 2006.

[9] G. Goetz. "Taking, editing, and sharing photos in iOS7," Gigaom.com, September 21, 2013.

[10] The Google Geocoding API. `developers.google.com/maps/documentation/geocoding/`

[11] A. Guttman. "R-Trees: a Dynamic Index Structure for Spatial Searching," In *Proc. of ACM SIGMOD*, 1984.

[12] E. Kalogerakis, O. Vesselova, J. Hays, A. Efros, and A. Hertzmann. "Image Sequence Geolocation with Human Travel Priors," *IEEE ICCV*, 2009.

[13] M. Kjaergaard, J. Langdal, T. Godsk, and T. Toftkjaer. "EnTracked: Energy-Efficient Robust Position Tracking for Mobile Devices," In *Proc. of ACM MobiSys*, 2009.

[14] J. Liu, B. Priyantha, T. Hart, H. Ramos, A. Loureiro, and Q. Wang. "Energy Efficient GPS Sensing with Cloud Offloading," In *Proc. of ACM SenSys*, 2012.

[15] OpenStreetMap. `www.openstreetmap.org`

[16] OpenStreetMap Nominatim `nominatim.openstreetmap.org`

[17] OpenStreetMap - Planet.osm file. `wiki.openstreetmap.org/wiki/Planet.osm`

[18] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. "Fine-Grained Power Modeling for Smartphones Using System Call Tracing," In *Proc. of EuroSys*, 2011.

[19] S. Perez. "Newly redesigned Cluster makes photo sharing among small groups simpler, more personal," Techcrunch.com, April 23, 2013.

[20] T. Phan, A. Baek, A. Singh, and Z. Guo. "Caching Reverse-Geocoded Locations on Smartphones," In *Proceedings of IEEE ICCVE*, 2013.

[21] H. Samet. *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann, 2006.

[22] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. "Mining Interesting Locations and Travel Sequences from GPS Trajectories," In *Proceedings of WWW*, 2009.

[23] Z. Zhuang, K.-H. Kim, and J. Singh. "Improving Energy Efficiency of Location Sensing on Smartphones," In *Proc. of ACM MobiSys*, 2010.