Caching Geospatial Objects in Web Browsers (Demo Paper)

Taewoo Kim #1, Vidhyasagar Thirumaraiselvan +2, Jianfeng Jia #1, Chen Li #1
University of California, Irvine
[1] {taewok2,jianfenj,chenli}@ics.uci.edu, [2] thirumav@uci.edu

ABSTRACT

Map-based services are becoming increasingly important in many applications. These services often need to show geospatial objects (e.g., cities and parks) in Web browsers, and being able to retrieve such objects efficiently is critical to achieving a low response time for user queries. In this demonstration we present a browser-based caching technique to store and load geospatial objects on a map in a Web page. The technique employs a hierarchical structure to store and index polygons, and does intelligent prefetching and cache replacement by utilizing the information about the user's recent browser activities. We demonstrate the usage of the technique in an application called TwitterMap for visualizing more than 1 billion tweets in real time. We show its effectiveness by using different replacement policies. The technique is implemented as a general-purpose Javascript library, making it suitable for other applications as well.

Keywords

Map services, Geospatial objects, Caching, Prefetching, Replacement, TwitterMap

1. INTRODUCTION

Map-based services are becoming increasingly important in many applications. For example, a Tableau application [12] shows the history of the U.S. on a map. A MapD demo [6] shows taxi dropoff locations in New York. Such map services allow users to get insights of the spatial aspect of the data. Moreover, by linking the spatial results on the map with other analytical results, they provide an interactive interface that allow users to explore and visualize the underlying data. TwitterMap [5] currently developed by our team supports interactive exploration and visualization of more than 1 billion tweets on a map by allowing users to specify spatial, temporal, and keyword conditions.

As the user explores the map by doing zoom-in, zoom-out, panning operations, the service needs to efficiently update the interface with the related geospatial objects in the query region. The interface often shows two types of geospatial objects. The first type is image tiles, possibly at various levels with different resolutions. These objects are typically static, with fixed shapes and locations. The second type is geospatial objects that have various shapes and distributions at different locations. For instance, in TwitterMap, a user can type in keywords, specify a time interval, and choose a spatial region, then the system can show the tweet aggregation results per state, county, or city for the specified query region satisfying those conditions, while each of these geospatial entities has its own polygon shape. In order to support interactive exploration of the underlying tweet data, it is important to answer each such request efficiently, ideally within one second. It is challenging to efficiently retrieve the relevant geospatial objects and display them in the browser, especially when the number of relevant objects is large and the network bandwidth is limited. It is infeasible to store all these objects in the browser due to their large number. For instance, in TwitterMap, there are about 30,000 city polygons with a total size of 50 megabyte in the geoJSON format. Depending on the network speed, it can take up to seconds or even minutes to transfer the polygons. Moreover, if the browser needs to render all the 30,000 polygons using different thematic colors on the page, the required memory size in the Web page can increase up to one gigabyte, which can cause the Web page not to be responsive.

In this paper we study how to reduce query-response time by caching geospatial objects in a Web browser. We focus on the second type of objects that have various shapes and region distributions, making their caching technically more challenging than the first type. Our proposed technique employs a hierarchical indexing structure to store and index polygons inside the browser and support fast retrieval of polygons matching a query region. It also supports intelligent prefetching and cache replacement by utilizing the information about the user's recent activities in the browser to predict the next requested region. The technique is implemented as a general-purpose Javascript library, making it a solution suitable for many services. We will demonstrate the technique using the TwitterMap service. Figure 1 shows a screenshot of its interface.

1.1 Related Work

There is existing research on server-side caching in map-based Web services [10, 13]. These techniques are not suitable for our setting since we focus on caching in the Web browser to reduce the communication cost between the frontend and the backend. This setting requires techniques based on Javascript that runs inside a Web browser. The general-purpose browser-level Web storage [8, 7] allows a Web page to maintain a key-value database in the browser, whereas a geospatial query often involves a search region that is different from the key-value search interface. Our cache technique is able to efficiently retrieve the matching geospatial objects by a multi-dimensional spatial range query. There are also several front-end spatial index solutions, e.g., [11], to store complex spatial objects. We use one of these solutions as part of our

SIGSPATIAL'17 November 7–10, 2017, Los Angeles Area, CA, USA

ACM ISBN 978-1-4503-5490-5/17/11.
DOI: https://doi.org/10.1145/3139958.3140043

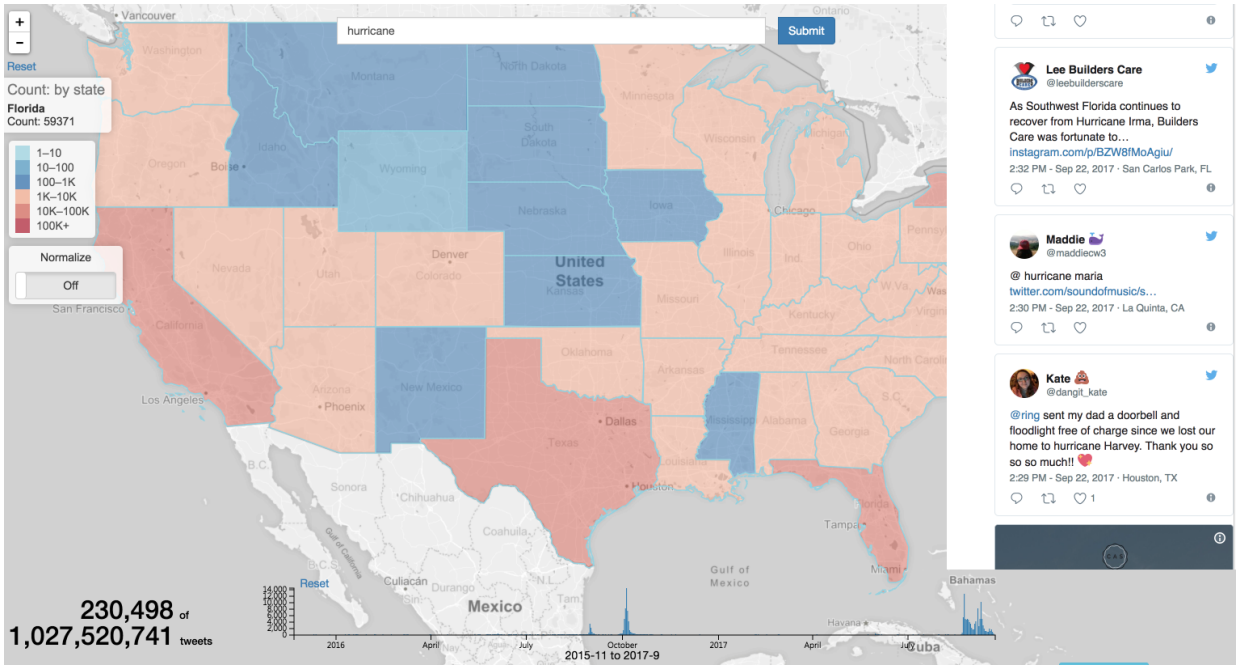


Figure 1: TwitterMap to support interactive exploration and visualization on more than 1 billion tweets.

technique. In addition, since the resources in a Web browser can be limited, besides storing the objects, we also need cache replacement to keep track of the resource usage. Our technique includes a replacement strategy that can remove irrelevant objects to reduce the memory consumption in the browser. Furthermore, it also includes a prefetching strategy that preloads potentially useful objects by examining the previous activities in the browser. Different from the existing prefetching strategies [9] that use complex probability models on the server side, our approach is lightweight, hence is more suitable for Web browsers.

2. CACHING GEOSPATIAL OBJECTS

In this section, we describe the proposed technique, including a system architecture where it can be used, and cache replacement and prefetching. We use the TwitterMap application as an example to illustrate this technique.

2.1 System Architecture

Figure 2 shows the architecture of a map system that can adopt the proposed technique. There are three main modules in the architecture: a cache module, a map connector, and a backend server. The cache module consists of a manager and a store. The cache manager is responsible for handling requests from the map connector. It also conducts replacement and prefetching activities. The cache store keeps cached geospatial polygons in an R-tree as well as the user's recently requested regions. It also keeps the entire cached region as one polygon. The purpose of keeping this region is to determine whether a new query region is a cache hit or miss. The map connector is responsible for presenting the map data along with geospatial polygons on the map. The backend module, which is a database or a web server, provides geospatial polygon data.

In TwitterMap, the cache module along with the map connector is written using Angular JS [4], and they both reside in the user's Web browser. The map connector uses the OpenStreetMap API [2] to fetch image tiles. Without the cache module, the map connector module always needs to communicate with the backend directly to

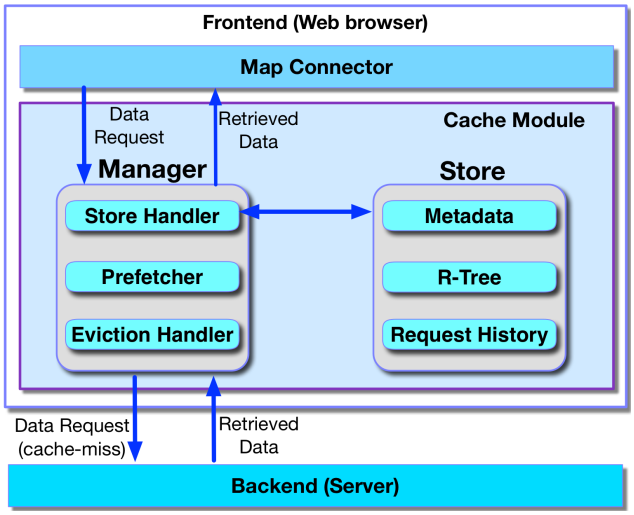


Figure 2: System architecture

retrieve geospatial polygons whenever it receives a user request. If the cache module is present, as we see in Figure 2, all of the map region requests now go through the cache module. For conducting geospatial operations such as intersection to find whether the requested region is contained in the cache store, we use the TURF library [3]. We define a cache hit when the requested region is in the cache store. In this case, the cache module returns the geospatial polygons stored in that region to the map connector module. If the region is not present, the cache module sends a request to the backend to fetch the geospatial polygons in that region that are not cached. It first stores the geospatial polygons in the region to the in-memory R-tree in its store. When sending a request to the backend, the cache module prefetches some additional regions to the current region from the map connector to increase the possibility

of a cache hit in the future. We will discuss this prefetching policy in detail.

2.2 Indexing Using R-tree

To store and access geospatial polygons, we use an in-memory R-tree, implemented as a Javascript library called RBush [11]. We store the geoJSON [1] format of each polygon in the R-tree by using the minimum bounding rectangle (MBR) of each polygon as its key and geoJSON-formatted data as its value so that we can retrieve the original polygon data, not just its MBR. Figure 3 shows a part of a city's geoJSON data.

```
{
 "type":"FeatureCollection",
 "features":[
   {"type":"Feature","geometry":{
    "type":"Polygon", "coordinates":
       [[[-73.868917,40.69515], ......
   ,[-73.868917,40.69515]]]},
    "properties":{
      "stateID":36, "stateName":"New York",
      "countyID":36047, "countyName":"Kings",
      "cityID":36047, "name":"Brooklyn" }
   }]
}
```

Figure 3: A part of sample city's geoJSON data

2.3 Cache Hit or Cache Miss

The cache store always keeps track of the entire shape of the cached region as a polygon after each operation. When the user's requested region (represented as a rectangle) is passed to the cache module, the module computes the spatial difference between them. If the requested region is contained in the cached region, we have a cache hit. In this case, the cache module provides the geospatial polygons in the R-tree. Otherwise, it is a cache miss, and the cache module inserts geospatial polygons into the R-tree after fetching them from the backend. Cache replacement, if needed, happens before geospatial polygons are inserted to the R-tree.

2.4 Cache Replacement

In case of a cache miss, the cache module fetches geospatial polygons from the backend, and tries to save them to its store first before returning them to the map connector. When this attempt fails due to limited space in the store, some of the cached regions need to be evicted. Figure 4 shows an eviction case when there is a cache miss.

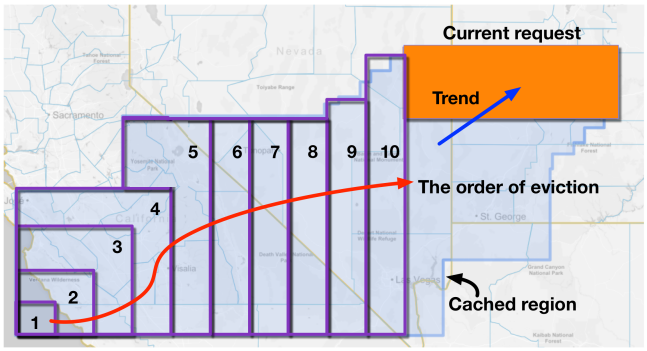


Figure 4: Cache replacement

In this case, the module first identifies the direction of the user's current request based on previous requests, i.e., where the user is heading on the map. Using the identified direction, it finds the farthest region in the cached store. It evenly divides the distance between that region and the current region into a certain number of parts. For example, we can divide the distance into a number of chunks, e.g., 10. The module starts evicting the farthest chunk first, based on the user's request trend since we deal with the user's current request on a geospatial object. We assume the spatial information is relatively more important than the temporal information. That is, rather than considering the time that a geospatial polygon was added to the cache store, we put the priority on the trend of the request and the distance between the current request and the trajectory of the user's recent requests (instead of using LRU). For example, the oldest region in the cache store can be the nearest region to the current request. Then, we do not want to evict that region. Also, we do not simply evict the farthest region from the current request region since the user's trend history plays a role here. For example, if the current request is heading towards the farthest region in the cache store, it is better to keep that region since it might be requested soon. Therefore, we consider the trend of the current request to decide the direction of user's request history and then remove the farthest region based on that trend. If there is still not enough space after an eviction, it evicts the second farthest chunk. This process continues until it consumes all chunks or it gets enough space to store geospatial polygons in the current request. If it consumes all chunks and still does not find enough space, then all of the cache regions will be cleared.

2.5 Cache Prefetching

To increase the possibility of cache hit, we do prefetching whenever there is a cache miss due to two reasons. First, we fetch additional regions the user may choose in the next request based on the trajectory of their recent requests. Second, fetching these regions does not significantly increase the communication cost, since the cache module needs to talk to the backend anyway to fetch the geospatial polygons of the current region. The additional regions are calculated based on the trend of recent requests. This trend can be calculated in many ways. For instance, as shown in Figure 5, we can calculate the trend based on the previous request and the current request. Specifically, we find the centroids of two requests and form a line by connecting them. After that, we compute the slope of the line and extend the line beyond the edge of the current request for a certain distance (e.g., 25 miles). The yellow region in the figure depicts the prefetched region for this case. For example, if we know that the current user is moving towards to northeast, we fetch more regions in that direction.

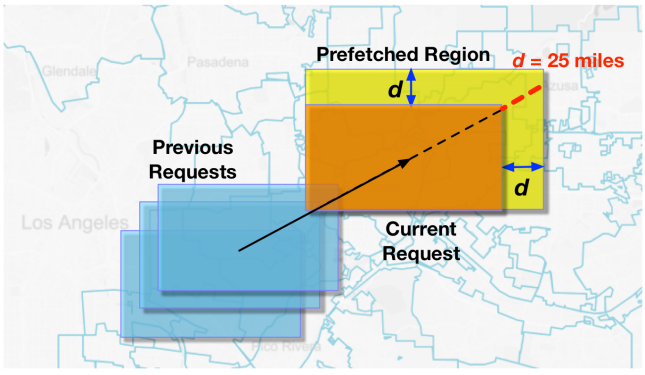


Figure 5: Prefetching

3. DEMONSTRATION

In the demo, we will show how the technique works for various user requests on the TwitterMap application. Normally, Twittermap does not show any of cache-related activities such as cache miss, hit, or prefetching. For demonstration purposes, we modify the frontend to display such activities using certain colors.

3.1 Cache Hit

Figure 6 shows the TwitterMap interface after several requests have been made from a single user on the map. The cached regions are displayed in the blue color. It shows a case where the current request is fetching the already cached region, which is a cache hit. We visualize the given cache-hit case using the green color. We also visualize the city polygons in the requested region in the black color for illustration purposes. Normally, what happens underneath is that the cache module returns the stored city polygons to the map connector without communicating with the backend. This is the reason why there is no prefetching in a cache-hit case, since we do not want to incur communication cost just to execute a prefetch operation.

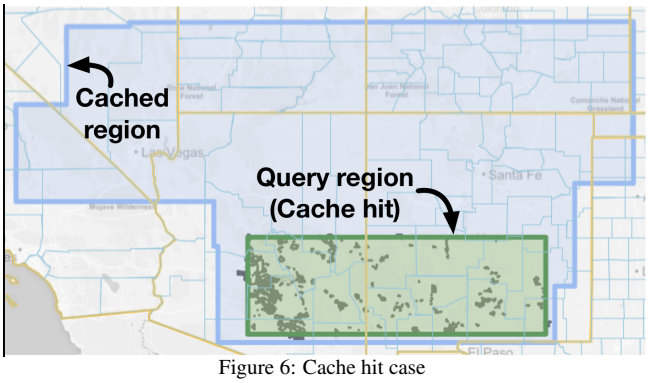


Figure 6: Cache hit case

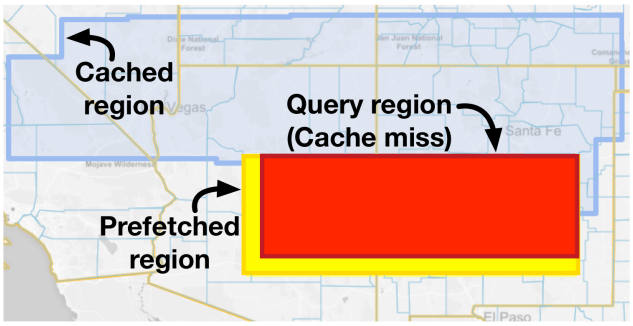
3.2 Cache Miss and Prefetching

Figure 7 shows two cache-miss cases. Figure 7a shows a cache miss without an eviction and Figure 7b shows a cache miss with an eviction. The request (in red) in Figure 7a was sent to the system and the request in Figure 7b was sent to the system right after that.

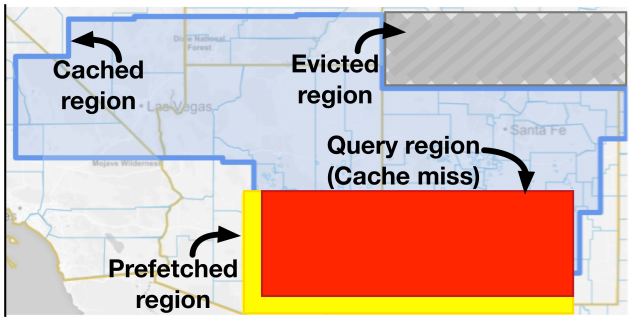
As described earlier, when there is a cache miss, the cache module fetches city polygons in the requested region from the backend. It also prefetches some additional regions (in yellow) to prepare for the future requests from the user. From both figures, we can see that the trend angle is towards southwest. Therefore, we can see that the prefetched region is placed around the boundary of that side in the current request region. Figure 7b also illustrates an eviction case. We can see that the some of the northeast region (in the gray box) is evicted from the cache store to accommodate the city polygons for the current request.

In the demonstration, we will show the behavior of the technique in terms of cache replacement, prefetching, cache hit, and cache miss. We will illustrate how the technique can reduce query response time. We will also show the effect of changing those parameters, such as cache size, number of recent requests to decide the prefetching direction, and size of prefetching region. These features will be illustrated by both user interactions as well as automated simulation using an animation script.

Acknowledgements This work has been supported by NIH award 1U01HG008488-01 and NSF CNS award 1305430. We thank the Cloudberry research team at UC Irvine for many technical and fruitful discussions.



(a) A cache miss without an eviction



(b) The next request - a cache miss with an eviction

Figure 7: Two cache-miss cases

4. REFERENCES

[1] Geojson. https://geojson.org/.
[2] Openstreetmap. https://www.openstreetmap.org/.
[3] Turf library. https://turfjs.org/.
[4] Angular JS, https://angularjs.org.
[5] J. Jia, C. Li, X. Zhang, C. Li, M. J. Carey, and S. Su. Towards interactive analytics and visualization on one billion tweets. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, pages 85:1–85:4, 2016.
[6] Mapd demo. https://www.mapd.com/demos/taxis.
[7] J. Mickens. Silo: Exploiting javascript and DOM storage for faster page loads. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010.
[8] M. D. Network. Dom storage. 23, 2011. https://developer.mozilla.org/en/DOM/Storage.
[9] D.-J. Park and H.-J. Kim. Prefetch policies for large objects in a web-enabled gis application. *Data & Knowledge Engineering*, 37(1):65–84, 2001.
[10] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.
[11] RBush Javascript Library, https://github.com/mourner/rbush.
[12] Tableau demo. https://public.tableau.com/en-us/s/gallery/history-us.
[13] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *NLANR Web Cache Workshop*, volume 97, 1997.