GloBiMaps – A Probabilistic Data Structure for In-Memory Processing of Global Raster Datasets

Martin Werner
martin.werner@unibw.de
Institute for Applied Computer Science, Bundeswehr University Munich

ABSTRACT

In the last decade, more and more spatial data has been acquired on a global scale due to satellite missions, social media, and coordinated governmental activities. This observational data suffers from huge storage footprints and makes global analysis challenging. Therefore, many information products have been designed in which observations are turned into global maps showing features such as land cover or land use, often with only a few discrete values and sparse spatial coverage like only within cities.

Traditional coding of such data as a raster image becomes challenging due to the sizes of the datasets and spatially non-local access patterns, for example, when labeling social media streams.

This paper proposes GloBiMap, a randomized data structure, based on Bloom filters, for modeling low-cardinality sparse raster images of excessive sizes in a configurable amount of memory with pure random access operations avoiding costly intermediate decompression. In addition, the data structure is designed to correct the inevitable errors of the randomized layer in order to have a fully exact representation.

We show the feasibility of the approach on several real-world data sets including the Global Urban Footprint in which each pixel denotes whether a particular location contains a building at a resolution of roughly 10cm globally as well as on a global Twitter sample of more than 220 million precisely geolocated tweets.

CCS CONCEPTS

• **Information systems → Point lookups**; **Data compression**; **Geographic information systems**; • **Computing methodologies → Image representations**.

KEYWORDS

Image Representation, Data Sparsity and Compression, Randomized Data Structures, Geographic Information Systems

ACM Reference Format:
Martin Werner. 2019. GloBiMaps – A Probabilistic Data Structure for In-Memory Processing of Global Raster Datasets. In *27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19), November 5–8, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3347146.3359086

SIGSPATIAL '19, November 5–8, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359086

1 INTRODUCTION

Managing big geospatial data has been a major research topic for the spatial computing community in the last decades. Larger and larger data sets lead to more and more distributed and parallel computing and special approaches are needed for spatial data as the parallelization of typical spatial algorithms is far from trivial.

At the same time, Earth observation has become a highly mature field in which regional and global data sets are acquired, combined, and published every day. For example, the NASA night light imagery provides a regularly updated raster image covering the whole Earth representing the average light emission at night time. This beautiful imagery is, of course, correlated to the human activities on Earth, though not in an absolutely obvious way. The Copernicus program of the European research agencies is publishing all satellite data from Sentinel missions to the public and the United States provide access to LandSat imagery. These globally available data sources can be used to create global datasets at incredibly high spatial, temporal, and thematic resolution. In addition, commercial satellites map the Earth surface with unprecedented resolutions of few centimeters and platforms such as Google Earth, Google Maps, ESRI maps, and HERE provide many ready-to-use global raster products derived from the available data.

Furthermore, social networks have grown almost global: services like Facebook, Twitter, Flickr, and Instagram have become the leading social network platforms in many countries of the world. These networks provide multimedia information consisting of location, time, text, and imagery. These sources of information are important as they are very timely and augment the birds-eye perspective of remote sensing with a ground-level view. The combination of social media and remote sensing is envisioned to become a major driver of land-use estimation which is essential to many applications. For example, the United Nations raise attention to a set of Sustainable Development Goals many of which are related to how land is being used and where people live [9]. In this context, this paper proposes a data structure GloBiMap for sparse binary rasters with the following key properties:

- The data structure provides constant-time spatially random access to individual pixels
- The data structure provides stateless access, that is, no temporary memory is needed to access the information
- The data structure provides an efficient compression method halving its size at the cost of additional error probability
- The data structure provides incremental low-resolution sketches of the underlying data
- The data structure is fully configurable in a triple tradeoff between error probability, memory usage, and computational effort

- The error type is controlled and an exact version with small additional memory overhead is provided

The remainder of this paper is structured as follows: the next Section 2 presents related work, Section 3 presents the data structure and associated algorithms, Section 4 introduces datasets for evaluation, Section 5 fixes evaluation measures for this study, Section 6 evaluates the system on real-world global data sets, and Section 7 concludes the paper and discusses future work.

2 RELATED WORK

In this section, we recall three basic concepts related to how sparse data sets can be represented. In the first section, we recall some background on matrix representations for sparse linear algebra. These are very efficient, but as they are usually optimized for linear algebra access using atomic units like rows, columns or rectangular slices, they don't perform well for fully random access to single locations. Then, we recall some background from image file formats, which basically share the same structure of cutting large images into scanlines, groups of scanlines, or rectangular slices following the idea that nearby spatial points will usually be consumed together. Similarly, random access is inefficient as full atomic units need to be decompressed and provided in dense representation using main memory. Finally, we present Bloom filters, which have been invented to represent very sparse subsets of strings, namely English words with irregular spelling. Though this data structure is widely applied in big data systems, to the best of our knowledge, it has not been applied to sparse raster representation which we propose in this paper. However, it has been applied to trajectories [14] as well as for representing small sets of locations in web browsers [11].

2.1 Sparse Binary Matrices

A sparse matrix is a matrix in which most of the values are zero. In scientific computing, this sparsity is usually related to the break-even of storing a matrix in dense form as an array of numbers in contrast to storing only the values of the matrix that are non-zero. That is, a sparse matrix in computation is a matrix that is represented by representing the set of non-zero entries organizing tuples (i, j, v) where i and j represent the row and column and v is the value. Sparse matrices can be identified with point clouds of integer coordinates and all aspects of spatial indexing can be applied in order to speed up access to entries of sparse matrices. However, sparse matrices are usually represented either in a naive form in which the tuples are ordered and organized as a dictionary or a list of lists or in a form optimized for linear algebra including compressed sparse row (CSR) and compressed sparse column (CSC) [2]. All of these data structures for sparse matrices, however, store two integer indices in addition to the data. In CSR and CSC, one of these integer indices can be compressed to store only the beginning of each line, yet the second index column is as large as the number of non-zero elements. Therefore, a sparse matrix typically uses at least $2n + m$ memory cells, where n is the number of non-zero entries and m is the number of rows (in CSR, resp. columns in CSC).

2.2 Image File Formats

Image compression is a rather classic topic and has been widely discussed in the computer graphics and signal processing domains. However, the presented image formats are often based on assumptions that are invalid for global raster data sets. Some of these file formats expect that a file is used completely and is decoded as a whole before using. This is, for example, true for portable network graphics and JPEG images. In practice, most implementations (e.g., libpng and libjpeg) will load the whole image into main memory and make it accessible as a dense matrix. Still, these formats are widely applied in the context of global raster data sets, for example, for disseminating spatial data sets to users over the Internet. A web map service based on tiles of small 256x256 pixel images is still the de-facto standard for providing huge raster graphics on the web. Another image file format, the tagged image file format (TIFF), is optimized for large data sets and allows for decoding only parts of the image [7]. Therefore, the image is split into strips (e.g., multiple consecutive scanlines) or tiles (e.g., rectangular pieces of data) which can be read individually. This architecture does not allow for true random lookups, because each access needs to load the whole strip or tile where the data resides in. But, the TIFF image file format is widely used in Geographic Information Systems (GIS) as there is a standardized extension known as GeoTIFF [10], where the header of the image file contains information about the spatial projection of the image data.

In summary, we conclude that storing a global raster data set as a set of image files is a valid strategy for most application scenarios, but it incurs large overhead if pixelwise random access happens often, because a large environment of the pixel is decoded into main memory before answering the query for this pixel.

2.3 Bloom Filter for Set Representation

In 1970, Bloom introduced a probabilistic data structure nowadays called the Bloom filter which is capable of representing sets in main memory [3, 4]. It provides the freedom to choose the amount of memory used to model the set trading of memory consumption and computational effort with error probability. The most important property of bloom filters is, however, that they have only one type of errors: false positives. While the data structure representing a set does never report an element not to be in the set that has been added before, it might report elements that have not been added to the set. The basic construction of the Bloom filter is as follows: first, let F denote a bit array of size m. Then, choose k pairwise independent hash functions h_i mapping all possible elements of the set to a non-negative integer number smaller than m. The empty set is represented by an all-zero Bloom filter array F. The Bloom filter now exposes its functionality through two operations Insert and Test. Given an element e, the function Insert modifies the Bloom filter to represent the fact that the element has been inserted. In the same context, the function Test reports whether the data structure indicates that the element has been inserted before, given the limitation of false positive results. Both operations start by first calculating the value of the chosen k hash functions applied to the given element e. The Insert function sets all the slots identified by the outcomes of the hashing to one while the Test function reports the element not to be in the set if and only if one of these outcomes points to a zero slot. It is obvious that this data structure cannot have false negatives, as the Insert operation sets the very same bits to one that the Test function is searching for a zero. On the other

hand, when all the slots indexed by the hash functions are filled with ones, this means that we can believe that the element has been inserted though the values might have come from a combination of inserting different elements due to possible hash collisions.

For a data structure with errors, it is most interesting, how these errors can be controlled and for the case of the Bloom filter this is quite simple. In fact, one can view a Bloom filter with n elements as the result of setting an essentially random bit in the underlying array to one for kn times as the hash functions turn the elements into uniformly distributed random numbers. Now, finding a false positive is as probable as it is to find k ones in random slots of the filter, which is as probable as finding a one k times in a random slot. In fact[1], the following formula relates the parameters m, k, and the number of elements n that have been inserted with a measure of the fraction of zeros (foz).

$$\text{foz} = \left(1 - \frac{1}{m}\right)^{kn} \approx \exp\left(\frac{-kn}{m}\right)$$

Given that the hash functions are uniformly random and pairwise independent, this fraction of zero is a good proxy for the probability of finding a zero. Therefore, the false positive probability of the filter can be derived as follows: a false positive happens, if k random accesses to the bit field find a one, e.g., in

$$p = (1 - \text{foz})^k \approx \left(1 - \exp\left(\frac{-kn}{m}\right)\right)^k \tag{1}$$

cases. Given that the right hand side is differentiable, we can search for a minimum of k by differentiating this expression and finding the roots. In this way, one can show with straightforward calculations that

$$k_* = \frac{m}{n} \log 2$$

is a global minimum of the (expected) false positive rate. That is, given a memory budget of m bits and a number of elements n to be inserted into the filter, we can choose an optimal value k_*. However, note that k is constrained to be an integer number of hash functions, therefore, we will need to round k_* to a nearby integer.

3 GLOBAL BINARY MAPS (GLOBIMAPS)

In this section, we introduce the data structure GloBiMap in which a Bloom filter is used to store the set of ones in a global binary bitmap. In addition, the data structure has an optional error correction table for exact representations.

3.1 GloBiMaps Construction

The central part of a GloBiMap is a Bloom filter of chosen size m. It is mainly used to represent the raster cells that are occupied. The rationale behind using a Bloom filter is that in many practical Earth observation scenarios, the number of interesting cells is very small compared to the number of measurement cells. Fixing a WGS84 global raster (e.g., representing the surface of the Earth in WGS84 coordinates and discretization the coordinate space in latitude and longitude direction forming a raster), only 32.5% of the pixels of a WGS84 water mask are not covered by water and only a small fraction of about 0.13% is actually human built-up area. Therefore, a lot of information related to social and socio-economic factors is taking place in a very small fraction of the Earth surface allowing for a better representation as compared to a dense raster that is widely applied in remote sensing.

In order to construct the GloBiMap, we first convert each pixel into a binary value and insert the pixels with the value true by hashing two 64-bit coordinates (that is an array of 16 bytes) using the Murmur3 hash function [1]. This hash function is known for its excellent tradeoff between quality of distribution and performance.

In order to reduce the number of hash computations, we will use the following hashing trick with which we can generate sufficiently independent hash functions for a Bloom filter from just two actual hash function calculations [6]. As the Murmur3 hash returns a hash of 128 bit, we take the first 64 bits and the second 64 bits for the two hash functions in the following formula:

$$h_i(x) = h_{\text{low}}(x) + i \cdot h_{\text{high}}(x) \bmod 2^m, i \in \{1 \dots k\}$$

This allows us to use a single computation of Murmur hash to create a family of k hash functions based on a linear congruential generator. Furthermore, note that a modulo of 2^m can be computed without a division as it is equal to computing a binary AND with $2^m - 1$. This can greatly increase performance yet limits us to the use of power of two sizes for GloBiMap storage.

In addition, this puts a penalty on GloBiMaps larger than 4GB: computational effort will grow as one needs more than 32 bits for addressing the hash table leading to operations in the linear congruential generation exceeding the typical 64bit address space of a CPU. Instead, the program needs to split the numbers each into a low and high word and perform complicated and time-consuming partial operations for the hashing trick.

3.2 Multi-Layer GloBiMaps

As the construction and analysis of the GloBiMap data structure depends only on the fraction of zeros, we can actually store more than one layer of a GlobiMap into the same hash table. Therefore, one would prefix the given array of 16 bytes representing coordinates with any string leading to additional independent hash results. The error analysis keeps intact, one only needs to consider that the fraction of zeros governing the probability of finding a one for a query now stems from knl times trying to set a slot to one where l is the number of independent layers. Note that l layers allow for modelling 2^l values though false positives would now imply false classifications in which the estimated value at a location is bit-wise larger than the truth. Therefore, one should order the classes in increasing importance.

3.3 Low-Resolution Incremental Rendering

The data structure has been designed to store very high resolution imagery, because they are more relevant to applications and increasing resolutions often increase sparsity as well. It is, however, possible as well to use the GloBiMap data structure for statistical incremental low-resolution rendering.

Exploiting the fact that the distribution of false positives is uniform, we can exploit the fact that many pixels of a high-resolution raster overlap with a pixel in a low-resolution sketch. One gets

[1] with neglectible and correctable technical errors related to statistical dependencies

a first impression by testing one high-resolution pixel per low-resolution pixel. Then, one can incrementally refine the result by testing more and more different pixels overlapping into the area of a low-resolution pixel. For visualization, one would use the mean value of true and false to depict a summary of the underlying high-resolution pixels.

The false positives inherent to the system would lead to the fact that a known number of the tests (given by the false positive rate) will return a true result though a false would have been appropriate. This means that the mean for each pixel is too large by the false positive rate times the number of pixels contained. Therefore, we can subtract this constant bias from all pixels during the mean calculation. For very low false positive rates, however, we can as well ignore this bias if it remains much smaller than the expected error of the sample mean or the color accuracy of the resulting display image.

3.4 Alternating Coding

In order to improve the speed of rendering low-resolution sketches, we can as well directly model the low-resolution sketches in the probabilistic layer. This can be done by first storing a low-resolution binary version of the dataset in the GloBiMap. Now, similar to a quad-tree, we subdivide the low-resolution pixel and add all pixels to the next zoom level that differ from the current zoom level. Concretely, we name each pixel not only with its coordinates, but as well with its zoom level starting with zero. That is, a pixel "0_42_11" is the pixel in the 42nd row and 11th column of zoom level zero. For zoom level one, each pixel is now subdivided into a number of pixels. Some of these pixels will differ from the value that has been modeled for the first zoom level. We store exactly those into the first layer. That is, if a pixel is zero and the pixel in the previous zoom level is one or if a pixel is one and the pixel in the previous zoom level is zero, we add this pixel to the current layer.

With this data structure in place, we query for a pixel on a particular zoom level by iteratively querying through all zoom levels flipping the value between 0 and 1 each time we see a point. This leads to a significantly quicker rasterization of low-resolution sketches at the expense of a more involved pixel-wise rasterization operation querying multiple zoom levels.

As a consequence, the false positives of the underlying Bloom filter act in both directions: they can flip a one to zero and a zero to one. However, the succeeding layer can flip the value back as well. Furthermore, the error correction scheme introduced in the next section can as well deal with this alternating behavior.

3.5 Error Correction Table

Additionally, the data structure can be augmented by a lookup table modeling false positives. This, of course, reduces the amount of compression previously introduced by the Bloom filter construction. When applied with care it will create a flexible and fast compressed data representation. Furthermore, the error correction data can be restricted to areas of interest limiting the additional overhead introduced by this error-correction layer. This error correction construction can result in tiny tables in practice if the false positive rate is kept low enough. Then, an efficient table lookup (using, for example, hash tables or binary search on both coordinates) per pixel becomes feasible. Note that this error correction can be delayed such that it only occurs if the application really needs a refinement. In this way, a spatial organization of larger error correction data is still sensible, if error correction is applied only locally and comparably seldom.

To illustrate this fact, we give the following example: *When applying low-resolution incremental rendering, we can avoid error correction, as the error can be statistically corrected by subtracting the bias. As soon as the user zooms in, however, we switch to using error correction.*

In this example, error correction is applied in a spatially local setting only and, thus, spatial indices including grids and R-trees can be used to organize the error correction data and load it into memory only when needed and only for relevant regions.

In order to estimate the amount of storage needed for error correction, we rely on the analysis of the Bloom Filter as follows. First, we can use the analysis of the Bloom filter directly to understand the performance of the GloBiMap construction.

From Equation 1, we can calculate the probability of false positives as

$$p \approx \left(1 - \exp\left(\frac{-kn}{m}\right)\right)^k$$

Note that we can reduce the number of free parameters from three to two using the relation for an optimal k_*

$$k_* = \frac{m}{n} \log 2$$

Assuming that the chosen hash functions (and the actual implementation of both the hash function and the hashing trick) behave really well, we can conclude some aspects for GloBiMaps. Given a binary raster of width w and height h, we first need to compute the number n of pixels with value true. The value m is chosen as a power of two in order to avoid costly modulo operations though this can be relaxed at the expense of higher computational overheads.

Then, the expected number of false positives can be computed and the number of wrong pixels is going to be about

$$E = (wh - n) \cdot p$$

With respect to error correction tables, this implies that we expect to store E pairs of coordinates.

3.6 Mod-2 Compressions

If the size of the underlying filter is a power of two, we can build a pyramid of Bloom filters by iteratively cutting the Bloom hash field into two halves and combining these with a binary OR operation. The resulting hash field is identical to the hash field that would have been generated with the same set of hash functions but a value of $m/2$ due to the modulo operation being used to map integer hash values into the index range. Therefore, we can compress existing GloBiMaps quite efficiently to reduce memory usage without recomputing GloBiMaps from scratch. Furthermore, this allows for an iterative construction strategy in which we start with a huge GloBiMap and create various variants of lower accuracy by iteratively compressing the bit field. The whole set of GloBiMaps constructed in this way needs a storage of

$$\sum_{k=0}^{l} \frac{m}{2^k} = m\left(2 - \frac{1}{2^l}\right)$$

That is, given a GloBiMap we need to afford less than twice the storage to store all Mod-2 compressions allowing for access to the data structure with varying memory overhead, pixel access performance, and error rate.

3.7 GloBiMaps Construction Strategies

There are two major approaches to constructing GloBiMaps. The first approach exploits a known data set and its exact properties while the second approach exploits assumptions about the data that might be handled by a GloBiMap.

We can actually configure the parameters based on the expected amount of information. Taking into account that Bloom filters with less than the number of elements used in configuration have a smaller false positive rate and that Bloom filters can be compressed, it makes sense to configure based on very simple assumptions about sparsity.

If the data set is static, we can search for optimal values for the seed, the hash functions, m and k in an empirical setting to come up with the best set of parameters.

In contrast, one can configure a small Bloom filter and replace it (by recomputation) each time the data is growing beyond the current capacity of the filter. This approach is most widely used in database systems relying on Bloom filters, because the inability to delete information from Bloom filters and usually as well from compact disk files leads to a compaction operation that is routinely run over all data from time to time.

The third approach is to generate a huge Bloom filter, insert all data and take the Modulo-2 compression that best fits the current task. This approach is most efficient, when the system can gain from many different instances of varying size and false-positive rate.

3.8 GloBiMaps Interface

The chosen construction limits the way we can interact with the dataset represented by GloBiMaps. In fact, we only provide three operations to work with GloBiMaps: Get, Set, and Rasterize. The Get operation is given a coordinate (x,y) and checks whether the relevant element is in the Bloom filter. Optionally, it also checks whether the element is a false positive by looking up in the error correction table. For a multilayer GloBiMap, this can be iterated using the prefix and for alternating coding, this can be iterated for each layer. In general, this operation is quite quick involving two hash calculations, several multiplications and several memory lookups to check the bit array. Optionally, the Get operation can be performed against one of the optional compressions of the Bloom filter allowing for a smaller main memory consumption.

The Set operation can be used to set a coordinate in the GloBiMap to one. It is similarly efficient as the Get operations and updates all involved data structures such as the filter F, the error correction table and, possibly, available compressed filters P_i.

The Rasterize function is a convenience function in which the Get operation is applied over a rectangle and the rectangle is returned as a matrix. As it is a read-only operation, it is naturally parallelizable and, in fact, if GloBiMaps are implemented on a GPU it can bring down the execution time of Rasterize to the speed of a single Get times the number of pixels divided by the number of cores. For the CPU case, the rasterize operation can also be optimized with read pooling in which the data access to the global array is organized in a way such that the CPU cache is invalidated less often. Both optimizations, GPU implementation and CPU cache optimizations are, however, beyond the scope of this paper. Still, the interface reflects that a naïve implementation of Rasterize in terms of a loop over Get is far from optimal.

4 EVALUATION DATASETS

GloBiMaps represent a general data structure capable of representing raster images. In order to evaluate GloBiMaps, we measure several aspects characterizing the functionality and performance of the data structure in the context of a multitude of datasets of different nature.

4.1 Raster Datasets

Several datasets are needed in order to evaluate the GloBiMaps data structure. This section introduces the choice of raster datasets represented as GloBiMaps for the evaluation of the approach in this paper.

4.1.1 Computer Game Raster Maps. In order to evaluate GloBiMaps, we first conduct experiments on small datasets derived from computer game maps [13]. The dataset being used consists of 462 game maps each of which consists of a rectangular space in which passable and non-passable terrain is being distinguished, Figure 1(a) depicts some examples. These maps are rather small, but they contain many realistic spatial distributions. The fraction of walkable pixels ranges from 0.01 to 0.99 with a mean of 0.32 (variance 0.04).

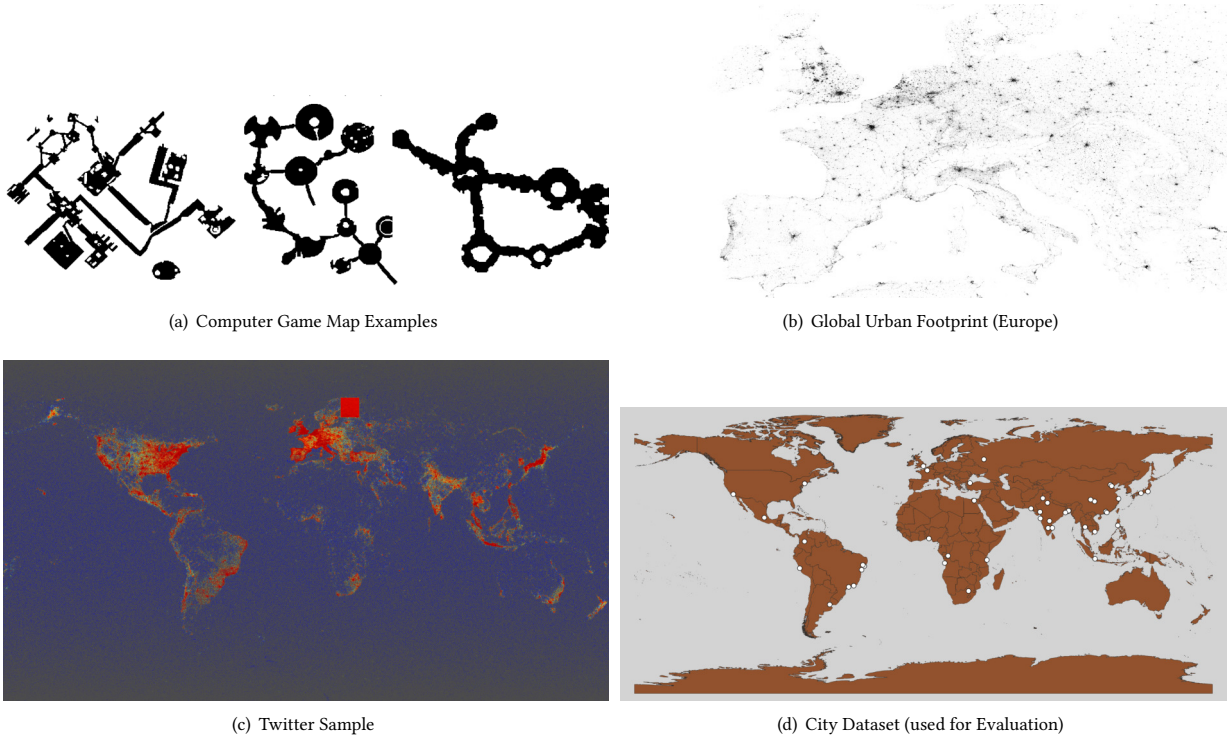
4.1.2 Global Urban Footprint. The Global Urban Footprint is a global raster product in which each place of the earth is assigned a binary label of whether it is human built-up (urban) area or not [5]. This dataset has been produced by using about 308 TB of data from the German radar satellites TerraSAR-X and Tandem-X. Figure 1(b) depicts an excerpt for Europe.

4.1.3 Twitter Occurence. We collected a large sample from the Twitter API including 220 million precisely geolocated tweets. As you can clearly see in Figure 1(c), the dataset is skewed towards the United States and some European countries. The major areas of China, Russia, and Africa except the coastal regions and South Africa did not come up significantly in this dataset. For China and Russia, this is related to the fact that Twitter is not used there.

4.2 Regions of Interest

Some strategies of building and optimizing the representation of a binary raster are based on regions of interest. While we are not using regions of interests for the very small computer game datasets, we employ them for global datasets. We used a set of challenge cities roughly related to the cities that the United Nations expect to have more than 10 million inhabitants in 2030[2] [8]. These form a natural,

[2]This list of cities consists of Tokyo, Delhi, Shanghai, Mumbai, Beijing, Dhaka, Karachi, Al-Qahirah, Lagos, Ciudad de Mexico, Sao Paulo, Kinshasa, Osaka, New York-Newark, Calcutta, Guangzhou, Chongqing, Buenos Aires, Manila, Istanbul, Bangalore, Tianjin,



(a) Computer Game Map Examples

(b) Global Urban Footprint (Europe)

(c) Twitter Sample

(d) City Dataset (used for Evaluation)

Figure 1: Dataset Overview

multi-cultural global challenge dataset for sparse raster datasets representing intra-urban characteristics. Figure 1(d) depicts the chosen city locations on a world map.

A number of four additional locations have been added to debug and include a few extremely sparse, non-urban regions as well.

5 EVALUATION

The GloBiMap data structure has been proposed to avoid costly disk lookups and paging architectures when working with global sparse classification data sets. In general, two orthogonal aspects of quality need to be discussed as this data structure consists of a lossy encoding with an optional error correction scheme. The first aspect "Representation Quality" discusses how well the data structure is able to model the data and the second aspect "Representation Efficiency" discusses how computationally efficient this data structure turns out to be in practice for both operations of encoding and decoding data sets.

5.1 Representation Quality

The representation quality of a lossy binary raster image data structure is commonly analyzed in terms of error rates and error distributions. The basic such measure is given by the bit error rate that is the probability of observing an erroneous bit in a uniform random location. This can be roughly identified with the fraction

$$\text{BER} = \frac{\#\text{ Erroneous Pixels}}{\#\text{Total Pixels}}.$$

As GloBiMaps are guaranteed to have no false negatives, this measure can be refined taking into account how difficult it is for such a data structure to represent the data. As negative pixels cannot induce errors, we normalize against all positive pixels only. That is the negative bit error rate NBER is defined to be

$$\text{NBER} = \frac{\#\text{ Erroneous Pixels}}{\#\text{Negative Pixels}}.$$

This measure ranges from zero to one, while the BER depends on the sparsity of the dataset.

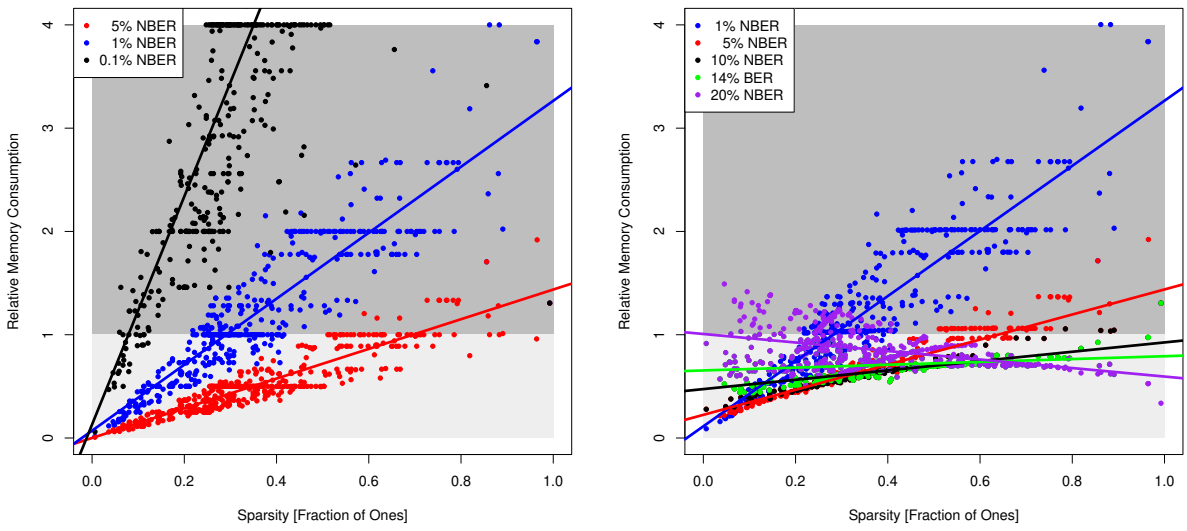
With respect to spatial distribution of errors, we note that the construction of a single global GloBiMap without any tiling or spatial subdivision implies a nearly uniform spatial error distribution given that the errors stem from uniformly distributed hash collisions.

It would, however, be an interesting question for future research, whether one can come up with a family of hashing functions that introduce a certain sensible spatial autocorrelation of false positives such that false positives are more likely near true positives [12].

5.2 Representation Efficiency

For representation efficiency, we have to discuss theoretic measures independent from the compute architecture and as well comment on some real-world effects that have a non-neglectible effect on

Rio de Janeiro, Chennai Madras, Jakarta, Los Angeles, Lahore, Hyderabad, Shenzhen, Lima, Moscow, Bogota, Paris, Johannesburg, Bangkok, London, Dar es Salaam, Ahmadabad, Luanda, Ho Chi Minh, and Chengdu.



(a) Without Error Correction

(b) With Error Correction

Figure 2: Representation Efficiency (e.g., the main memory consumed by an uncompressed in-memory raster divided by the GloBiMap memory consumption) in relation to Sparsity on the Computer Game Map Dataset

the practicability of the proposed approach. In summary, we record the memory consumption, the fraction of zeros of the filter, and the size of the correction map as measures for the computational demand of an instantiated measure. Note that the fraction of zeros is actually a measure of information, because the Shannon entropy of the filter can be directly computed given this fraction as f:

$$H(F) = -\left(f \log_2 f + (1-f)\log_2(1-f)\right)$$

In order to account for real-world aspects, we measure as well the performance of the encoding and decoding operations rasterizing certain urban areas across the globe using a parallel implementation on a typical desktop CPU.

6 EXPERIMENTS

This section presents experiments evaluating the GloBiMap data structure including a discussion of sparsity, compression efficiency, and real-world performance on a representative set of tasks.

6.1 Representation Efficiency of GloBiMaps in Relation to Data Sparsity

We discuss the question of how much sparsity is actually needed to successfully apply the GloBiMap data structure on the computer game dataset. The computer game datasets serve as a real-world set of spatially sensible rasters with varying sparsity.

For this discussion, we use the memory consumption of a GloBiMap with and without error correction as a single indicator of performance and compare it to the traditional representation using an in-memory raster.

Therefore, we build GloBiMaps of increasing sizes until they hit several threshold values τ for the NBER. All other parameters stay fixed. As growing the data structure reduces the NBER, this process provides the smallest GloBiMap fulfilling the threshold τ using the given other parameters. Note that we do not apply the error correction scheme in this experiment.

Figure 2 depicts the result of this analysis for $k = 2$ hash functions. The X-axis depicts a measure of sparsity given as the fraction of ones in the input raster. In this measure, maximal sparsity relates to small values on the X-axis. The Y-axis divides the memory used for an error-corrected GloBiMap divided by the space needed for representing the raster in a traditional in-memory matrix. The admissible area where GloBiMaps have a smaller memory footprint is shaded light gray.

Essentially, this figure illustrates the linear relationship between NBER and sparsity. For a fixed NBER (e.g., color), the memory consumption is a roughly linear function of the sparsity. Moreover, significant amounts of sparsity (that is very small X-values) are needed in order to encode datasets with small NBER. For the computer game dataset, it has not been possible to encode the majority of the maps with a NBER of less than 0.1%. This translates to the fact that this dataset is not sparse enough for GloBiMaps.

There are some non-intuitive clusters of points with the same Y coordinate. This is a coincidence of powers of two being used: 142 of the 462 computer game maps have a size of 512x512 pixels and, therefore, a power-of-two memory consumption in the naïve representation. As we are growing GloBiMaps with power-of-two sizes, these horizontal clusters appear.

A second experiment involves the use of the error correcting mechanism. For this, we extend the previous experiment and add error correction after encoding. Thereby, the size of the data structures grows depending on the error rate of the encoding. The NBER depicted in the figure represents the NBER used to configure the filter size. The resulting data structures are all error-free due to applying the error correction scheme. In other words, the given NBER values steer between the performance of the probabilistic layer and the size of the lookup table for error correction. We give the total memory consumption including both layers assuming that 8 bytes per error are sufficient to represent the error coordinate. This limits individual coordinate axes to values between zero and 4,294,967,295, sufficiently large for many applications.

What this figure actually illustrates is the fact that for sparse datasets, the smallest NBER is not an ideal option. Instead, allowing more errors and correcting them is better in terms of memory consumption. For 10%, for example, this approach has chosen about half the size of the GloBiMap as opposed to 5% and nearly a fourth of the size of 1%. The error correction information often fits into this additional free memory. With error correction and a NBER of 10% for the probabilistic layer, the GloBiMap data structure turned out to be larger only in 5 of the 462 cases and in the worst case it was only 1.31 times larger than the classical representation. As the figure motivates, the configuration NBER can be used to gradually turn the lines from ascending to descending. There is a break-even at about 14% for which the trend of the relative memory fraction is nearly horizontal with sparsity. This can be seen as an ideal configuration independent of sparsity. In this case, only 6 out of the 462 computer game maps use more memory in the GloBiMap framework as opposed to the common raster memory representation and on average the GlobiMaps used 70% of the memory of the usual raster representation in this configuration setting.

6.2 Performance for the Global Urban Footprint

The data structure has been specifically designed to capture the sparsity of global spatial datasets related to anthropogenic effects, especially in binary or low cardinality discrete cases. One canonical dataset of this type is the Global Urban Footprint containing a one in each and every place where a classification system has identified built-up area.

6.2.1 Representation Efficiency. The dataset itself is provided in various resolutions, where the highest resolution dataset contains pixels of varying height and a slightly varying number of coordinates in the horizontal direction as well due to rounding errors from the tiling process. We combine these files into a raster by first creating a point cloud with pixel center coordinates and then rasterizing this point cloud to a uniform raster with a pixel size of 0.000111 degrees. This results in a uniform raster with about 3.24 million columns and 1.62 million rows for an overall raster size of about 5.25 trillion pixels (e.g., 5.25 terapixel). However, only 0.13% of these pixels represent urban regions. This amounts to a storage size of 611GB if each pixel is represented with a single bit. Using the theoretical analysis machinery, we configure variants given in Table 1 based on choosing various sizes m and then optimally choosing k minimizing the size of the optional error correction tables.

Name	Size	k	FOZ	FPs (FPR)	ECI
Small	8 GB	7	0.501	40 bn (0.76%)	307.3 GB
Medium	16 GB	14	0.501	309 mn (5.91e-05)	2.36 GB
Large	32 GB	28	0.501	18,278 (3.4e-9)	0.14 MB

Table 1: Real-World Performance of applying the GloBiMap scheme for the Global Urban Footprint Dataset (FP=false positive, FPR=false positive rate, FOZ=fraction of zeros, ECI=size of the error correction table)

As one can see from this table, good false positive rates are reached in all of the cases. However, the number of false positives for the whole image greatly differs. For the small encoding with only 8GB of main memory, the error correction table is clearly huge with more than 300GB and it cannot be used without spatial indexing, tiling, or selecting certain regions of interest only in which error correction information is stored. Therefore, we propose to use this representation only for a coarse filtering or visualization of high-resolution datasets in lower resolutions without even computing the error correction information. When doubling the size (medium) to 16 GB, the optimal number of hash functions doubles as well. This set of parameters results in a quite feasible pair of a 16GB probabilistic layer based on 14 hash functions together with less than 2.5GB error correction information. This can be easily held in main memory on PCs and servers for processing and provides an error free and high-performance representation of the global raster. If main memory is a smaller concern, the large variant provides almost error-free coding in a 32GB probabilistic layer. The expected error rate will be about 3.4 pixels per megapixel. With about 140 kB, the error correction information is even smaller than usual CPU cache and error correction will have a neglectible overhead as opposed to memory access. *In summary, we were able to construct two lossless encodings supporting random access with 18.5 GB total memory consumption and a little more than 32GB for a raster that would at least occupy 611GB of main memory if each pixel would be represented with a single bit.* Relaxing the limitation of using only power-of-two main memory sizes, we can even modulate between those two examples at the cost of slower modulo computation.

6.2.2 Wall-clock Performance of Encoding. The encoding using GloBiMaps in general incurs some additional overhead as opposed to holding the data in memory. However, these overheads are quite small if this allows avoiding costly disk accesses.

The following numbers were generated using a typical PC (Intel Core i7 6700K, four cores, eight threads, 8MB cache) without an SSD or RAID using a single 2 TB spinning disk for holding the data. For the hashing trick, we use 64 bit instructions always. Similarly, the MurmurHash3 implementation is using native 64 bit instructions all over the place and has been adopted from the reference implementation of Appleby [1].

One should expect that encoding speed and decoding speed differ a lot: for encoding, we need to compute all k hash functions. Furthermore, it is not easily possible to implement atomic bit setting on a bit-compressed array. For ease of implementation, we therefore

make the Set method use the Bloom filter memory exclusively limiting parallel efficiency. But as encoding efficiency is of minor interest, this is tolerable. In future work, we envision to work on an (almost) lock-free encoding scheme. For decoding, we can use asynchronous memory access as the values do not change anymore. Furthermore, we can stop the evaluation of a pixel at the first test that hits a zero.

For performance estimation, we used the high-resolution global urban footprint with its more than 6.78 billion non-zero pixels. The dataset is represented as a point cloud in WGS84 coordinates, that is, each pixel is represented as two 64 bit IEEE floating point numbers for sufficient accuracy. This amounts to an input file size of about 163 GB. During encoding, this file is read in chunks of 16.7 million entries (that is in blocks of 256 MB) and a parallel encoding is performed using the parameter settings given in Table 1.

In general, CPU hashing dominates performance leading to the expected behavior in encoding times: For the small setting with 7 hash functions on an 8GB hash field, the encoding process took 6,675 seconds (about 1.8 hours). This realizes a rate of about one million insert operations per second or about seven megahashes per second. Similarly, the medium setting employed 14 hash functions on a 16GB bit field and needed an encoding time of 12,155 seconds (about 3.37 hours). The marginal increase in hashing performance can be explained from the fact that a larger fraction of CPU time is used for hashing and, therefore, the probability of waiting for the synchronous write operation to the filter is slightly reduced. Finally, the large scenario (32GB, 28 hash functions) needed 27,049 seconds (about 7.51 hours) for encoding. This is again in the range of four times the smallest scenario for an insert rate of 0.2507 million inserts per second, roughly a fourth of the small scenario.

In general, these are tolerable times given that such datasets do not change often and that the compression is impressive. In practice, all three versions can be easily held in main memory leading to the ability to perform global random access to the data.

6.2.3 Information-theoretic Evaluation. It is known that the information embedded in a bit array is maximal if the bits form a random sequence. For Bloom filters, the filter will reach this state if it is optimally configured. Plugging in the optimal value of k_* leads to a fraction of zero of exactly 0.5 and the sequence is random as it is constructed using uniformly random coin flips.

Observing the performance of the GloBiMap probabilistic layer, we see fractions of zeros very near to the optimal point, but slightly larger: 0.501191 for the small and the medium scenario and 0.501192. This is a hint that both the Murmur hash and the hashing trick did a reasonable work in providing the Bloom filter with nearly uniform random information. With these numbers, we can actually expect that the theoretical analysis holds and that the expected false positive rates will be very near to the true false positive rate. For future work, one could investigate in how far crypto hardware such as the AES-NI or the RDRAND CPU instruction can improve over the Murmur3 hash this in terms of quality or performance. This, however, violates the platform independence of the current implementation.

6.2.4 Wall-Clock Performance of Decoding. In order to check the practical query performance, we rasterize a 2,000x2,000 pixel window around each of the selected 45 cities and measure wall-clock time. Note that this time includes the time needed to write the results into a file for later inspection. This workload amounts for a total rasterization of about 180 megapixels from all around the world.

For the small scenario, this process was finished in 45 seconds leading to an average performance of 4 megapixels per second. This time, the performance is bound by disk I/O. For the medium scenario, the production of the raster images took 54s, still a decent speed of about 3.33 megapixel per second. For the large scenario, the rendering took 76.7s, a performance of about 2.35 megapixels per second.

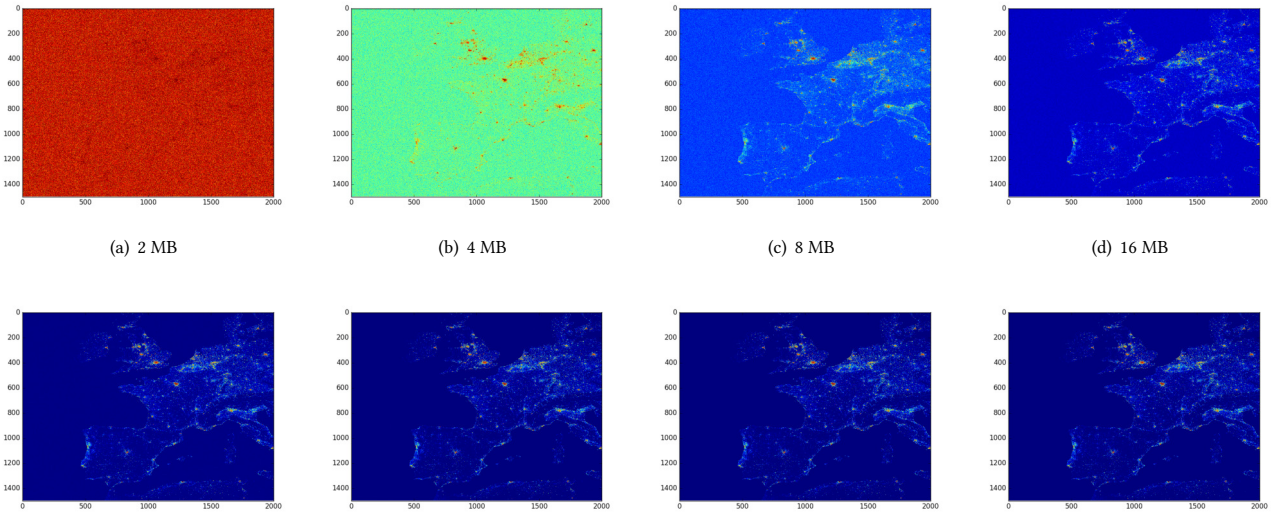
6.2.5 Qualitative Inspection over Europe. In the previous sections, we discussed the GloBiMap data structure in terms of an excessively high-resolution raster. In this section, we change to a lower-resolution version of the same dataset to show how GloBiMaps can compress image information into extremely small buckets of information, which are helpful in parallel processing, because many of those could be held in main memory in parallel or can be exploited in resource constrained contexts such as within a GPU or on a mobile device.

For this section, we use a low-resolution version of the Global Urban Footprint consisting of 192,857 rows and 462,859 columns for a total number of about 89.256 gigapixel. We show renderings of Europe taken from a global representation of the GUF created by averaging 20x20 input pixels into a single pixel value. We create modulo-2 compressions with sizes between 8MB and 512MB and depict the results of rendering without error correction in Figure 3.

These images clearly show that 2MB of memory are not enough for a reasonable representation of the GUF, while a size of 4MB already shows a non-uniform distribution around capitals. With 8MB, you see a pretty clear picture of the urban structure of Europe overlayed with some uniform noise while for 32MB it gets hard to spot the noise with bare eyes in this representation of averaging blocks of 20x20 pixels.

6.3 Results for other Datasets

Compressing the spatial substructure of Twitter social media data is an interesting operation as Twitter provides a vital resource of information for many applications and streams in ordered by time leading to random spatial access. We conducted a large data acquisition in which we collected a global sample of 220 million precisely geolocated tweets (that is tweets with specified spatial coordinates). We use the same grid as for the highly precise global urban footprint with a total size of 5.25 trillion pixels to represent even the finest spatial structures. Now, as many of the 200 million tweets fall into the same pixel, we are finally able to compress the occurrence of tweets in this very fine raster with only 64MB of main memory and 4.5MB of error correction data. The encoding took 490s (slightly more than 8 minutes). Note that we did not sort the 220 million tweets in order to remove duplicates. Instead, we just add different tweets that end up in the same pixel individually. This avoids a costly sort and unique operation on the dataset. Rendering the 45 test cities took a total of 45 seconds. A rendering from this representation around the city of Tokyo is depicted in Figure 4.



(a) 2 MB (b) 4 MB (c) 8 MB (d) 16 MB

(e) 32 MB (f) 64 MB (g) 128 MB (h) 256 MB

Figure 3: Various Renderings of the lower-resolution GUF over Europe using differently sized GloBiMap representations.

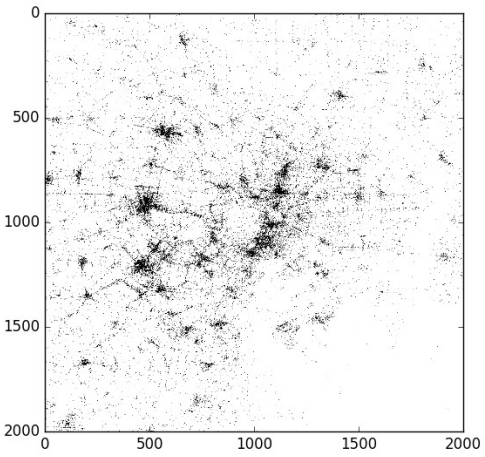


Figure 4: Twitter Stream around Tokio (GloBiMap of 64MB)

7 CONCLUSION

With this paper, we have shown that even highly precise global datasets can be held in main memory when exploiting the actual sparsity of most effects of interest. We show that pixelwise random access is possible leading to applications in spatial stream labeling, where a temporally ordered stream of information (e.g., the Twitter stream) is labeled from a sparse spatial knowledge source (e.g., land cover). Aside its efficient main memory representation and its unique support for pixelwise access, the data structure is as well computationally feasible as shown with pretty small encoding and decoding times on CPUs. For future work, we envision to work on the hashing scheme and to combine the methodology with more traditional spatial indexing techniques including embedding GloBiMaps into inner nodes of indexing trees. In addition, a discussion of the tradeoff between direct and alternating coding would be an interesting direction for further research.

REFERENCES

[1] Austin Appleby. 2008. Murmurhash 2.0.

[2] Randolph E Bank and Craig C Douglas. 1993. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics* 1, 1 (1993), 127–137.

[3] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[5] Thomas Esch, Mattia Marconcini, Andreas Felbier, Achim Roth, Wieke Heldens, Martin Huber, Max Schwinger, Hannes Taubenböck, Andreas Müller, and Stefan Dech. 2013. Urban footprint processor—Fully automated processing chain generating settlement masks from global data of the TanDEM-X mission. *IEEE Geoscience and Remote Sensing Letters* 10, 6 (2013), 1617–1621.

[6] Adam Kirsch and Michael Mitzenmacher. 2008. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms* 33, 2 (2008), 187–218.

[7] Samuel Leffler. 2003. LibTIFF–TIFF Library and Utilities. remotesensing.org/libtiff.

[8] The United Nations. 2016. The World's Cities in 2016 - Data Booklet. http://www.un.org/en/development/desa/population/publications/pdf/urbanization/the_worlds_cities_in_2016_data_booklet.pdf. Accessed 2017/11/30.

[9] United Nations. 2019. Sustainable Development Goals. Retrieved from https://www.un.org/sustainabledevelopment/.

[10] N Ritter and M Ruth. 1997. The GeoTiff data interchange standard for raster geographic images. *International Journal of Remote Sensing* 18, 7 (1997), 1637–1647.

[11] Peter Ruppel and Axel Küpper. 2014. Geocookie: a space-efficient representation of geographic location sets. *Journal of Information Processing* 22, 3 (2014), 418–424.

[12] Mirco Schönfeld and Martin Werner. 2013. Node Wake-Up via OVSF-Coded Bloom Filters in Wireless Sensor Networks. In *Proceedings of the 5th International Conference on Ad Hoc Networks (ADHOCNETS 2013)*. 119–134.

[13] N. Sturtevant. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* 4, 2 (2012), 144 – 148.

[14] Martin Werner. 2015. BACR: Set Similarities with Lower Bounds and Application to Spatial Trajectories. In *23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2015)*. ACM, 10.