SAC: Semantic Adaptive Caching for Spatial Mobile Applications *

Chang Liu
University of Maryland
liuchang@cs.umd.edu

Brendan C. Fruin
University of Maryland
brendan@cs.umd.edu

Hanan Samet
University of Maryland
hjs@cs.umd.edu

ABSTRACT

Mobile location-based applications rely heavily on network connections. When the mobile devices are offline, such applications become less accessible to users. A cache-based method is proposed to improve the offline accessibility for mobile location-based applications. The central idea is that when users are browsing information, the client program not only submits the current query window to the server, but also attempts to predict the most likely (from a probabilistic standpoint) query windows that would be submitted to the server in the future. The major challenge is the very large number of possible future query windows. This challenge is tackled by proposing a *discretization* technique that makes predictions over a finite subset of all possible query windows. A probabilistic model is proposed for prediction, which is trained using the query log recorded by the client, so that the prediction can be executed entirely on the client side. The advantage of this technique is that it requires no modification on the existing server side, so it can be adapted by most existing applications easily. The usability of the technique is demonstrated by prototyping it on top the NewsStand system so that the query window is constantly changing as users pan and zoom around the world using a gesturing interface, among others. Evaluation shows the prototype to be effective while decreasing the response time.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*; H.3.3 [**Database Management**]: Information Storage and Retrieval—*Information Search and Retrieval*

General Terms

Algorithms, Design, Performance, Reliability

Keywords

Adaptive caching, window queries, mobile devices

*This work was supported by the NSF under Grants IIS-10-18475, IIS-12-19023, IIS-13-20791, and CNS-1314857.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGSPATIAL'13, November 05 - 08 2013, Orlando, FL, USA
Copyright 2013 ACM 978-1-4503-2521-9/13/11 ... $15.00.

1. INTRODUCTION

Mobile location-based applications, are gaining increasing use in our daily lives. They are rooted in the desire to incorporate the functionality previously available only in geographic information systems (e.g., [16, 17]) for the representation of spatial, spatio-temporal, and distributed data (e.g., [7, 9, 14, 18, 24]). This had led to modern applications such as Google Maps[1], and Yelp[2]. People use them to search for local attractions. The accessibility of these applications becomes an important issue. If the mobile device hosting the applications goes offline, then the applications become inaccessible to users, and thus users' experiences are sacrificed. Existing work [4, 5, 6, 10, 12, 13, 23] has proposed a cache-based method to solve this problem. Such a method works well for those applications whose data is static, but has trouble dealing with applications with rapidly changing data, such as NewsStand [25], a system developed at the University of Maryland for online browsing of news using a map query interface.

In this paper, a semantic-aware cache-based approach, named SAC, is proposed to deal with applications with dynamic data. Instead of passively waiting for users to submit queries, SAC actively predicts which queries will be submitted in the future, and prefetches the answers for those queries. SAC is a layer on top of a location-based application, and thus brings two additional benefits: (1) no modification is needed on the server side to employ SAC; and (2) the prediction is based on users' query history, and thus provides a more accurate result for the user.

The key challenge of SAC is how to predict which queries will be submitted in the future. The target of SAC's predictor is to maximize the expected cache hits. Notice that the set of all possible queries is infinite, so that a naive model will assign each query with a probability 0, which is useless for the prediction problem.

To tackle these challenges we propose a map discretization method so that SAC can focus on a finite subset of queries. We then propose a probabilistic model to compute the probability of each query unit in the discretized query set to be submitted in the future. We then describe a tractable algorithm to find a set of queries to maximize the expected cache hit rate.

Much work [4, 5, 6, 10, 12, 13, 23] has studied client-side caching for mobile devices. The related work, however, will not cache previously unseen query results. Sun and Zhou [23] decompose the map into cells to enable a predic-

[1]http://maps.google.com
[2]http://www.yelp.com

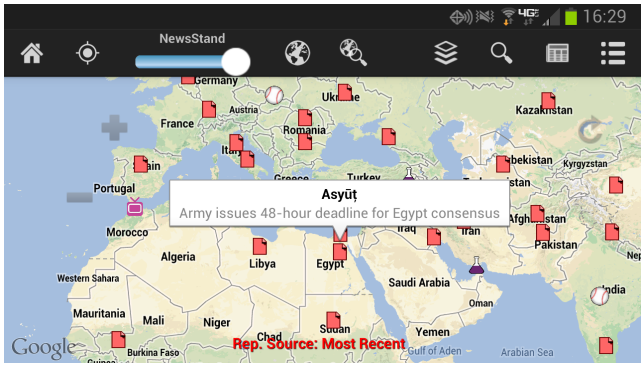


Figure 1: Screenshot of NewsStand for Android App on a Samsung Galaxy S3.

tive cache for zooming, but not for panning. Ren and Dunham [13] study predicting future queries based on the user's current location, direction and velocity derived from past locations. Amini et al. [4] developed Cache which groups a set of location queries to improve offline accessibility while providing a strong privacy. Different from these works, SAC targets at window queries and supports operations including zooming and panning.

SAC is applied to the NewsStand system [25]. NewsStand is a map-based application framework that aggregates and displays news from RSS feeds at their respective locations based on the content of location references in news articles [25]. Only the news stories associated with locations bounded by the current map viewing window are displayed. In the case of a mobile device, the current map viewing window would be the entire screen when the application is being used. Each time a user updates the map viewing window with an action (i.e. a swipe, click, pinch, or tap) a request is sent to the server for the news stories in the updated viewing window. The map-based approach of NewsStand affords an inherent granularity to search based on zoom level providing an approximate search.

NewsStand currently has a web interface and mobile interfaces for the Android (Figure 1) and iOS platforms. We apply SAC to the mobile applications in particular to the Android NewsStand application.

We summarize our contributions as follows:

1. We propose SAC, which is a cache layer on top of typical location-based applications, to improve the offline accessibility for applications; with static or dynamic data;
2. We propose a discretization method, which makes reasoning about the probability of queries possible;
3. We propose a probabilistic model to reason about the probability of a query being submitted in the future;
4. We implement a SAC prototype on top of the NewsStand mobile application. We evaluate this prototype with respect to both effectiveness, and efficiency. The result shows that SAC can achieve a good prediction accuracy, and the query response time decreases with respect to the no-cache solution.

The rest of this paper is organized as follows. Section 2 describes the SAC architecture, along with its key design of its predictor. Section 3 presents the prototype implementation

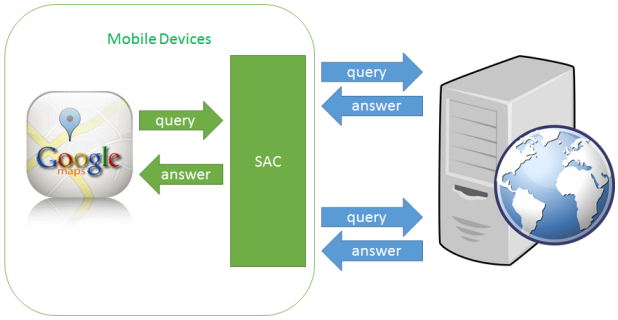


Figure 2: Illustration of how SAC works

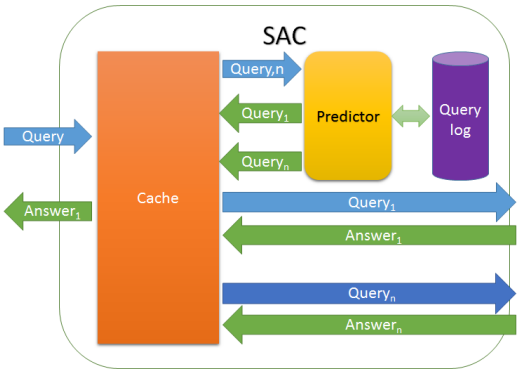


Figure 3: SAC architecture

on top of NewsStand. The evaluation results are discussed in Section 4. Section 5 reviews related work, while Section 6 contains directions for future research.

2. SAC

In this section, we introduce SAC, a cache layer on top of typical location-based mobile applications, which can improve offline accessibility. We first illustrate how SAC can make location-based applications on offline devices accessible to the users using a cache. Next, we formally define how to select content in the cache to maximize the expected cache hit rate. Finally, a probabilistic model is presented, and based on this model, prediction algorithms are developed to solve the cache selection problem.

2.1 Workflow

Figure 2 illustrates how SAC can improve location-based applications' availability when the devices are offline. In typical location-based applications, a client communicates directly with the server. As a result, once the mobile devices go offline, the client will not receive responses from the server, and thus the accessibility will be broken. SAC is a layer that sits on top of applications, that can store a set of queries along with their responses. Using SAC, a client instead submits the queries to SAC, and SAC will respond to the clients with the answers that it stores. If the devices are online, then SAC will actively submit non-cached queries to the server and store the responses in the cache. In doing so, once the mobile devices go offline, SAC is still able to respond to the clients' queries, when their responses are stored by retrieving cached elements.

Figure 3. shows the architecture of SAC which consists

of a cache (orange) and a predictor (yellow). The cache can store up to n queries and responses. Once a client submits a query, SAC transmits this query along with the cache size n to the predictor. The predictor computes the n predicted queries, submits the queries to the server that are not already in the cache and stores the n responses in the cache. The predictor guarantees that the original query can be responded to using these n predicted queries' responses. Those queries, along with their responses, which are stored in the cache, but are not in the set of n predicted queries returned by the predictor, are called stale. SAC drops all stale queries; queries remaining in the cache are called fresh.

Some queries and answers may remain in the cache for a long time, and for an application with rapidly changing data, they may become out of date. To overcome this problem, SAC allows setting a parameter τ, and all queries staying in the cache longer than τ are also made stale. SAC will drop them from the cache.

For those queries in the set of n predicted queries in SAC that are not fresh, SAC will submit them to the server. The queries' responses are labeled as fresh, and both the queries and the corresponding responses are stored in the cache. Once there are enough fresh answers to respond to the client, SAC will immediately transmit them to the client. To reduce the response time, SAC will asynchronously submit queries to the server in the order of their priority, i.e. those predicted queries whose answers can be used to respond to the client's original query have a higher priority.

If the query submitted by the client can be answered by fresh content in the cache, then no communication between the mobile devices and the server is needed, and thus a response to the query can be generated immediately. In this case, when responding to the previous query, the predictor in SAC already successfully predicted the next query. So the performance of SAC greatly depends on whether the predictor can predict the queries that will be submitted in the future based on the query submitted currently. The rest of this section is devoted to the predictor algorithms. To make the predictor as accurate as possible, it is necessary to know the queries that the clients has already submitted. In SAC, a query log (purple) is stored for this purpose.

2.2 Problem Definition

In this subsection, we formally define the prediction problem. A query Q is a quadruple

$$Q = \langle lon_{low}, lat_{low}, lon_{high}, lat_{high} \rangle,$$

representing a rectangle on the map, where lon_{low} and lon_{high} are the lower and higher longitudes respectively, and lat_{low} and lat_{high} are the lower and higher latitude respectively. Such a query is naturally a window area restricted by the device screen, so we use the terms, query and *query window* interchangeably.

The first prediction problem is defined as follows

PROBLEM DEFINITION 1 (NEXT QUERY). *Given query Q, and an integer $n > 0$, find a set QS of n queries, such that*

1. *Q can be answered by QS*
2. *QS maximizes $E(\delta(Q' \text{can be answerd by } QS)|Q)$, where Q' is the next query, and $\delta(\cdot)$ is the indicator function: $\delta(true) = 1$, and $\delta(false) = 0$.*

We defer the definition of "a query can be answered by a set of queries" to the next subsection, where this will become obvious. This definition imposes two requirements on the returned value QS of the predictor algorithm. First, Q must be able to be answered by QS. This requirement guarantees that SAC can respond to the current query. Second, QS maximizes the expected cache hit rate for the next query. We can rewrite the expected cache hit rate as follows:

$$\begin{aligned} & E(\delta(Q' \text{ can be answered by } QS)|Q) \\ = \; & Pr(Q' \text{ can be answered by } QS|Q) \end{aligned} \quad (1)$$

Equation 1 implies a naive solution to Problem 1: choose QS to be the top-n queries $Q_1, ..., Q_n$ which maximize the probability $Pr(Q' = Q_i)$ $(i = 1, ..., n)$. However, since there are infinite possible queries, for any particular Q'', $Pr(Q' = Q'') = 0$. Therefore, this naive solution cannot be adapted directly. In Section 2.3, we discuss how to use discretization to tackle this problem.

We want to maximize not only the next query's cache hit rate, but also the cache hit rate of the next few queries. Therefore, we define the second problem as follows.

PROBLEM DEFINITION 2 (NEXT k QUERY). *Given a query Q_0, an integer $k > 0$, and an integer $n > 0$, find a set QS of n queries, such that*

1. *Q_0 can be answered by QS*
2. *QS maximizes*

$$E(\sum_{i=1}^{k} \delta(Q_i \text{ can be answerd by } QS)|Q_0))$$

where Q_i is the i-th query submitted after Q_0 $(i = 1, ..., k)$.

Similarly, we have the following equation:

$$\begin{aligned} & E(\sum_{i=1}^{k} \delta(Q_i \text{ can be answered by } QS)|Q_0) \\ = \; & \sum_{i=1}^{k} Pr(Q_i \text{ can be answered by } QS|Q_0) \end{aligned} \quad (2)$$

and based on this equation, we know the solution to Problem 2 is the top-n queries maximizing

$$\sum_{i=1}^{k} Pr(Q_i = Q|Q_0)$$

Again, this value is 0 for any Q. We shall show that our discretization method is also the key to solving this problem.

2.3 Discretizing the map

As mentioned above, the first challenge is how to deal with the infinite size of the full query set. We show in this subsection that discretization can reduce the full query set to a finite query set, i.e., the *discretized query set*.

To simplify our discussion, we assume that the full map is a rectangle $\langle 0, 0, X, Y \rangle$, and that a query window is a rectangle $\langle x_1, x_2, y_1, y_2 \rangle$, where $0 \le x_1 < x_2 \le X$, and $0 \le y_1 < y_2 \le Y$. We discuss this transformation with respect to an application using a Mercator projection in Section 3.1.

Since the device screen has a fixed size, the query window should have the same shape, i.e. there should be a constant C, such that the following equation holds:

$$\frac{y_2 - y_1}{x_2 - x_1} = C \tag{3}$$

As a result, there exists a one-to-one mapping f between a query window $\langle x_1, x_2, y_1, y_2\rangle$ and a triple $\langle x_1, y_1, l\rangle$ where (x_1, y_1) is the lower-left corner of the query window, and l is the width:

$$f(\langle x_1, x_2, y_1, y_2\rangle) = \langle x_1, y_1, x_2 - x_1\rangle$$

$$f^{-1}(\langle x_1, y_1, l\rangle) = \langle x_1, x_1 + l, y_1, y_1 + lC\rangle$$

Now, according to Equation 3, the height can be computed as $l \times C$. We call (x_1, y_1) the *corner point*, and l the *scale*.

Discretization means choosing a finite subset of the full query set. To achieve this, we restrict the dimensions, x, y, and l, to take values from a finite value set.

In practice, l is naturally discretized. l usually takes values from a fixed set of values, $\{l_1, ..., l_d\}$, defined by the underlying system. The ratio of two adjacent scales, i.e. $\gamma = l_{i+1}/l_i$, should be a constant. The size of the scale set, denoted by d, is also defined by the system. In Android Google Maps v2, d is 18 correlating to the allowed zoom levels.

The scale defines not only the size of the query window, but also the detail level of the answers. A query window with a larger scale may have fewer responses than a query window with a smaller scale in the same region. For two query windows with the same scale; the responses in their overlapping area should be the same. As a result, the query Q can be answered by a set S of queries, if those queries in S can "cover" Q. Formally, we have the following definition.

Definition 1. *A query $Q = \langle x, y, l\rangle$ can be answered by a query set QS if and only if $\forall(p_x, p_y) \in \langle x, y, l\rangle.\exists(x', y', l) \in QS.(p_x, p_y) \in (x', y', l)$. Here a point (p_x, p_y) belongs to a query window $\langle x, y, l\rangle$ if and only if $x \le p_x \le x + l$ and $y \le p_y \le y + lC$.*

The next problem is how to discretize x and y. For a specific scale l, we define the discretized query set at scale l, denoted by DQS_l [3] , as follows:

$$DQS_l = \{(il, jlC, l) | i \in \{0, ..., \lfloor X/l \rfloor\}, j \in \{0, ..., \lfloor Y/lC \rfloor\}$$

Each query (x, y, l) can be answered by a set of four discretized queries in DQS_l: $(x_i l, y_i lC, l)$, $(x_i l + l,\ y_i lC, l)$, $(x_i l, y_i lC + lC, l)$, and $(x_i l + l, y_i lC + lC, l)$, where $x_i = \lfloor x/l \rfloor$, and $y_i = \lfloor y/l \rfloor$. To prove this point, it is enough to show that $x_i l \le x < x + l \le x_i l + 2l$ and $y_i \le y < y + lC \le y_i lC + 2lC$. Since $x_i = \lfloor x/l \rfloor$, we have $x_i \le x/l < x_i + 1$. Therefore, we have $x_i l \le x$ and $x + 1 < (x_i + 2)l = x_i l + 2l$. Similarly, we can prove $y_i \le y < y + lC \le y_i lC + 2lC$. Fig. 4 illustrates the relationship between a query (x, y, l), and its four discretized queries.

We define the discretized query set as

$$DQS = \bigcup_{1 \le i \le d} DQS_i$$

DQS has the following properties:

[3]Notice that for $i = \lfloor X/l \rfloor$, or $j = \lfloor Y/lC \rfloor$, the discretized query may violate the boundary requirement, i.e., $il + l > X$ or $jlC + lC > Y$. Since most servers can handle this case, we allow it to happen.

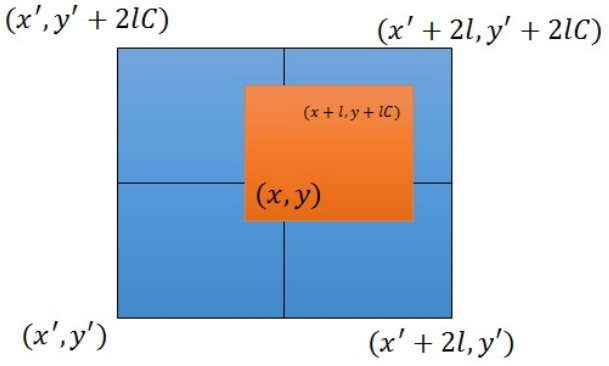


Figure 4: Query (orange) covered by four discretized queries (blue) with the same scale

Proposition 1. *The discretized query set is finite.*

Proposition 2. *Each query Q can be answered by a subset of the discretized query set containing up to 4 queries.*

As a result, the predictor algorithms can focus on queries from DQS instead of the full query set.

2.4 Semantic Adaptive Caching

To solve the accessibility problem, a naive approach is to use a cache to store all results returned from the server. A variety of algorithms [4, 5, 6, 10, 12, 13, 23] have been proposed to optimize the cache hit rate with respect to the limited cache size. Such an approach, however, suffers a major problem, that if a user has not submitted a query before, the result of the query will not be cached. To tackle this problem, we propose a probabilistic model to predict which queries are most-likely to be submitted in the following few query submissions. We present our model here, and the algorithms to compute the probability in Section 2.5.

Since any query can be covered by four discretized queries, we treat four discretized queries as a *query unit*, denoted by the lower-left query. The prediction problem is thus converted into one of computing the probability of the next unit based on the current unit. For a cache of size n, at most $\lfloor n/4 \rfloor$ query units can be stored in the system. To avoid cluttering, we slightly abuse the notion n to refer to the number of query units in the cache.

SAC's cache-update algorithm is given by Algorithm 1. A pair of training and prediction algorithms must be provided. SAC will first compute a model by running the training algorithm over the query log. Then, for each submitted query, SAC will run the prediction algorithm to compute a set QS of queries that are most likely to be submitted. SAC will then drop all queries not in QS, and add those queries in QS that are not in the cache. The key problem is how to design $\mathcal{T}$ and $\mathcal{P}$, discussed in the rest of this section.

In the following, we first show that the number of different operations in location-based applications is small, and how the operations on a query are mapped to operations on a query unit. Next, we propose our probabilistic model to predict the next query unit. Finally, we provide the algorithm to solve the next-k query problem.

Operations.

Data: $\langle \mathcal{T}, \mathcal{P} \rangle$ is a pair of training and prediction algorithms
Data: $\mathcal{C}$ is the cache
Data: n is the cache size, which is equal to $|\mathcal{C}|$
Data: *log* is the query log, which is a list of query units
Data: *para* is the parameters required by $\mathcal{P}$, other than n and the current query
$\pi \leftarrow \mathcal{T}(log)$
while $Q \leftarrow$ *a new query input* **do**
  $QS \leftarrow \mathcal{P}(Q, n, para)$
  for $Q \in \mathcal{C} \wedge Q \notin QS$ **do**
    Remove Q *from* C
  end
  for $Q \notin \mathcal{C} \wedge Q \in QS$ **do**
    Add Q *into* C
  end
end

Algorithm 1: SAC Cache Update Algorithm

Data: *log* is a list of query units
for $\alpha \in \Gamma$ **do**
  $\pi(\alpha) \leftarrow 0$
end
$sum \leftarrow 0$
for $i = 1 \rightarrow log.size() - 1$ **do**
  for $\alpha \in \Gamma$ **do**
    if $\alpha(log[i]) = log[i+1]$ **then**
      $sum \leftarrow sum + 1$
      $\pi(\alpha) \leftarrow \pi(\alpha) + 1$
    end
  end
end
for $\alpha \in \Gamma$ **do**
  $\pi(\alpha) \leftarrow \pi(\alpha)/sum$
end
return π

Algorithm 2: Training π

We formally define an operation α as a mapping from one query into another. In a location-based application, there are two ways to change the query window: panning or zooming. Panning only changes the position of the window on the map, but will not change the scale, while zooming will change both the position and the scale. Zooming operations can be further classified into three types: zooming-in, zooming-out, and pinching. The first two are the most common operations, since they only require one thumb to perform the operation, while the last one requires two fingers. In this work, we consider only three kinds of operations: panning, zooming-in, and zooming-out as pinching is a two-fingered variant of zooming.

A panning operation may move the current query unit to an adjacent query unit. For example, if the current query unit is $\langle i, j, l \rangle$, then by performing a panning operation, the next query unit may be $\alpha(\langle i, j, l \rangle) = \langle i + \Delta i, j + \Delta j, l \rangle$, where $\Delta i, \Delta j \in \{-1, 0, 1\}$. Notice that it is possible that $\Delta i = \Delta j = 0$. In this case, the panning operation does not change the query unit. We use the term *stay* for this operation as it leaves the query unit the same.

A zooming-in operation may change the current query unit at scale l into one of nine smaller query units at scale γl (for $\gamma > 1/2$). A query unit $\langle i, j, l \rangle$, performing a zooming-in operation may result in one query unit from the set $\{\langle i', j', \gamma l \rangle : \lfloor i/\gamma \rfloor \leq i' \leq \lfloor (i+2)/\gamma \rfloor, \lfloor j/\gamma \rfloor \leq j' \leq \lfloor (j+2)/\gamma \rfloor\}$.

A zooming-out operation is different from the other two types of operations in that a zooming-out operation will always change the current query unit at scale l into one query unit at scale $2l$. In particular, if the current query unit is $\langle i, j, l \rangle$, then the result query unit from performing zooming-out will be $\langle \lfloor i/2 \rfloor, \lfloor j/2 \rfloor, 2l \rangle$.

A Probability Model for Query Prediction.

Assume the operation set is Γ, whose each element α is a mapping from the query unit set to the query unit set. Intuitively, $\alpha(q)$ is the result query unit from performing operation α over the query unit q.

A *model*, π, is a mapping from Γ to $[0, 1]$, such that

$$\sum_{\alpha \in \Gamma} \pi(\alpha) = 1$$

Therefore, π is a distribution over Γ.

Given a model π and the current query unit Q_0, in order to compute the probability of a query unit Q to be the next one submitted Q', it is sufficient to compute

$$Pr(Q' = Q|Q_0) = \sum_{\alpha \in \Gamma} \delta(\alpha(Q_0) = Q) \times \pi(Q_0) \qquad (4)$$

Further, to predict the most probable query units in the next k submissions, the key question is how to compute $Pr(Q_i = Q|Q_0)$ for a given Q, and $i \in \{1, ..., k\}$. To this end, we make two assumptions: (1) that the query submissions satisfy the Markov property. That is, the probability of a query that will be submitted next only depends on the current query. (2) the operation distribution is stationary. That is, π remains the same over time.

Based on these two assumptions, we have

$$\begin{aligned} & Pr(Q_i = Q|Q_0) \\ = & \sum_{Q' \in DQS} Pr(Q_i = Q|Q_{i-1} = Q') Pr(Q_{i-1} = Q'|Q_0) \\ = & \sum_{Q' \in DQS} \sum_{\alpha \in \Gamma} \delta(Q = \alpha(Q')) \pi(\alpha) Pr(Q_{i-1} = Q'|Q_0) \end{aligned} \qquad (5)$$

2.5 Predicting Algorithm

We first discuss Problem 1. Based on Equation 4, the probability $Pr(Q' = Q|Q_0)$ can be directly computed, if π is known. Therefore, the key challenge is to learn π. Since there are a limited number of operations, we compute the frequency of each operation α to be taken from the query log, and treat it as $\pi(\alpha)$. The training algorithm is presented in Algorithm 2. In this algorithm, initially for every operation α, $\pi(\alpha)$ is set to 0. Then the algorithm scans through the query log, to identify which operation is performed to move from the previous query unit to its following query unit. During this loop, the frequency of each operation taken is recorded, along with *sum*, the total number of operations taken. Finally, $\pi(\alpha)$ is normalized using *sum* to ensure $\sum_{\alpha \in \Gamma} \pi(\alpha) = 1$.

The query prediction algorithm for Problem 1 is provided in Algorithm 3. This algorithm computes the probability of each query unit according to Equation 4. Since the next

Data: Q_0 the current query unit
Data: n the cache size
$QS \leftarrow \emptyset$
for $\alpha \in \Gamma$ **do**
  $QS[\alpha(Q_0)] = \pi(\alpha)$
end
sort $QS = \{(Q, p)\}$ *by* p
return top-n *elements from* QS

Algorithm 3: Predicting the next query

query must be one that we can achieve by performing an operation over the current query unit, we can ignore all other queries, whose probability to be submitted is zero. We use a dictionary, i.e. QS, to store queries along with their probabilities of being submitted next. We enumerate each possible operation α, performing α over the current query unit Q_0 to get the next query unit $\alpha(Q_0)$, and assign its probability, $QS[\alpha(Q_0)]$, to $\pi(\alpha)$. We finally sort these query units according to their probabilities, and return the top-n one.

Notice, that this algorithm is equivalent to choosing the top-n common operations in Γ in the query log, which should provide us the best guess on the next operation. Furthermore, it is worth mentioning that not all possible queries are concerned here since there are operations that we do not consider, such as pinching and keyword search.

Next, we discuss Problem 2. Equation 5 suggests the following algorithm to compute $Pr(Q_i = Q|Q_0)$: The algorithm iteratively goes from $i = 1$ to $i = k$, and in each iteration, $Pr(Q_i = Q|Q_0)$ are computed for all Q such that $Pr(Q_i = Q|Q_0) > 0$. This algorithm, however, has a time and space complexity exponential to k. Since zooming-in operations will produce 9 new query units with a positive probability in each iteration, there will be at least 9^k query units with positive probabilities. Therefore, the space and time complexity is $\Omega(9^k)$. Thus this algorithm is impractical for large k values.

To tackle this problem, we developed an approximation algorithm, given in Algorithm 4. This algorithm accepts one more argument, the threshold T. The difference is that, after each iteration, NEW_OS drops all query units except those top-T ones with the highest probability. The query units on the long tail (with small probability) are probabilistically less likely to be submitted by the client. Therefore, a threshold T is set to cut those queries on the tail.

The time complexity of Algorithm 4 is $O(k|\Gamma|T \log |\Gamma|T)$. To show this, at the beginning of each iteration, NOW_QS contains at most T entries. For each entry, there will be at most Γ entries generated and stored in NEW_QS. So at the end of each iteration (before the sorting in each iteration), NEW_QS contains at most $T|\Gamma|$ query units. Sorting will dominate the time complexity, so the total time complexity is $O(|\Gamma|T \log |\Gamma|T)$ for each iteration.

3. IMPLEMENTATION

We applied the proposed SAC algorithm to the Android version of NewStand. This section describes the methods by which SAC was implemented on NewsStand and how its map projection can be divided into a grid. We then describe how caching can be applied to the client side of NewsStand.

3.1 Grid Structure

Data: Q_0: the current query unit
Data: n: the cache size
Data: k: next k moments
Data: T: threshold
$QS \leftarrow \emptyset$
$NOW_QS \leftarrow \{Q_0 : 1\}$
for $i = 1 \rightarrow k$ **do**
  $NEW_QS \leftarrow \{\}$
  for $(q, p) \in NOW_QS$ **do**
    for $\alpha \in \Gamma$ **do**
      $nq \leftarrow \alpha(q)$
      if *not* $NEW_QS.containsKey(nq)$ **then**
        $NEW_QS[Q] = \pi(\alpha) \times p$
      end
      else
        $NEW_QS[Q] = NEW_QS[Q] + \pi(\alpha) \times p$
      end
    end
  end
  for $(q, p) \in NEW_QS$ **do**
    if *not* $QS.containsKey(q)$ **then**
      $QS[Q] = p$
    end
    else
      $QS[Q] = QS[Q] + p$
    end
  end
  sort $NEW_QS = \{(q, p)\}$ *by* p
  keep only top-T *elements from* NEW_QS
  $NOW_QS \leftarrow NEW_QS$
end
sort $QS = \{(q, p)\}$ *by* p
return top-n *elements from* QS

Algorithm 4: Predicting the next k queries

The Android version of NewsStand uses the Google Maps Android API v2 [2] for its map. The Google Maps Android API uses the spherical Mercator map projection based on the WGS-84 coordinate system used in many commercial mapping APIs including OpenStreetMap, Bing Maps and HERE Maps which preserves the angles of the meridians, but as a result distorts the size and shape of large areas as the object moves away from the Equator. By limiting the maximum latitude to 85.05112878° North and the minimum latitude to 85.05112878° South, the map projection is a square [1]. This square can be mapped to a coordinate grid structure where 179° West and 85.05112878° South corresponds to the point (0,0) and 180° E and 85.05112878 ° N corresponds to the point (360, 360). Latitude and longitude coordinates can be mapped to this grid structure [3] with longitude, *lon*, directly mapping to the first dimension:

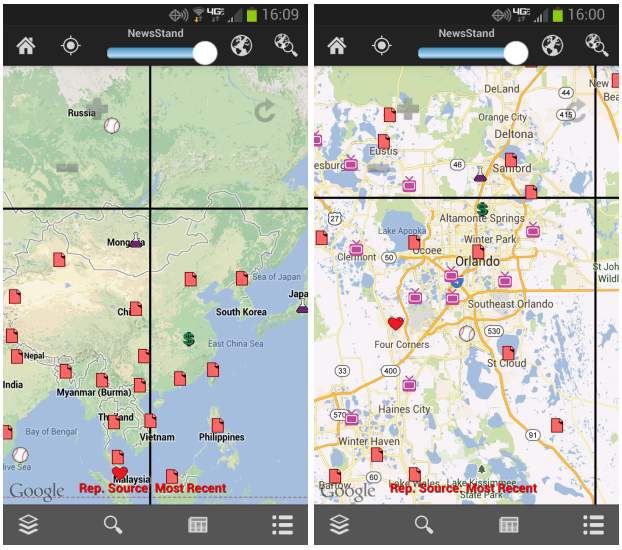
$$x = lon + 180$$

and latitude, *lat*, being mapped to the grid with the following formula:

$$y = \frac{180}{\pi} \log\left(\tan\left(\frac{\pi}{4} + \frac{lat}{2} * \frac{\pi}{180}\right)\right) + 180$$

The respective grid coordinates can be mapped back to the projection with longitude:

$$lon = x - 180$$



(a) (b)

Figure 5: Examples of the grid structure (outlined in black) applied to the NewsStand Android App with (a) showing the grid cells at the maximum zoom out level and (b) showing the grid cells at zoom level 9.

and latitude:

$$lat = \frac{180}{\pi}\left(2\arctan\left(e^{((y-180)*\frac{\pi}{180})}\right) - \frac{\pi}{2}\right)$$

We then subdivide the grid into equal-sized rectangles based on the current viewing window and the zoom level based on a set of simple rules. The width and the height of the rectangular grid cells must be at least as large as the corresponding width and height of the map viewing window where the width and height correspond to the points of the projection's coordinate system visible for a current window given a zoom level. The width and height must also evenly divide 360, the square projection's side length. For example a Samsung Galaxy S3 at the lowest zoom level (furthest zoomed out), contains a map viewing window of approximately 63.28 grid units for its width and 91.23 grid units for its height and the grid at this level is divided into cells that have width 72 units and height 120 units as seen in Figure 5(a). While zooming in to the city level (zoom level 9) has a map viewing window of approximately 0.98 grid units for its width and 1.43 grid units for its height has a grid that is divided into cells that have width 1 unit and height 2 units as seen in Figure 5(b). Given the fact that all grid cells must be at least as large as the current map viewing window, we know that any map viewing window is contained in one, two or four grid cells.

3.2 Client Caching

Due to the nature of news needing to be up-to-date and window queries being sent even with the most minor of updates to the map viewing window, some form of caching must take place in order to reduce duplicate or near-duplicate queries from having to be recalculated. As was seen in Section 3.1, the proposed grid structure could be used so that all window queries could be answered from the sum of at most four grid cells. By keeping track of the results of previously seen grid cells and predicting future grid cells, the grid cell's associated data can be stored in a cache and future requests for these cells may not need to be calculated.

```
Data: window is the current map viewing window
Data: C is the current cache
Data: query_log is the query log of the last 200 queries
grid_cells = calculate grid cells associated with window
cached_results = initialize to empty
for curr_cell ∈ grid_cells do
    cell_cached = false
    for cached_cell ∈ C do
        if curr_cell = cached_cell then
            cell_cached = true
            if cached_cell is downloaded then
                Append cell_cached data to
                cached_results
            end
        end
    end
    if cell_cached = false then
        request data for curr_cell from server
    end
end
Wait (for all grid_cells)
Display (results of grid_cells)
predicted_cells = predict (grid_cells, query_log)
for predicted_cell ∈ predicted_cells do
    cell_cached = false for cached_cell ∈ C do
        if predicted_cell = cached_cell then
            cell_cached = true
        end
    end
    if cell_cached = false then
        request data for predicted_cell from server
    end
end
```

Algorithm 5: Map Window Change

We begin by initializing an empty cache, C, to store the results of previously seen or current grid cells with their associated data and the time that they were downloaded from the server and a τ value which is the maximum time downloaded data can be used before it is considered stale as described in Section 2.1. Each time the map viewing window is updated begin by removing all elements from C whose download time is greater than τ which in the case of NewsStand should be fairly short (since it deals with streaming news) such as 120 seconds. Once the stale elements have been removed, pass the current cache C and the current map viewing window to Algorithm 5.

Algorithm 5 begins by calculating the grid cells that intersect with the current window storing the intersected grid cells in *grid_cells* and initializes an empty results list, *cached_results*. Each grid cell in *grid_cells* is then checked for membership in C. If a grid cell in *grid_cells* is not contained in C, an asynchronous server request is dispatched to download the associated data. If a grid cell in *grid_cells* is contained in C and the cached grid cell has finished downloading the data, then the results are appended to the *cached_results*. Note that due to the requests to the server being sent asynchronously, a grid cell may exist in the cache that is still waiting for its data to be returned from the server. On com-

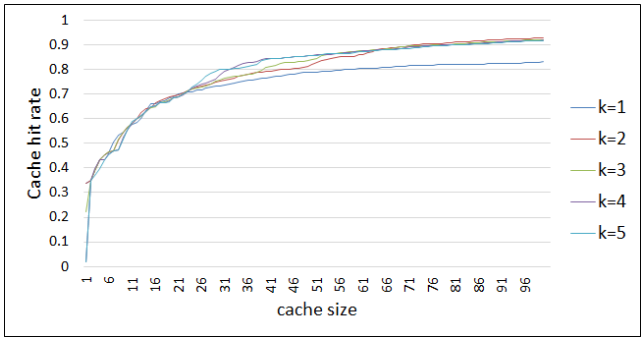


Figure 6: Cache hit rate on server side query log

k	1	2	3	4	5
cache size	17	49	52	66	82

Table 1: Cache size to achieve 90% hit rate

pletion of an asynchronous call associated with the current map viewing window, the data is appended to *cached_results* and a check is made to determine whether all of the current grid cells have been downloaded. The *cached_results* are not displayed until all of the data for the current grid cells have been downloaded thereby preventing data from populating in only certain grid cells of the map while other grid cells remain empty.

After each grid cell in *grid_cells* has been checked for membership in C and the server has responded with all windows not in C, the map is updated with the results from *grid_cells*. Future grid cells are then predicted using *grid_cells* and the query log of the last 200 window queries, *query_log*, using the algorithm proposed in Section 2.5 and the results are stored in *predicted_cells*. Each grid cell in *predicted_cells* that is not contained in C sends an asynchronous server request to retrieve the data for the current grid cell. C is set to the union of the *grid_cells* with their associated data and the *predicted_cells* with their associated data.

4. EVALUATION

We evaluate our implementation with respect to both effectiveness and efficiency. The NewsStand server is deployed in a cluster [11]. The mobile device is a Samsung Galaxy S3 running Android 4.1.2.

4.1 Study using NewsStand query log

We first conduct an empirical study on a real query log from NewsStand system. We collected all queries submitted to the system during September 2012 to November 2012. We cleaned up the query log to contain only those queries submitted by performing a window movement operation. There are 10,105 queries in total. 75% of all queries are used as training data, and the remaining 25% are used as testing data.

We use Algorithm 4 as the prediction algorithm, since Algorithm 3 is a special case of Algorithm 4 by setting $k = 1$. For each query in the test data, we run the algorithm to get a set of queries, and we count how many times the next query in the list appears in this set.The cache hit rate is the ratio between this count and the total number of queries.We vary the cache size from 1 to 100 and k from 1 to 5, and compute

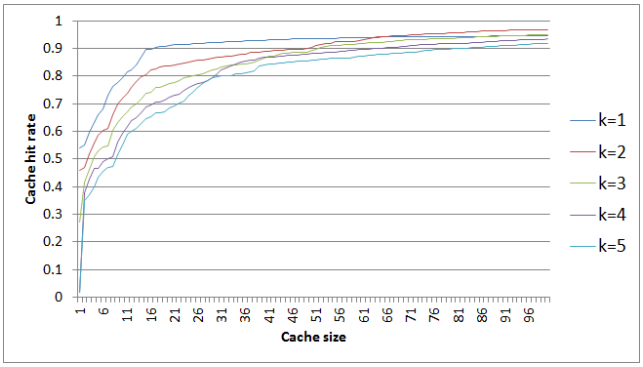


Figure 7: Cache hit rate for next 5 queries on server side query log

the ratios. Figure 6 plots the cache hit rate versus cache size. From the plot, we can observe that Algorithm 3, i.e. $k = 1$, can achieve a 90% cache hit rate once the cache size becomes larger than 17.

We further observe that the curve for a larger k lies below the curve for a smaller k. We attribute this to the fact that for a larger k, the algorithm favors the queries that will be submitted in the future, but we only count the next one query. We also observe that for all k, the cache hit rates achieve more than 90% when the cache size is 100. Table 1 shows the smallest cache size for each k, which the cache hit rate achieves 90%.

Finally, we study the offline accessibility of Algorithm 4 by measuring the cache hit rate in the next 5 query submissions. Figure 7 shows the plots of the cache hit rate versus cache size for $k = 1$ to 5. For $k = 1$, which is equivalent to Algorithm 3, the cache hit rate is lower than 90% even for a cache size 100. For $k > 1$, the algorithm significantly improves the cache hit rate. But we observe no big differences among $k = 2, 3, 4$ or 5. This phenomenon illustrates that the queries after at most 5 operations that users are most likely to perform, can also be achieved by performing two operations. We attribute the reason for this to be that users are more likely to redo their recent old queries.

4.2 Study on mobile devices

We studied the performance of SAC on a mobile device by recording window movements on the NewsStand Android application. We created and analyzed three datasets and evaluated the effectiveness of our cache along with the query running times.

Datasets.

We simulated users inputs by recording window movements on the NewsStand Android application on a Samsung Galaxy S3 capturing three different datasets. Each dataset contained 200 window movements. Dataset 1 consisted of primarily (over 60%) panning operations while Dataset 2 consisted of primarily (over 60%) zooming operations. Dataset 3 contained a similar amount of pan and zoom operations. The results for both the effectiveness and running time are the average of each of the datasets of three subsequent runs to control for varying overhead when communicating with the server.

k	0	1	2	3	4	5
Dataset 1						
Cache Hit (%)	53.6	55.0	55.0	54.5	54.0	52.4
Coverage (%)	73.1	75.1	74.0	74.0	73.7	73.0
Dataset 2						
Cache Hit (%)	44.0	45.2	45.0	44.8	44.2	43.8
Coverage (%)	58.7	60.9	61.0	60.3	60.6	59.9
Dataset 3						
Cache Hit (%)	68.7	70.0	69.3	68.2	68.5	69.0
Coverage (%)	80.0	81.0	81.0	80.7	80.6	80.4

Table 2: Effectiveness study on mobile devices looking at the cache hit and coverage percentage for three datasets.

Effectiveness.

To evaluate the effectiveness, we computed the *cache hit* rate and the *coverage*. The *cache hit* rate is computed with the same method as in the query log which is the percentage of queries that can be completely answered by the cache. For some queries, even though not all responses are in the cache, a partial result may be available. The *coverage* computes the percentage of the responses for the next query that can be answered based on the contents in the cache.

We vary k from 0 to 5, and evaluate the system using a cache size of 100. Note that when k is zero we are using a cache with no prediction, i.e. we are storing only previously accessed windows in the cache so by comparing it to $k > 0$ we can measure our predictions' success. The results are shown in Table 2. As can be seen the cache hit varies from 43.8% in Dataset 2 to 70.0% in Dataset 3 while Dataset 1 falls in between. The highest cache hit rate is achieved when k is set to 1 in all three datasets (note k=2 gives same cache hit rate for Dataset 1). The coverage percentage varies from 58.7% in Dataset 2 to 81.0% in Dataset 3 while Dataset 1 again falls in between. Dataset 1 has the highest coverage percentage when k is set to 1 and in Dataset 2 the highest coverage percentage is achieved when k is set to 2. In Dataset 3, the highest coverage percentage is when k is set to 1 or 2. We see that both the cache hit rate and the coverage are quite similar for different k for a given dataset. We observe that overall the results for $k = 1$ and $k = 2$ are slightly better than those for $k > 2$. This observation is the same as we observed in the study of the query log. We believe that this is due to our prediction algorithm only looking at the overall past movements and not with respect to the user's current location or their most recent of requests.

The cache hit rate shown in Table 2 means that over 43.8% of all window queries can be answered without additional communication to the server since the results were previously cached. Additionally, over 58.7% of the discretized windows are in the cache. Important to note, is that the simulated dataset most like a typical user, Dataset 3, can achieve both a 70.0% cache hit rate and 81.0% coverage percentage when k is set to 1. This allows for high usability when SAC is in offline mode due to limited or no connectivity.

Running Time.

We evaluated the query response time using SAC compared with a no-cache solution. For SAC, we varied k from 0 to 5, and the cache size was 100. The average response time for Dataset 3 for varying values of k and the no-cache solution is plotted in Figure 8. From this figure, we observe that the no-cache solution has a more stable response time with periodic longer queries, while SAC's response time is more unstable with many queries taking under 50 milliseconds when SAC uses the cache. For those cache hit queries, SAC's response time is less than the no-cache solution. The reason is that the no-cache solution will always submit a query to the database server, while SAC will respond to a user immediately. For cache miss queries, SAC may result in a larger latency as up to four queries will be requested from the server. Even though these are asynchronous calls, the database server's response time increases when responding to those queries and the client's resources are divided in order accommodate each of the requests.

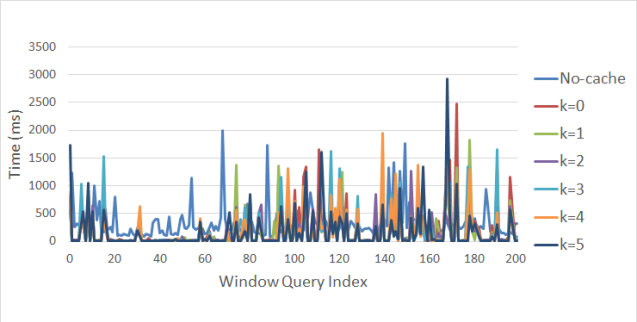


Figure 8: Query response time (ms)

k	0	1	2	3	4	5	No-cache
Dataset 1 (ms)	260	216	209	204	183	201	263
Dataset 2 (ms)	272	266	261	280	294	326	350
Dataset 3 (ms)	188	162	147	170	171	161	325

Table 3: Query response on mobile device (in ms)

The average response times in milliseconds for the three datasets is shown in Table 3. Here we see that in all circumstances, including when k=0, on the average the cache outperforms the no-cache solution. We see that by setting k to 2 in Dataset 3 we get over a 50% reduction in average query response time. Interestingly, an increase in k may decrease the average query response time even though there is a decrease in the cache hit rate and coverage as was seen in Table 2. This is most likely due to queries being predicted that are eventually used, but not necessarily when they were expected to be called. In some cases, the SAC query response time decreases compared to the decrease in the no-cache solution may be considered negligible, but more important is the fact that these windows are available offline.

5. RELATED WORK

Client-side caching for mobile devices has been widely researched [4, 5, 6, 10, 12, 13, 23]. Barbará and Imieliński [5] studied the issue of a large number of mobile devices querying remote databases assuming there would be periods of both awake and sleep times for the devices similar to current mobile applications switching between other applications or the devices being on standby. In this model, they believed that cached data would have to be explicitly invalidated with an invalidation report sent from the server while our invalidation is determined on the client side requiring no additional communication with the server. Lee et al. [10]

looked into semantic query caching for mobile devices and had a timer associated with each caching unit where on expiration a refresh would be sent to the server to update the expired cache data. The cache accommodation used by Lee et al. is similar to our approach in that it utilizes semantic locality, adapting to the patterns of query results. However, their proposed caching algorithm only accounts for previously seen query results while our algorithm uses both predictive analysis of future queries and the results of previously seen queries resulting in an increase in network flow in order to potentially reduce wait times.

Semantic caching for mobile devices has been researched with regards to spatial applications [4, 13, 23]. Similar to our work, Ren and Dunham [13] looked into applying semantic caching techniques to store location queries for areas that may have wide areas of overlap in order to answer future queries locally as opposed to always making server calls. They looked at predicting future queries based on the user's current location, direction and velocity derived from past locations, while we look at a comparable problem of caching and predicting window queries. Sun and Zhou [23] created a system for semantic caching of location queries by iteratively decomposing the map into cells which they term Peano cells which are able to answer queries at different zoom levels based on the combination of lower level Peano cells. These different levels of caching allow for zooming to be contained in the cache, but not panning without predictive caching or prefetching. Based on the level of the decomposition, the proposed grid structure may require a large amount of space to be allocated to the cache for each of the Peano cells. Amini et al. [4] developed Caché which groups a set of location queries in order to issue one large request to the server. By issuing one large request, the user is granted stronger location privacy in that their exact location is not continuously sent to the server and offline accessibility is improved by downloading and storing and a larger area on the client. This work primarily considers point queries while we focus on window queries.

6. DIRECTIONS FOR FUTURE WORK

Future work includes improving our prediction algorithm so that as the number of predictions increases the cache hit rate and the coverage percentages increase as well. Possible improvements for our prediction algorithm include taking into account the user's current location and what the user's past actions were at this location. For example, in a general case a user may be more likely to zoom in over a land mass or to pan over an ocean. Greater improvements to the prediction algorithm may come from more specific prediction based on a current grid cell or collection of grid cells. For example, user A may live in Orlando, FL so that s/he often zooms into grid cells that contain Orlando while as user B from Miami, USA may zoom into some grid cells that contain Orlando since it is relatively close to Miami from a world perspective, but at a close enough zoom level may pan to the south in order to reach Miami. In addition to historical prediction, prediction can be modified to give weighting to recent queries, i.e. given the user has panned to the left 5 times in a row the chance that s/he will continue in this direction is increased.

Additional future work involves caching the results of other operations which can be the subject of subsequent zooming and panning operations. This is especially relevant for nearest neighbor (e.g., [15, 19, 22]), shortest path (e.g., [20, 21]), and spatial join (e.g., [8]) queries.

7. REFERENCES

[1] Google map types. `https://developers.google.com/maps/documentation/javascript/maptypes`.

[2] Google maps android api v2. `https://developers.google.com/maps/documentation/android/`.

[3] OpenStreetMap Mercator. `wiki.openstreetmap.org/wiki/Mercator`.

[4] S. Amini, J. Lindqvist, J. I. Hong, J. Lin, E. Toch, and N. M. Sadeh. Caché: caching location-enhanced content to improve user privacy. In *MobiSys*, pages 197–210, 2011.

[5] D. Barbará and T. Imieliński. Sleepers and workaholics: caching strategies in mobile environments. In *VLDB*, pages 567–602, 1995.

[6] B.-G. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *MobiSys*, pages 141–154, 2012.

[7] G. R. Hjaltason and H. Samet. Speeding up construction of PMR quadtree-based spatial indexes. *VLDBJ*, 11(2):109–137, 2002.

[8] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *VLDB*, pages 606–618, 1995.

[9] G. S. Iwerks, H. Samet, and K. Smith. Maintenance of spatial semijoin queries on moving points. In *VLDB*, pages 828–839, 2004.

[10] K. C. K. Lee, H. V. Leong, and A. Si. Semantic query caching in a mobile environment. In *SIGMOBILE*, pages 28–36, 1999.

[11] M. D. Lieberman and H. Samet. Supporting rapid processing and interactive map-based exploration of streaming news. In *GIS*, pages 179–188, 2012.

[12] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Web caching on smartphones: ideal vs. reality. In *MobiSys*, pages 127–140, 2012.

[13] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *MOBICOM*, pages 210–221, 2000.

[14] H. Samet. A quadtree medial axis transform. *CACM*, 26(9):680–693, 1983.

[15] H. Samet. K-nearest neighbor finding using MaxNearestDist. *TPAMI*, 30(2):243–252, 2008.

[16] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *CACM*, 46(1):63–66, 2003.

[17] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using quadtrees. *Pattern Recognition*, 17(6):647–656, 1984.

[18] H. Samet and M. Tamminen. Bintrees, CSG trees, and time. In *SIGGRAPH*, pages 121–130, 1985.

[19] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.

[20] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *ICDE*, pages 652–663, 2009.

[21] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.

[22] J. Sankaranarayanan, H. Samet, and A. Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics*, 31(2):157–174, 2007.

[23] S. Sun and X. Zhou. Semantic caching for web-based spatial applications. In *APWeb*, pages 783–794, 2005.

[24] E. Tanin, A. Harwood, and H. Samet. A distributed quadtree index for peer-to-peer settings. In *ICDE*, pages 254–255, 2005.

[25] B. Teitler, M. D. Lieberman, D. Panozzo, J. Sankaranarayanan, H. Samet, and J. Sperling. Newsstand: A new view on news. In *GIS*, pages 144–153, 2008