AID*: A Spatial Index for Visual Exploration of Geo-spatial Data

Saheli Ghosh, *Member, IEEE*, Ahmed Eldawy, *Member, IEEE*

Abstract—Visual exploration has become an integral part of big spatial data management. With the increase in volume and number of spatial datasets, several specialized mechanisms have been proposed to speed up the exploration of these datasets. However, the existing techniques have major limitations which make them incapable of providing visual exploration for hundreds of thousands of big datasets on a single machine. This paper introduces a new index structure, termed AID*, that facilitates the visual exploration of an arbitrarily large number of big spatial datasets on a single machine. The AID* index defines multi-resolution fixed-size tiles on the input and classifies them as image, data, shallow, or empty tiles, based on their processing cost. Then, it uses this classification to build an index with a minimal index size and construction time, while supporting the desired real-time exploration interface. The index is constructed in parallel, using Hadoop or Spark, and is accessible to end users through a standard web interface similar to Google Maps. The small size of the index allows a single-machine server to host arbitrarily many datasets. Our experiments, on up-to 1 TB of data and 27 billion records, show that the construction of the proposed index is up-to an order of magnitude faster than the baselines without compromising the end-user interactivity.

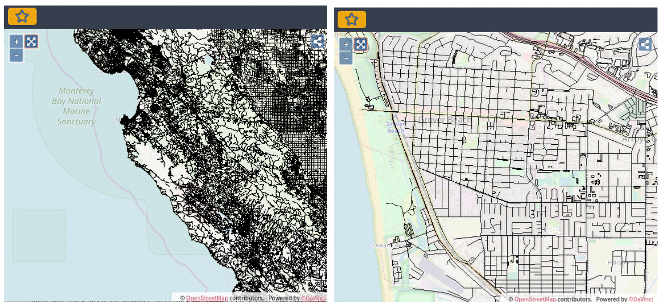
Index Terms—Interactive Exploration, Big spatial data, Indexing

1 Introduction

With the advent of IoT sensors, autonomous vehicles, satellite data, 3D microscopes, and social networks, we have hundreds of thousands and peta bytes of geospatial datasets at the disposal of scientists and developers. At the same time, there is a global movement of *open data* which aims at making large amounts of data available for the public. One prime example is data.gov which is run by the US federal government and hosts more than 250,000 datasets to the public, out of which more than 60% are geospatial. Other examples include data.gov.uk, World Bank Open Data, and United Nations Open Data. Unfortunately, users get flooded with hundreds of thousands of datasets with no easy way to choose the ones that fit their needs. For example, Data.gov lists more than 140,000 geospatial datasets in a mere textual form with no insight about these datasets. Users need to download tons of these, run a tool to process them, and finally discard most of them as unsuitable which wastes hundreds of hours of work.

Figure 1 shows the alternative method that we advocate for in this paper, i.e., the exploratory interface. In this figure, users can *see* the data on an interactive map where they can either zoom out to a bird view to inspect data coverage and distribution or zoom in to inspect individual records and assess the quality and accuracy of the data. Figure 2 depicts another example from the popular NYC Taxi Cab dataset. This figure clearly highlights the noisy nature of the data and its coverage which helps the users to decide whether or not to use this dataset. Interested readers can refer to hundreds of additional examples at https://star.cs.ucr.edu which are all built by the method proposed in this paper.

• *S.Ghosh and A.Eldawy are from University of California, Riverside E-mail:sghos006,eldawy@ucr.edu*


Monterey Bay National Marine Sanctuary
© OpenStreetMap contributors. Powered by ©DaVinci

(a) Overview of data coverage (b) Displays individual records

Figure 1: World-wide road network that can show both the a high-level coverage overview or individual records

Requirements: The goal of this paper is to empower open data repositories with a simple interactive exploratory interface that allows users to quickly explore the datasets before downloading them. Such a system must have the following five requirements: (1) **Many datasets**: The system must be able to host a large number of datasets (100,000 approx.) concurrently as we already have that many datasets available and more datasets are constantly being added. This means that users should be able to quickly switch between datasets without a significant overhead to load the data. For example, each city might publish a dataset about traffic violation and the users should be able to visualize any of them. There are some systems that can efficiently support one big dataset by pre-rendering all possible visualizations [1], [2] which would not scale to 100,000 datasets since there will be too many pre-rendered data. (2) **Big data**: The proposed system must be able to handle big data efficiently as some of these datasets can be in the order of terabytes. These datasets are not only big in terms of volume but also in terms of variety. Not only some are as big a few terabytes, they also come in different formats like geojson, csv, shapefile, wkt and so on. (3) **Individual records**: The users must be able to zoom in to see individual records as shown in Figure 1(b) to assess their quality and accuracy. (4) **Cost Effective**: The system must be hosted on a single commodity machine for cost effectiveness. A cluster of machines or multiple GPUs might be used to index or prepare the data but it cannot be used 24/7 by this system. (5) **Interactivity**: The system must be able to respond to user interactions in less than 500 milliseonds as recommended by existing HCI studies [3].

Limitations of existing work: All existing work for big spatial data visualization and exploration would fail to satisfy one or more of the five requirements. HadoopViz [4] produces a multilevel pyramid image which grows exponentially in size as the users zoom in thus it fails to satisfy requirement #1 due to the large image size and #3 due to the limited zoom depth. GeoSparkViz [5] preloads data in the distributed memory of the Spark cluster which violates requirements #1 and #3. Web maps such as OpenStreetMap [6] rely on MapBox [7] to build and visualize data but they need to produce billions of image tiles that consume terabytes of disk space [2] which does not scale to accommodate a large number of datasets (fails requirement #1).OmniSci [8], [9] preloads the dataset to a GPU to provide fast access which makes it limited to the size of the GPU (fails requirements #1 and #2). If multiple GPUs are used to support big data, the system would be expensive and fail requirement #4. Some systems rely on sampling as in VAS [10] and aggregation as in NanoCubes [1] to reduce the data size, thus, they fail requirement #3.

To satisfy the above five requirements, this paper proposes two optimized *adaptive image-data* index, termed AID and AID*, which enable visual exploration of big spatial datasets. Both indexes are entirely stored on disk and have a very small size which allows it to support arbitrarily many datasets (requirement #1). The index can be constructed using Spark or Hadoop and can be constructed on terabyte-scale datasets (requirement #2). These indexes provide access to individual records as users zoom into the visualization (requirement #3). Despite the size of the dataset, the proposed indexes, especially AID*, can always provide a subsecond query response using a single machine query processor (requirements #4 and #5).

The key idea behind the proposed indexes is to build a quadtree-like pyramid structure that covers the entire input space in 20 zoom levels similar to existing web maps. While this pyramid structure can contain up-to hundreds of billions of tiles, the AID and AID* indexes use a cost model to classify these tiles into four categories based on their processing cost. Based on this classification, only a few tiles are generated (a few tens of thousands of tiles for the biggest datasets) which can be easily stored on disk. These tiles can be generated in parallel using Spark or Hadoop in a one-time offline job. After that, a query processor, that runs on a single machine, can use these tiles to generate a visual exploration where each of these tiles are accessed or generated in less than 500 milliseconds to ensure real-time response. Depending on the system requirements on interactivity, the proposed indexes can be tuned using a single parameter θ to generate as many or as few tiles as needed in the index. The two variations of the index proposed are as follows: (1) AID stores a mix of image and data tiles in the index, and (2) AID* which stores only image tiles and reuses an existing spatial index on the data. While both indexes scale to big data, the AID* is more practical due to its low overhead and better scalability.

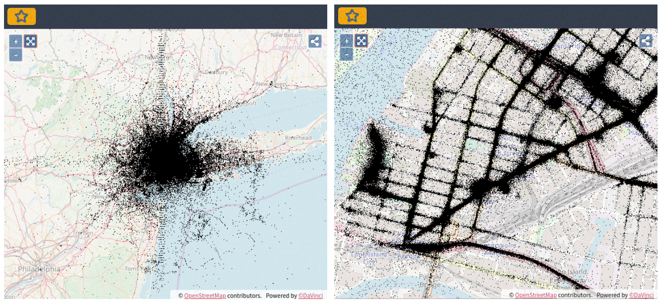


Figure 2: Scatter plot of the NYC Taxi data which reveals the noisy nature of this dataset

Our experimental evaluation shows that the proposed indexes can be generated in less than 20 minutes for a dataset of 27 billion records. At the same time, more than 99% of the visualization requests are processed within half a second on a single machine. The overhead of the index is less than 0.01% even for the 1 TB dataset which makes it perfect for handling hundreds of thousands of datasets.

Contribution The contribution of the paper can be summarized as: (1) It introduces a novel indexing technique AID* for multilevel visual exploration of spatial data. (2) AID*, with an index overhead of around 0.1% of the original data, is extremely scalable, easily providing 20 zoom-levels for 1-TB of data. (3) The extremely small size of this indexes enables building an open-sourced geospatial repository for interactive visual exploration. This repository can host more than hundreds of thousands of the spatial dataset within a single machine. (4) It introduces a visualization query that is fast and effective, keeping the response time within 500 milliseconds, making the system highly interactive. (5) It provides an analysis of the index construction method and how it affects the index sizes for uniform data which can further be extended to non-uniform data. (6) It provides the community an opportunity to visually explore and interact with the publicly available geospatial datasets, without actually having to download or process these data.

The rest of the paper is organized as follows Section 2 discusses the related work. Section 3 gives a high-level overview of the index. Section 5 details the index construction process. Section 6 describes the interactive visualization query processing. Section 7 provides an extensive experimental evaluation of the proposed work. Finally, Section 8 concludes the paper.

2 Related Work

This section covers the related work in the area of geospatial visual exploration. Table 1 summarizes how the existing work satisfy the five requirements of interactive

Work	Requirements				
	#1	#2	#3	#4	#5
VAS [10]			✓	✓	✓
imMens [11]			✓	✓	✓
Nanocubes [1]			✓	✓	✓
VisTrees [12]			✓	✓	✓
IGV [13]			✓	✓	✓
Mapzen [14]	✓			✓	
Tableau [15]	✓			✓	
OmniSci [8]	✓	✓	✓		✓
HadoopViz [4]		✓		✓	✓
Shahed [16]		✓			
Cloudberry [17]		✓			✓
GeoSparkViz [5]		✓			✓
EarthDB [18]		✓			
AID & AID*	✓	✓	✓	✓	✓

Table 1: Related work based on the five requirements

exploration. The works are classified under two categories, single-machine systems and distributed systems as follows.

2.1 Single Machine Visual Exploration Systems

The work under this category aims at using single-machine algorithms and indexes to speed up the visual exploration of big spatial data. In general, they use downsizing techniques such as aggregation, clustering, sampling, and approximation, to achieve fast response. Some of them use main-memory indexes to further speed up the query processing. For example many systems like VAS [10], imMens [11], Nanocubes [1], VisTrees [12], and IGV [13] focus on analyzing spatial data, relying heavily on aggregation and binning that negates requirement #3 preventing us from accessing individual records or giving access to only selected data. Moreover, the single machine processing of the input datasets restrict the amount of data it can process and render contradicting requirement #1. Another group of systems like Mapzen [14] and Tableau [15] focus on plotting data on top of an existing map. This visualization is done on the browser using JavaScript libraries. Due to browser limitations, these systems cannot scale to big datasets, resulting in poor interactivity and hence violating requirements #2 and #5. OmniSci [8], on the other hand is a GPU-accelerated system, limited by the size of the GPU.Much of its capacity of hosting big data or a large amount of dataset is dependent on the power of the GPU, since most GPU based systems needs the entire data to be loaded into the GPU memory for an efficient use of the GPU. A NVDIA gtx1080 has a GPU memory of 8 GB, and costs roughly around $500. Clearly one such GPU is not sufficient for even for one single dataset (say of 100 GB), making a GPU based solution super expensive.

2.2 Distributed System Visualizations

Most systems that work with big spatial data rely on distributed systems to preprocess the raw data to generate an efficient visualization. Systems like HadoopViz [4] and Shahed [16] employ Hadoop map-reduce and GeoSparkViz [5] uses Spark systems respectively, to preprocess the datasets to produce the desired visualization. They use a pyramidal

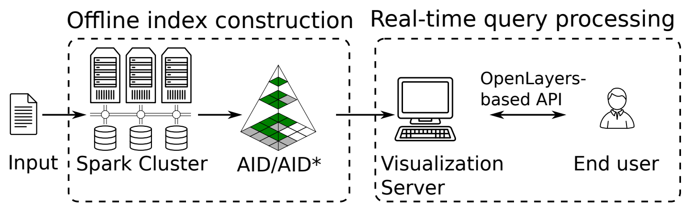


Figure 3: System overview

quad-tree [19] image index to generate a multilevel visualization. However, the amount of image tiles increase exponentially with each level. As a result of which HadoopViz, GeoSparkViz does not scale above 10 levels GeoSparkViz [20] and Cloudberry [17] provide an alternative solution that preloads and indexes the data on a cluster and generates visualizations on-the-fly. However, this defies requirement #4 which requires a single-machine system for cost efficiency. EarthDB [18] uses the array-based indexes of SciDB and Hierarchical Data Format (HDF) to speed up the selection and aggregation but similar to many other analytical systems, cannot fulfill requirements #3 and #4.

Our proposed approach is a mix of distributed and single-machine systems. It builds a light-weight disk-resident index which can serve 100,000's of datasets. These indexes (AID or AID*) are constructed on a Spark or Hadoop cluster to support big data and can provide an arbitrary number of zoom levels to visualize individual records. In AID, after the index is constructed, it is exported to a single-machine while the cluster can be used to do other work or be terminated to save the cost. For AID*, the image index as well the spatially indexed input dataset are both exported to the single-machine. Despite running on a single machine, the proposed indexes can provide an interactive response to any visualization request which makes them ideal for interactive data exploration.

3 Overview

Figure 3 gives a high-level overview of the proposed work. The system takes input geospatial datasets and provide an interactive web-based exploratory interface for end users. First, the input data goes through a distributed index construction process that runs on Spark or Hadoop. The output is an adaptive visualization index that is stored in the Hadoop Distributed File System (HDFS). Then, a visualization server provides an interactive web-based user interface to display the visualization from the generated index. The web server is hosted on a single machine for two reasons. First, it is cost effective as it releases any cluster resources so that the cluster can be used by other tenants or terminated if it is in the cloud. Second, Hadoop and Spark are not optimized for short real-time queries so a single-machine web-server can handle these visualization requests more efficiently. The proposed design supports arbitrarily many datasets by building a separate index for each one. The extremely small overhead of each of these indexes does not hinder the system from hosting hundreds and thousand of such indexes.

3.1 Offline Index Construction

This first step processes the input data in parallel to construct the proposed adaptive index (AID or AID*). We follow the multilevel pyramid structure shown in Figure 4 where each tile is stored as a separate file to allow a fully parallelized index construction where each machine generates a set of files without a need for a centralized step. Users can configure three parameters to control the index construction process, *number of zoom levels* (Z), *tile dimension* (T), and *threshold* (θ). The number of zoom levels (Z) defines the maximum zoom that the constructed index can provide. The tile dimension (T) indicates the size of each generated tiles in pixels. The threshold (θ) is used to balance the number of image and data tiles in the index while satisfying the user interactivity requirements. The index can either physically store each data tile in a file (AID), or reuse an existing R*-tree index to further reduce the index size (AID*).

3.2 Real-Time Query Processing

This part defines the visualization query and its processing using the adaptive index. We propose a simple query that retrieves a single tile as an image to be displayed to the user. This query is plugged into a web interface that uses a tile layer in OpenLayers [21] to provide the desired functionality. The challenge in this part is to generate the image tiles efficiently out of the index. Depending on the tile size, it can be available in the index as an image tile, data tile, or it might not be stored at all. Section 6 will show the details on how the visualization query handles all these cases.

4 Preliminaries and Problem Definition

This section gives some preliminary definitions for the multilevel visualization problem. Figure 4 illustrates an example of the pyramid structure that we define below.

Definition 1. A two-dimensional *bounding box* (BB) is a rectangular area in the two-dimensional plane $\mathbb{R}^2$ which contains all points (x, y) that satisfy $x_{min} \leq x < x_{max}$ and $y_{min} \leq y < y_{max}$. It has a width of $w = x_{max} - x_{min}$ and height $h = y_{max} - y_{min}$. Based on the context, we either define a BB by x_{min}, y_{min}, x_{max}, and y_{max}, **or** by x_{min}, y_{min}, w, and h.

Definition 2. A *feature* f is record that is associated with a bounding rectangle ($f.BB$). Similarly, a *feature set* is a set of features $\mathcal{F} = \{f\}$. The bounding box of $\mathcal{F}$ is the minimum bounding box that contains all features in the set. $\mathcal{F}.BB = BB\left(\bigcup_{f\in\mathcal{F}} f.BB\right)$

Definition 3. A *pyramid* $\mathcal{P}$ is defined by an integer Z that represents the number of zoom levels and a bounding box $\mathcal{P}.BB$. Each zoom level in the pyramid $0 \leq z < Z$ contains a set of tiles organized in a uniform grid with 2^z rows and columns. The total number of tiles in level z is 4^z. The total number of tiles in a pyramid with Z levels is $\sum_{z=0}^{z<Z} 4^z = (4^Z - 1)/3$. Since we start the zoom level at 0, the value of z always ranges from 0 to $Z-1$. For example for $Z = 20$ zoom-levels, z will range from 0 to 19. In summary, the number of tiles increases *exponentially* with the number of zoom levels.

Definition 4. Each tile t is identified by three parameters $\langle z, x, y\rangle$, where z is the zoom level, x and y are the column and row indexes of the tile in that level, $x, y \in [0, 2^z]$ and are both integers. Each tile covers an area defined by the bounding box $t.BB$ calculated as follows:

$$\begin{aligned} t.BB.w &= \mathcal{P}.BB.w/2^z \\ t.BB.h &= \mathcal{P}.BB.h/2^z \\ t.BB.x_{min} &= \mathcal{P}.BB.x_{min} + t.BB.w \cdot x \\ t.BB.y_{min} &= \mathcal{P}.BB.y_{min} + t.BB.h \cdot y \end{aligned} \tag{1}$$

Definition 5. We say that a tile $t_1 = \langle z_1, x_1, y_1\rangle$ is the *parent* of another tile $t_2 = \langle z_2, x_2, y_2\rangle$ if $z_1 = z_2 - 1$ and $x_1 = \lfloor x_2/2\rfloor$ and $y_1 = \lfloor y_2/2\rfloor$. The root tile $\langle 0, 0, 0\rangle$ does not have a parent.

Definition 6. Given a tile $t = \langle z, x, y\rangle$ and a feature set $\mathcal{F}$, the contents of the tile t, called $t.F$, is the set of all features $f \in \mathcal{F}$ that overlap the area covered by the tile based on their bounding boxes, i.e., $t.F = \{f \in \mathcal{F} : f.BB \cap t.BB \neq \emptyset\}$.

Definition 7. *Tile visualization* is the process of converting the contents of a tile t to an image with dimensions $T \times T$. In this paper, we use the visualization abstraction proposed by HadoopViz [4] which can generate a variety of visualization, e.g., scatter plot and heat map, in both raster and vector formats.

Problem Definition: Given a feature set $\mathcal{F}$, a pyramid $\mathcal{P}$ with Z zoom levels, and an image dimension T, we need to visualize any tile t in the pyramid within a specific time limit, e.g., 500 milliseconds.

Background: HadoopViz [4] solves the multilevel visualization problem by pregenerating and materializing *all* non-empty tiles which proved to work well for 10-12 zoom levels but does not scale beyond that due to the exponential increase in the number of tiles with zoom levels. It uses a traditional image index, which explodes exponentially with increasing zoom levels.

5 Index Construction

The index construction phase is responsible for processing the input features set $\mathcal{F}$ to generate the proposed AID or AID* index. This construction process runs in a distributed environment and we implement it on both Spark and Hadoop. Index construction is an offline phase which is carried out before users start visualizing the data.

There are two main design objectives of this phase, minimize the index size and reduce the index construction time. To achieve these goals, we propose two adaptive index construction methods, AID and AID* that classify the tiles based on their estimated processing time into four classes, *image*, *data*, *shallow*, and *empty* tiles. In **AID** [22], two types of tiles are materialized to disk, image and data. Image tiles contain prerendered images and data tiles contain data records that can be used to generate more tiles as needed on-the-fly. In **AID***, only image tiles are materialized in the index. Instead of storing the data tiles, an existing general purpose R*-tree index is utilized to generate the tiles that

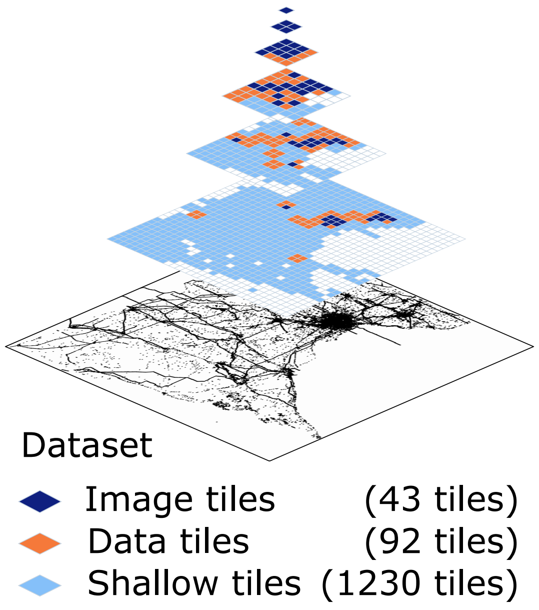


Figure 4: Adaptive indexing showing different number of image, data and shallow tiles

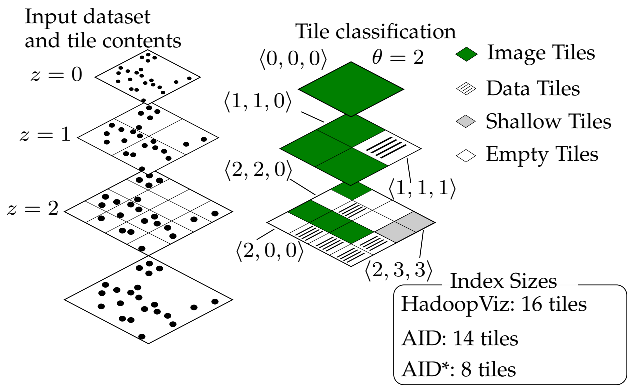


Figure 5: Tile classification example

are not materialized. The rest of this section describes the out in Section 5.1. Section 5.2 details the process of building the index. Section 5.3 derives a theoretical analysis for the index construction phase.

5.1 Index Layout

The proposed index follows the multilevel pyramid layout similar to the one illustrated in Figure 4.

Tile Classes: In this paper, we make the observation that not all tiles are equal from a data management perspective. For example, a tile that covers New York City is more expensive to visualize than a a tile in a small city like Palm Springs due to the disparity in the number of features in each tile. Based on this observation, we classify the tiles into four classes, namely, *image*, *data*, *shallow*, and *empty* tiles. We use Figure 5 as a running example where the input has 22 records indicated as points, and the four tile classes are illustrated. The ID of some tiles is indicated between $\langle \dots \rangle$ to refer to in the discussion. Below, we first define the parameter θ and then we define the four classes of tiles based on θ.

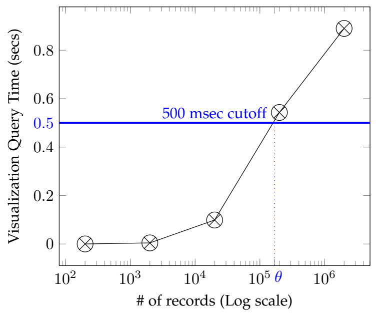


Figure 6: Tuning of the parameter θ

Definition 8. The threshold θ is an index parameter that defines the largest number of records that can be visualized within the time limit of 500 milliseconds.

The visualization performance relies on the implementation of the visualization abstraction (Definition 7). To tune the parameter θ, we simply run a mini-experiment on a single thread where we gradually increase the input size and measure the visualization time and observe when the 500 millisecond cutoff is reached. Figure 6 shows the results of this experiment on a point dataset where the cutoff is reached at around 160,000 records. Normally, this tuning step would take around 2θ time, i.e., one second.

Definition 9. A tile t is an **image tile** if it has a size $|t.F| > \theta$. In Figure 5, tile $\langle 1, 1, 0\rangle$ is an image tile because it has five records while $\theta = 2$.

Definition 10. A tile t is a **data tile** if its size $0 < |t.F| \leq \theta$ *and* its parent tile t_p has a size $|t_p.F| > \theta$. In other words, the parent tile t_p is an image tile.

In Figure 5, tile $\langle 1, 1, 1\rangle$ is a data tile because it has two records while its parent $\langle 0, 0, 0\rangle$ is an image tile.

Definition 11. A tile t is a **shallow tile** if its size $0 < |t.F| \leq \theta$ *and* the size of its parent $|t_p.F| \leq \theta$. In other words, its parent is *not* an image tile.

In Figure 5, tile $\langle 2, 3, 3\rangle$ is a shallow tile as it has one record while its parent is a data tile.

Definition 12. A tile t is an **empty tile** if it does not contain any features, $t.F = \emptyset$. Tile $\langle 2, 0, 0\rangle$ is an example of an empty tile.

In traditional image indexes [4], each non-empty tile is stored as an image in a separate file named '`tile-z-x-y.png`'. This makes the total number of tiles in a pyramid $\mathcal{P}$ with Z zoom levels, $|\mathcal{P}| = (4^Z - 1)/3$, which exponentially explodes as Z increases.

In the proposed AID index, we reduce the size of the index by only storing image and data tiles where an image tile is stored as a prerendered image and a data tile is stored as a data file. In AID*, we further reduce the size of the index by only storing image tiles while using a general purpose R*-tree index to replace all data tiles.

Each of these image tiles are .png files or raster images of a fixed size (256×256 default image size). However the data coverage that constitute these images can be way larger. For example, a data of 1000 Mb can generate an .png file of size 4 Mb. The size of the data file is dependent on the size of θ. A

Algorithm 1 Classify a tile in constant time

```
1:  function ClassifyTile($t = \langle z, x, y \rangle, H, \theta$)
2:      $k = H.size/2^z$                          ▷ Number of cells covered by $t$
3:      $(c_1, r_1) = (x \cdot k, y \cdot k)$ ▷ First (column,row) covered by $t$
4:      tileSize = $\sum_{c=c_1}^{c<c_1+k} \sum_{r=r_1}^{r<r_1+k} H[r, c]$   ▷ $O(1)$ operation
5:      if tileSize = 0 then return empty
6:      if tileSize > $\theta$ then return image
7:      if z=0 then return data   ▷ Special case for the root tile
8:      // Compute the size of the parent tile
9:      $x = \lfloor x/2 \rfloor$; $y = \lfloor y/2 \rfloor$
10:     $k = \lfloor k/2 \rfloor$; $(c_1, r_1) = (x \cdot k, y \cdot k)$
11:     parentSize = $\sum_{c=c_1}^{c<c_1+k} \sum_{r=r_1}^{r<r_1+k} H[r, c]$ ▷ $O(1) operation$
12:     if parentSize > $\theta$ then return data ▷ Parent is image tile
13:     return shallow
```

data tile can be extremely small, even smaller than the size of an image tile, if an extremely small value of θ is chosen. However, choosing extremely small θ can make it almost equivalent to image-only indexes. The results of such low θ are explained in more details in Section 7.6. The next part explains how the AID and AID* indexes are constructed.

5.2 Index Building

This part describes how to build the AID and AID* indexes. The input consists of a set of records $\mathcal{F}$, a threshold θ, and the number of zoom levels Z to generate.

In this paper, we consider the two types of data partitioning (default and pyramid partitioning) and we make two new contributions. 1) we add a preprocessing summarization phase that computes a histogram of the input data. This histogram is used to estimate the size of each tile. 2) we enrich the tile creation phase by employing a novel *tile classification* algorithm that accurately classifies each tile in constant time. The index building process consists of three phases, *data summarization*, *tile classification*, and *tile creation* as described below.

5.2.1 Data summarization

This first preprocessing phase summarizes the data to allow classifying the tiles based on their visualization cost. Based on the work in [23], we build a uniform grid histogram with dimensions $h \times h$, where $h = 2^{\min(z_{max}, h_{max})}$, $z_{max} = Z - 1$ is the maximum zoom level requested by the user and h_{max} is an upper bound which is configured based on the available memory in the system.

The summarization phase runs as two Spark jobs one computes the MBR of the input which is used to define the grid dimensions and the other uses those grid dimensions to compute the histogram values, in parallel. Finally, the histogram is processed on a single machine by computing the prefix sum along rows and columns to enable the constant time estimation and the final histogram is broadcast to all the cluster nodes. More details can be found in [23].

5.2.2 Tile classification

In this part, we show how we use only the histogram and the threshold θ to classify any tile in constant time. Algorithm 1 shows the pseudo code of the classification algorithm. The input is a tile ID $\langle z, x, y \rangle$, the histogram H which is a two dimensional array of numbers, and the size threshold θ. The output is one of the four labels, *image*, *data*, *shallow*, or *empty*. Lines 2-3 compute the range of grid cells in the histogram that overlap the given tile. The top-left corner of the overlapping block of cells is at (c_1, r_1) and the block contains $k \times k$ cells, as shown in the pseudo code. Line 4 computes the size of the tile. Notice that we use the summation notation for clarity but the algorithm uses the prefix-sum to compute the summation in constant time. If the computed size is zero, the tile is classified as empty. If it is larger than the threshold θ, it is classified as an image tile. Otherwise, we have to check the size of its parent tile as well to decide whether it is a data tile or a shallow tile. Line 7 handles the special case where the tile is the root and there is no parent tile in which case it has to be a data tile. Otherwise, Lines 9-11 compute the size of the parent tile similar to Lines 2-4. Finally, Lines 12-13 determine the type of the tile based on the size of its parent. If the parent is an image tile ($size > \theta$) (Line 12), then the tile is a data tile. Otherwise, the tile has to be a shallow tile (Line 13).

In the example in Figure 5, we need a histogram of size $2^{3-1} \times 2^{3-1}$. For this example, we set the threshold θ to two records. The root tile $\langle 0, 0, 0 \rangle$ is an image tile as it covers the entire histogram with a total size of 22 records. At level 1, three tiles are classified as image tiles except for tile $\langle 1, 1, 1 \rangle$ which is classified as a data tile because it has two records only. At level 2, more tiles are classified as data tiles. The two shaded tiles at level 2 are classified as shallow tiles because they are not empty and their parent is a data tile, e.g., tile $\langle 2, 3, 3 \rangle$.

Algorithm 2 Create tiles

```
1:  function CreateTiles($\mathcal{F}, H, \theta, [z_L, z_H]$)
2:  Initialize $\mathcal{P}$ to an empty hash map
3:  for each record $f \in \mathcal{F}$ do
4:      for each tile $t$ that overlaps $f$: $z \in [z_L, z_H]$ do
5:          $td$ = retrieve the tile data @$(z, x, y)$ from $\mathcal{P}$
6:          if $td$ is NULL then
7:              tile-class = ClassifyTile($(z, x, y), H, \theta$)
8:              if tile-class is image then
9:                  Initialize $td$ as an empty image canvas
10:             else if index type is AID then
11:                     and tile-class is data
12:                 Initialize $td$ as an empty set of records
13:             Insert $td$ into $\mathcal{P}$ at position $(z, x, y)$
14:         if $td$ is an image canvas then
15:             Plot the feature $f$ on the image canvas $td$
16:         else if $td$ is a set of records then
17:             Add the feature $f$ to the set $td$
18: return $\mathcal{P}$
```

5.2.3 Tile Creation

This phase scans the input and creates the image tiles and optionally the data tiles based on the histogram computed earlier. Similar to HadoopViz [4], we introduce two algorithms, namely, *default-partitioning-based* algorithm and *pyramid-partitioning-based* algorithm. The two techniques are synchronized together to produce the desired multilevel image. The primary logic of the two algorithms is based on the function CreateTiles listed in Algorithm 2 and described below.

The function takes four parameters, a feature set $\mathcal{F}$, a histogram H, a threshold θ, and a range of zoom levels $[z_L, z_H]$. It returns a pyramid $\mathcal{P}$ as a list of key-value pairs where the key is a tile ID and the value is either an image, for image tiles, or a set of records for data tiles. Notice that $\mathcal{F}$ is *not* the entire input set but a subset based on how the data is partitioned as explained shortly. The function has two big loops in Lines 3 and 4 that iterate over each feature $f \in \mathcal{F}$ and each tile that overlaps the MBR of that record in the given range of zoom levels $[z_L, z_H]$. For each overlapping tile t with a record f, Line 5 tries to retrieve the tile data td from the pyramid $\mathcal{P}$. If the tile data is not in the pyramid (Line 6) it has to be initialized to either an empty canvas (image), for image tiles, or an empty set of records, for data tiles. Line 7 uses the CLASSIFYTILE function (Algorithm 1) to determine the type of the tile which is then initialized in Lines 8-12 based on its class. Lines 14-17 process the record by either plotting it on the canvas (for image tiles) or adding it to the set of records (for data tiles). Notice that AID and AID* only differ in Line 11.

The CREATETILES function is used as a building block to create all the tiles in parallel. The pyramid is split horizontally into two ranges of zoom levels $[0, z^*]$ and $[z^*+1, Z-1]$, where z^* is computed as shown in [4]. The upper part of the pyramid is processed using the flat partitioning algorithm which partitions the data randomly and uses the CREATETILES with $z_L = 0$ and $z_H = z^*$ on each partition to generate partial tiles. The created partial tiles are shuffled, merged, and finally written to the output. The lower part of the pyramid is processing using the pyramid partitioning method which first partitions the data based on the tiles and then the tiles in each partition are generated using the CREATETILES function and written to the output. No shuffle or merge is needed for the pyramid partitioning algorithm.

5.3 Analysis of Index Construction

To further understand the cost of the index construction and index size, this part makes a simple analysis of the index size and relates it to the index construction time. Our approach is to estimate the number of tiles that affect the index sizes, i.e., image, data, and shallow tiles. Then, based on which tiles are constructed and stored in each type of index, we can estimate their cost.

Let us assume that the input has N uniformly distributed points, number of zoom levels Z, an image tile dimension T, and a threshold θ that represents the maximum number of records in a data tile. The uniform distribution results in all tiles in each zoom level z to have the same number of records:

$$N_z = N/4^z, z \in [0, Z) \tag{2}$$

Where N_z is the number of records in a tile at level z. As a result, all data tiles appear at one zoom level, say z_D, while all higher levels ($z < z_D$) contain image tiles. To determine the value of z_D, we notice that all tiles at $z \geq z_D$ have less than θ records as the tiles get smaller in deeper levels. That is, $\forall z \geq z_D : N_z < \theta$. Therefore, $\forall z \geq z_D : z > \log(N/\theta)/2$. Since $z \geq z_D$ in this inequality, z_D is the smallest integer that satisfies the inequality which results in:

$$z_D = \lceil \log_4 (N/\theta) \rceil \approx \log_4 (N/\theta) \tag{3}$$

Notice that if the user requests fewer levels, that is, $Z \leq z_D$, the data tiles are too deep to be reached and no data tiles will be processed or generated. The number of data tiles D is computed as:

$$D = \begin{cases} 0 & ; Z \leq z_D \\ 4^{z_D} & ; Z > z_D \end{cases} \tag{4}$$

Similarly, the total number of zoom levels with image tiles is $Z_I = \min\{z_D, Z\}$ which makes the total number of image tiles I given as below.

$$I = \sum_{z \in [0, Z_I)} 4^z = \frac{4^{Z_I} - 1}{4 - 1} \tag{5}$$

Finally, the number of shallow tiles (S) is calculated as

$$S = \sum_{z \in (z_D, Z)} 4^z = \frac{4^Z - 1}{4 - 1} - D - I \tag{6}$$

We approximate Equations 4, 5, and 6, by assuming $Z > z_D$ and removing the ceiling and floor functions. This results in the following approximations.

$$D \approx 4^{z_D} \approx N/\theta \tag{7}$$

$$I \approx \frac{4^{z_D} - 1}{4 - 1} \approx N/3\theta \tag{8}$$

$$S \approx 4^Z/3 - N/\theta - N/3\theta = \frac{1}{3}(4^Z - 4N/\theta) \tag{9}$$

Where I, D, and S are the number of image tiles, data tiles, and shallow tiles, respectively. Based on the above analysis, we can estimate the size of each of the three indexes as follows. (1) **AID***: The AID* index only materializes image tiles. This makes the index size upper-bounded by N/θ tiles. Each tile has a resolution of T^2 which makes the total index size to be $T^2 N/\theta$. (2) **AID**: The AID index stores both the image and data tiles. Since all the data tiles appear in one level, their total size is equal to the input size N. This makes the index size equal to $4N/3\theta$ files with a total size of $T^2 N/\theta + N$. (3) **HadoopViz:** In HadoopViz, all non-empty tiles are materialized as images which makes the index size equal to $D + I + S = 4^Z/3$ tiles with a total size of $T^2 4^Z/3$. Both AID versions increase linearly with the input size which makes them more scalable as we experimentally verify in the experiments section. In addition, AID* is much smaller in terms of number of files and total size which makes it more scalable.

An important observation is that the parameter θ defines the value of z_D which in turn define an upper bound for the number of image and data tiles. However, the number of shallow tiles can be arbitrary large. This explains why AID index can be much smaller than the image index and AID* is even smaller. In the image index like HadoopViz, all the shallow tiles are also materialized as images. Since we assume that data are uniformly distributed, we do not consider empty tiles in the cost model. However, this analysis can be extended for non-uniform distribution of data based on the box-counting technique [24], [25]. In such a case, the biggest change in the cost model would be introduction of empty tiles (E). The box counting technique estimates the number of non-empty buckets in a histogram given the cell size of the histogram using an exponential

Algorithm 3 Get a tile from the index

```
 1: function GetTile($z, x, y$)
 2: if an image tile tile-z-x-y.png is available then
 3:     return the pregenerated image tile
 4: // Compute the MBR of the requested tile
 5: tileMBR.width = InputMBR.width / $2^z$
 6: tileMBR.height = InputMBR.height / $2^z$
 7: tileMBR.x = InputMBR.x + tileMBR.width * x
 8: tileMBR.y = InputMBR.y + tileMBR.height * y
 9: if index type is AID* then
10:     Retrieve the records in tileMBR from the R*-tree
11:     Visualize the retrieved records and return the image
12: // AID index. Locate and process the ancestor data tile
13: while $z \geq 0$ and file 'tile-z-x-y' does not exist do
14:     $z = z - 1; x = \lfloor x/2 \rfloor; y = \lfloor y/2 \rfloor$ ▷ Go the parent tile
15: if 'tile-z-x-y.png' exists then
16:     return empty image
17: else if 'tile-z-x-y.csv' then
18:     // Either a data or a shallow tile
19:     Retrieve the records in tileMBR from the data tile
20:     Visualize the retrieved records and return an image
```

correlation function. To compute the correlation coefficients, we need to first compute several histograms of the data to catch its distribution. Once the number of non-empty tiles are calculated, Equations 4- 6 will need to be adjusted so that they add up to the number of non-empty tiles.

Non-uniform data can also result in each tile at a particular zoom level z to have unequal amount of records. As a result, there will not be a z_d that will have all the data tiles. Instead z_d can be defined as the level where first data tile/s appear. Similarly the definition of Z_i gets altered accordingly. Even though the actual numbers that our cost model produces might be inaccurate due to the uniformity assumption, we will show in the experiment section that the general trends are similar, so, we will leave the extension of the cost model to a future work.

6 Visualization Query

This section describes how the visualization server uses the indexes to provide an interactive exploratory interface. The main challenge is to provide a real-time experience to the end users. Especially, data and shallow tiles need to be processed to visualize. In the next part, we first define the visualization query and then we show how to answer it using the AID and AID* indexes.

6.1 Query Definition

The primary link between the front-end visualization interface and the visualization server is the query to fetch the tile called GetTile. The input is a tile ID $\langle z, x, y \rangle$ and the output is an image that represents this tile. The image can be either a vector image or a raster image as defined by the visualization abstraction (See Definition 7). Notice that while the index might contain image or data tiles, the return value of the GetTile query is always an image. The conversion of the raw data file into images in real time is the primary challenge in visualization query. This keeps the index structure transparent to the front-end interface in the browser.

6.2 Query Processing

Given a tile ID, this part describes how to generate and/or return the corresponding tile image given an AID or an AID* index. Since the visualization server can host many datasets and cannot keep all their histograms in the memory, the histograms are discarded after the index creation. Therefore, the query processor only relies on the generated tiles to answer the query. For simplicity, we assume that image tiles and data tiles are in '`.png`' and '`.csv`' files, respectively.

Algorithm 3 provides the pseudo code for the GetTile function which is more clearly explained with Figure 7. First, if the user requests a tile which is stored as an image tile (e.g., tile A in Figure 7), the corresponding image is returned. This case is handled in the code in Lines 2-3. Otherwise, an image might need to be generated on the fly. Lines 4-8 compute the MBR of the requested tile to retrieve the corresponding data from the index as explained in Definition 4.

At this point, the query processing of AID and AID* diverges. In Lines 9-11, if an AID* index is being processed, then there is an accompanying data index R*-tree. The index is directly processed with a range query of the tileMBR to retrieve all overlapping records. The retrieved records are visualized on-the-fly and the generated image is returned.

On the other hand, if an AID index is being processed, then there is no R*-tree index but there are data tiles. In this case, the index is searched for the corresponding data tile. Lines 13-14 keep climbing up the pyramid structure until the first file is located. If that file is an image tile (e.g., Tile B in Figure 7) this indicates that the requested tile is empty and an empty image is returned. If that file is a data tile, it indicates that the requested tile is either a data tile (e.g., Tile C) or a shallow tile (e.g., Tile E). Both are processed in the same way as shown in Lines 17-20. The contents of the data tile is searched for all records that overlap the tileMBR and those records are visualized and the generated image is returned. As an example, consider an AID index built on the dataset in Figure 5 and a visualization query that requests the tile $\langle 2, 3, 3 \rangle$ at level 2. First, the server looks for either an image or a data file named '`tile-2-3-3.png`' or '`tile-2-3-3.csv`' which are both not found. Next, it checks for the parent tile $\langle 1, 1, 1 \rangle$ which happens to be a data tile, i.e., a file '`tile-1-1-1.csv`' is found. It concludes that the requested tile is a shallow tile and generates it from the encountered data file.

One major difference between AID and AID*, is that AID can identify an empty tile (Line 16) and process it very efficiently by returning an empty image. On the other hand, when processing AID*, the algorithm will always search the R*-tree index for any tile that is not pregenerated which is typically more costly. On the other hand, data and shallow tiles are processed by first retrieving the records and the retrieved records are visualized. According to the design of both AID and AID*, the number of records to visualize is always upper-bounded by θ to ensure an interactive response time of the visualization query.

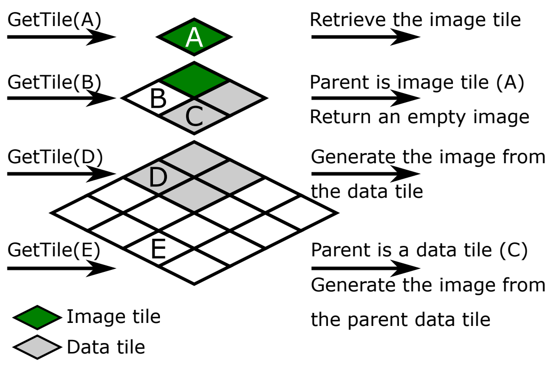


Figure 7: Examples of the visualization query

7 Experiments

This section aims at providing an extensive experimental evaluation of our proposed work. We primarily highlight the following: 1) Our index can scale up-to terabytes of data, 2) It is applicable across platforms (Spark or Hadoop), but performance varies for these two, 3) The preprocessing phase for visualization is significantly faster than the baseline, 4) There is a significant decrease in the index size with the proposed index, and 5) The visualization query operates within 500 milliseconds for almost all the queries.

The experiments are based on four performance metrics: 1) Index construction time 2) Index size in terms of number of files, 3) Index overhead as the percentage increase over the non-indexed data, 4) Visualization query time. Additionally we also depict the effect of threshold θ and tile dimension of each tile on the proposed index.

7.1 Experimental Setup

The experiments are executed on Amazon Web Service (AWS) with m3.xlarge nodes (30–150 nodes) for scalability and reproducibility. The visualization server runs on a single machine with 16 cores, 128 GB of RAM, and 10 TB HDD. The front-end of the server is implemented in JavaScript using OpenLayers APIs [21] while the back-end is implemented as a Jetty server in Java. Table 2 lists the datasets that we use in this work. Unless otherwise mentioned, the experiments use the default parameter values listed in Table 3. We denote the threshold θ in terms of bytes by multiplying with the average record size.

Table 2: Input Datasets

Dataset	Size	# records	Description
All-Nodes	96GB	2.7 B	All points on the map
All-Nodes$\times n$	$n\times$96GB	$n\times$ 2.7B	All-Nodes replicated n times (up-to 10)
Buildings	26 GB	115 M	Buildings footprint
Tweets	1.6 GB	20 M	Geotagged tweets

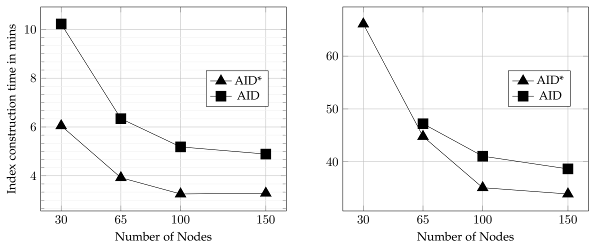
We use the following notation for the algorithms that we use in this part. The suffix '/Hadoop' and '/Spark' indicates which implementation is tested and on which platform.

7.2 Scalability and Platform Independence

To show the scalability of the proposed indexes, this experiment constructs AID and AID* indexes of $Z = 20$ zoom levels on the 1 TB `All-nodes×10` dataset. We show the index construction time on both Hadoop and Spark. This experiment propagates the following important points. 1) The proposed indexes can be be implemented in both Spark and Hadoop. 2) Both Hadoop and Spark scale well on large clusters. 3) Spark is generally more efficient due to its architectural differences. 4) Spark is up-to an order of magnitude faster than Hadoop.

Table 3: Parameters and default values

Parameter	Values (default)
Input size	1.6 GB – 1 TB
Number of levels (Z)	1–(20)
Threshold (θ)	10 KB $\cdots$ (10 MB) $\cdots$ 1 GB
Tile dimensions (T)	(256), 512, 1024 pixels
Cluster size	12, 30, 65, (100), 150 (nodes)



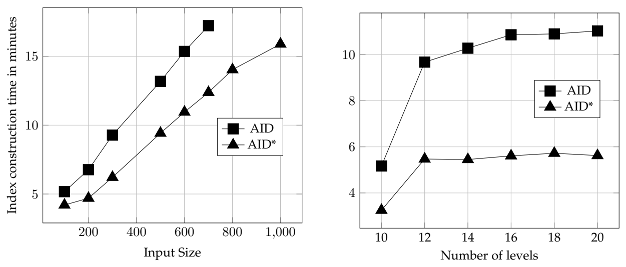
(a) Spark (b) Hadoop

Figure 8: Scalability of AID and AID* on Hadoop and Spark

Hadoop failed to produce an AID index of the 1 TB dataset with 30 nodes after taking too much time. This experiment not only shows the scalability of the proposed indexes, but also proves the advantage of AID* over AID.

7.3 Index Construction Time

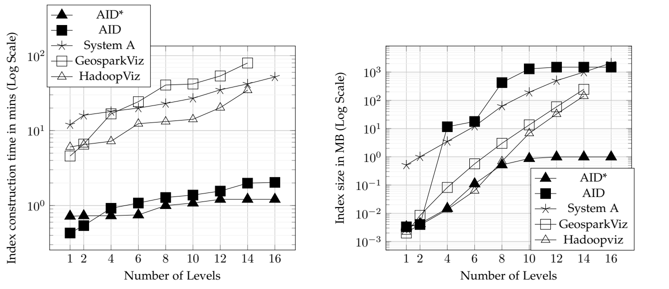
This section studies the index construction time for both AID and AID*. HadoopViz is kept out of these experiments because it doesn't scale beond level 10. In Figure 9(a), we vary the input size from 100 GB to 1 TB and report the overall index construction time. As shown, while both AID* and AID scale well up-to 800 GB, AID ceases to scale beyond that, while AID* continues up-to 1TB. Additionally, AID* is generally more efficient as it only generates image tiles while AID generates both image and data tiles.



(a) Effect of input size (b) Effect of number of levels

Figure 9: Index construction time

In Figure 9(b) we fix the input size at 100 GB and vary the number of levels from 10 to 20. We have the following four observations in this experiment. 1) All variations of AID and AID* scale well with large number of zoom levels. 2) For small number of levels, e.g., $Z = 10$, AID and AID* behave almost similarly with very small difference in construction time. According to our analysis in Equation 3, $z_D = \lceil \log(100\,GB/1\,MB)/2 \rceil = 9$ which means that most of the tiles in the generated pyramid are image tiles which makes both techniques similar. As Z increases, we start to see a difference between the different algorithms depending on which tiles are generated in each index. 3) As we increase the number of levels beyond 12 levels, the running time starts flattening for both AID and AID*, as they stop generating new tiles. In AID, as we move into deeper levels, the number of image tiles decreases, and the number of data tiles increases first for a few initial levels, and eventually we get shallow and empty tiles which do not require any computation time for AID. For AID*, we do not generate data tiles because of an already pre-existing data index which causes the index construction time to stabilize at an earlier level ($Z = 12$.)



(a) Index construction time across various systems (b) Index size for various systems

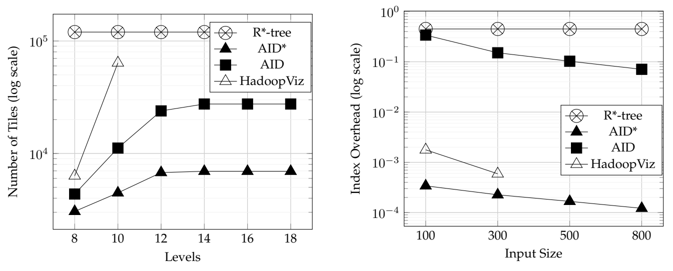
Figure 10: Various system comparisons in terms of index construction time and index sizes.

We conduct a similar experiment on other systems with Twitter data and a cluster of 12 nodes as shown in figure 10(a). A smaller input data was chosen to accommodate system A which runs on a single machine. It should be noted that despite choosing a smaller dataset, while generating indexes for System A, we had to choose an option that allows dropping dense tiles as and when required without which, System A failed to generate indexes beyond level 6. It should also be noted, according to the latest implementation of GeosparkViz [20], their emphasis is on generating specific image tiles based on spatial range query, instead of generating the entire index together. The observations are as follows: 1)AID and AID* takes extremely small construction time as compared to other systems, mainly because, they spend way less time creating images from the data. 2) Geosparkviz, which is designed to produce extremely fast spatial range query visualization, takes longer preprocessing time with increasing zoom levels. 3) System A takes the longest time, mainly because it is a single machine system and despite being able to operate on parallel threads, it fails to match up with a distributed system set-up. 4) GeosparkViz fails to produce an index beyond level 14. It makes sense because GeosparkViz is not designed for a deep multilevel visualization. Instead it more efficient for visual analysis and queries.

7.4 Index size

In this section we measure the index size. The experiments measure the index size using two metrics. 1) The number of files which is an important metric given that the index will be hosted on a single machine web server. The performance of many file systems degrade as the number of files increase so we would like to reduce this number. For the sake of comparison, we set the node size in R*-tree to 1 MB (equal to θ) and count the number of leaf nodes. 2) The index overhead as the ratio of the added size divided by the original size. This measures how much additional data we need to write to build the index. HadoopViz rarely scales above level 10. Hence to keep the comparison fair, we have computed the index overhead for AID and AID* upto a zoom level 10.

Figure 11(a) depicts the index size, in terms of number of materialized tiles, for `All-nodes`. For lower levels, AID*, AID and HadoopViz produce almost the same number of tiles which indicates that Z is too small to generate many data tiles. As the number of levels increases, the number of tiles of HadoopViz index exponentially explodes while both AID and AID* keep it under control. HadoopViz fails to generate an index beyond level 10 as it takes too long to execute. R*-tree is oblivion of the zoom levels and is not affected by changing zoom levels.



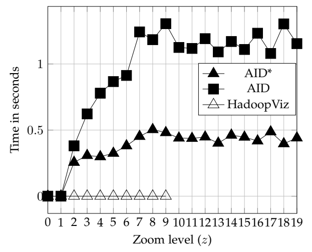
(a) Tiles Vs Zoom Levels (b) Index Overhead Vs Input Size

Figure 11: Index size

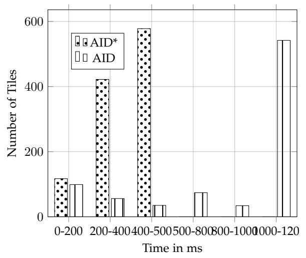
In Figure 11(b), we increase the input size from 100 GB to 800 GB and measure the index overhead. 1) The overhead of the proposed AID* index is minimal and it is always less than 0.1% as it only contains a few image tiles. The extreme small index overhead also ensures that 1000,000 of big datasets can be hosted on a single machine. 2) The index overhead for AID and AID* decreases with increasing input size, whereas for R*-tree index it remains constant for any input size. For R*-tree, the index overhead is around 45% which is in line with the work in literature. For AID and AID*, the overhead decreases because the number of tiles is upper bounded due to the fixed size of the pyramid ($Z = 20$) while the input size can increase indefinitely. Therefore, it is expected that the index size increases at a slower rate as compared to the input size. 3) HadoopViz fails to run on any index sizes beyond 300 GB of data. 4) AID* has almost 3.5 times less overhead compared to AID, which is caused by the AID's requirement to generate both data and image tiles. This confirms our theoretical analysis in Equations 7 and 8 where the number of images tiles is roughly one third the number of data tiles.

Figure 10(b) shows how index size increases for different systems with increasing zoom levels for Twitter data. The figure provides the following observations: 1) AID* takes least indexing space, which makes it obvious why it can support multiple datasets unlike many other systems. 2) The index size becomes constant for AID and AID* after level 8 while others show an exponential increase. 3) The index size of GeosparkViz and Hadoopviz is almost same and it expands exponentially with the zoom levels.

7.5 Visualization Query



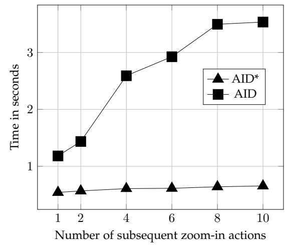
(a) Average visualization time per zoom level



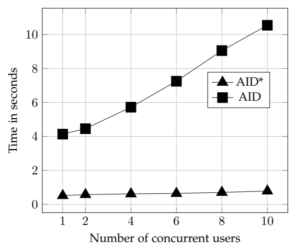
(b) Number of generated tiles grouped by visualization time

Figure 12: Visualization query processing time

This part evaluates the performance of the visualization query described in Section 6 In this experiment, we use a benchmark that comprises a set of 1,000 random points in the input space. For each point, we generate all the overlapping tiles in levels 0 to 19 for `All-Nodes` dataset in HadoopViz, AID, and AID*. This benchmark simulates the real workload of users zooming in from the root tile to a chosen location on the image. This runs on a single machine. We can see the following observations in this experiment. 1) For levels $[0, 2]$ all techniques, AID, AID*, and HadoopViz, take almost the same time because all the tiles at these top levels are usually image tiles which are just retrieved from disk. 2) Since HadoopViz materializes all possible images, the visualization time remains constant regardless of the zoom level. While being efficient in this query, it is also considered *over-optimized* from the visualization perspective as the users do not usually realize the difference between a 1 millisec and a 500 millisec [3] response time. Moreover since HadoopViz does not scale over 10 levels in the index construction phase, we can explore it visually only up-to level 10. 3) The performance of AID and AID* increases as the zoom level increases as they start to processes data records to generate images on the fly. Both AID and AID* do a good job at keeping the running time below 1 second on average for the initial 8-9 levels. Although AID* is significantly more efficient throughout the 20 levels as it relies on an efficient R*-tree index while AID relies on small non-indexed raw files of 1 MB each. It should also be noted that the AID's query time increases significantly after level 10. The primary reason for this phenomenon is, all the tiles beyond level 10 are shallow tiles and each time a shallow tile is requested the server goes up to fetch the parent tile and generate the image of the shallow tile. The deeper the zoom level is the higher the server has to go up the pyramid to get the parent tile of the requested shallow tile. This increases the average query processing time for the tiles belonging to the parent tiles' level (say level 9,10 and 11 for all nodes).



(a) Visualization query time for subsequent zoom-in action



(b) Visualization query time for multiple users

Figure 13: Visualization query time with parallel requests

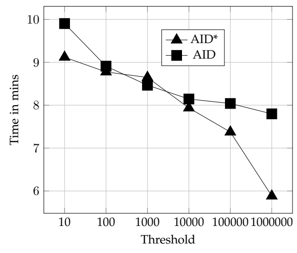
Figure 12(b) projects the number of tiles that are computed within a specific time to project the number of tiles that need more than 500 milliseconds query processing. As we can see, for AID* the entire set of tiles up-to level 20 can be queried within 500 milliseconds. In AID*, each time a non-image tile is requested, they are fetched from a R* index of the input data. For deeper levels these requests fetch very small amount of data making the query time extremely small. For AID on the other hand, for a shallow tile requests, the server needs to find its parent data tile by climbing up the pyramid. Each of these data tiles are almost of size θ, making it computationally heavy. This is primarily the reason for AID having so many tiles taking more than 500 ms to execute the query.

In order to measure how the visualization query responds to an extremely fast zoom-in or zoom-out actions we generated the entire hierarchy of tiles of subsequent zoom levels together. For example if a user zooms from level 15 to level 20 very fast, the children, grandchildren of tile 15 to 20 were requested simultaneously and must be generated in real time. Figure 11(a) shows the results for such actions. We chose data tiles, since image tiles are fetched in constant time. As we can see though, AID took roughly 3500 millisecs for zooming into 10 levels together, AID* could competently keep it around 500 millisecs.

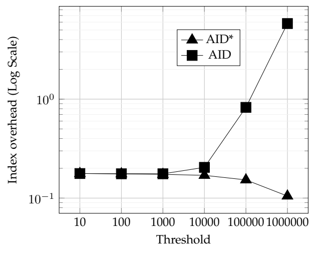
Another scenario could be when multiple users want to visualize a dataset simultaneously. To address more complex operation, we expect each of these users to zoom-in upto the deepest level. Figure 13(b) shows the number of users varying from 10 to 1, while for each of these tiles, each user zooms-in upto level 20. As we can see though AID increases exponentially with increase in tile numbers, AID* remains almost constant. The reason for AID* having almost a constant time for visualizing these tiles is because AID* spends most of the time in reading the files, but do not spend much time in processing, so parallelism makes it more efficient. On the other hand, AID has a complicated process of reading really big files and generating images for each of the shape in the .csv files. Hence adding more tiles to that process adds more to the processing and hence adds more time.

7.6 Threshold

In this section, we vary the threshold (θ) from 10 KB to 1 GB and measure the indexing time, the index overhead, and query processing times for different thresholds. Figure 14 shows the results of this experiment on the ALL NODES dataset. Since HadoopViz does not use a parameter θ, we don't show it in our figure.

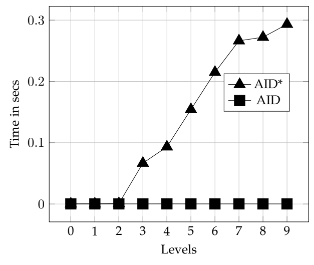


(a) Time Vs Threshold

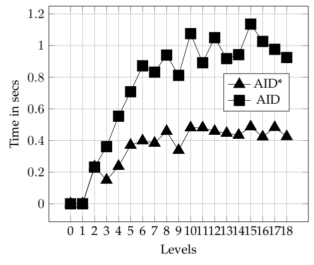


(b) Index Overhead Vs Input Size

Figure 14: Effect of changing threshold on index construction time and number of tiles generated



(a) Vis query performance with θ 10



(b) Vis query performance with θ 1000000

Figure 15: Effect of changing threshold on visualization query

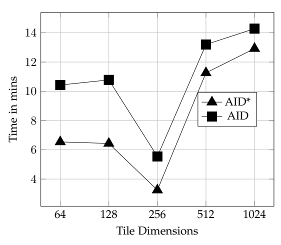
From Figure 14 a), we can simply say that increasing the threshold θ reduces the indexing time as well as the number of tiles. But the index overhead of AID increases significantly as depicted in Figure 14 b). As we increase the threshold, we are making image tile generation more restrictive. This results in less image tile generation, which also results in reducing the index construction time. Besides, it results in more records in a single data tile for AID, which although reduces the number of data tiles, increases the individual size of the files. This causes a drastic increase in index overhead in AID. Since AID* generates no data tiles, the index construction time of AID* is significantly low from the very beginning. Finding the optimal value of θ depends on how the users weigh the indexing time and query processing time.

In figure 15 we provide the results of querying AID, AID*. In figure 15 a), a low threshold results in most of the tiles being image tiles and hence for AID, all the tiles could be retrieved in constant time. Despite, AID and AID* generating same number image tiles, AID* star experiences a growth in time for visualization query. Unlike AID, in AID* for each empty, data or shallow tile requested by the user,it needs to refer to its R*-index to return the appropriate tiles. This phenomenon caused the visualization query time to grow in figure 15 a). However, it never crossed 500 millisecs. It should also be noted that a low threshold of 10 KB generates almost as same number of tiles as HadoopViz, making it impossible to scale upto level 20. On the other hand, figure 15 b), the threshold being as high as 1 GB not only makes the image tiles less in number, but also makes each of the data tiles of AID, as big as 1 GB. Such big data tiles are computationally exhaustive to be generated on fly. Hence the visualization query of AID reaches almost 1.2 secs for many tiles. AID* being devoid of any data tiles, does not encounter any such issues and successfully keeps the visualization query time less than 500 millisecs.

7.7 Tile Dimension

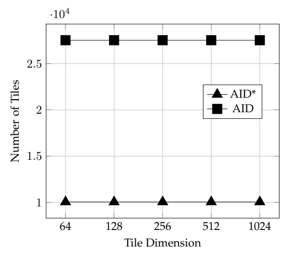
This section focuses on the effect of tile dimensions of the image tiles on the performance of the AID index. It emphasizes on finding an optimal tile dimension that 1) does not take too long in index construction 2) is not too big to be optimally queried by the visualization server

In this experiment, we used five tile dimensions



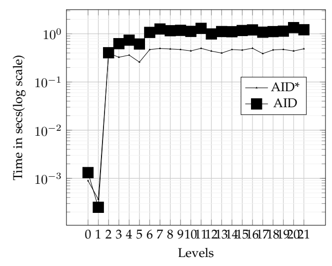
m

(a) Time Vs Tile Dimension

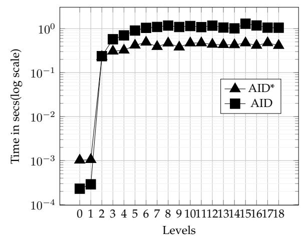


(b) Number of Tiles Vs Tile Dimension

Figure 16: Effect of tile dimension in index construction time and number of tiles generated



(a) Vis query for tile dimension 64



(b) Vis query for tile dimension 1024

Figure 17: Visualization query performance with changing tile dimension

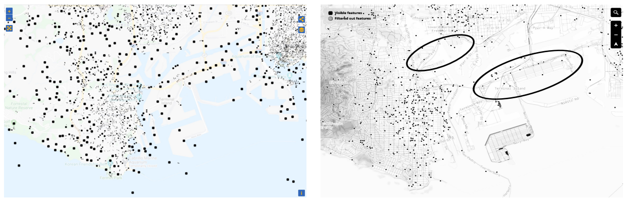
In Figure 16 we are conducting the experiments to measure the index construction time and number of tiles generated in the index construction phase. It should be noted that as we move up in tile dimension, we construct one less level to keep the same resolution. So this is a trade-off between many small tiles, increasing the number of files in the index or some small number of tiles containing a lot more information. The reduction in the number of levels does not affect the time though as we see in figure 16 b). This is because for levels as deep as 19-22, the tiles are mostly shallow or data for AID, and AID* has no extra image tiles so deep. Most image tiles are realized by level 14, as seen in earlier experiments. As figure 16 a) suggests, a tile dimension of 256×256 takes the least index construction time. This can be explained by the fact that in lower tile dimension, we have to construct more tiles to represent the same visual area. This increases the index construction time. On the other hand, for higher tiles dimension, since a single tile represent a big area, it might end up having very less info in one tile. As a result higher tile dimension has fewer empty tiles, which results in a longer index construction time. When we test these various tile dimension for visualization queries, the time taken to visualize the different tile dimensions are not very significantly different for 64×64 tiles and 1024×1024 tile as portrayed in figure 17.

7.8 Qualitative Comparison

This section compares the visual exploration provided by AID* with two different commercial systems referred in the paper as System A and System B.

7.8.1 System A

In this experiment, we visualize the `Twitter` dataset with 20 million records, on both AID* and System A. Similar to AID*, System A preprocesses the data according to a given number of zoom levels and produces vector tiles that are then used by a visualization server to display on the map. The index construction time and index size in System A in comparison with other systems, are provided in figures 10(a) and 10(b).



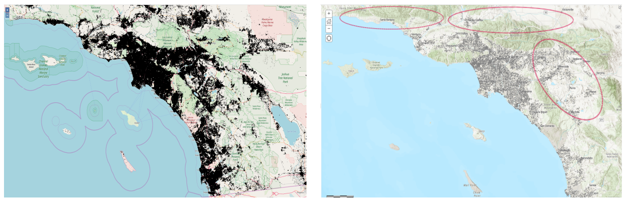
(a) AID* - Index construction in 11 minutes. Index size is 989.4 K. All records are visualized.
(b) System A - Index construction in 42 mins. Index size is 1 GB. Only a sample of the records is visualized

Figure 18: Comparison of exploring the `Tweets` dataset on AID* and System A

Figure 18 shows that AID* is able to visualize *all* the records which gives a superior user experience while System A chooses to sample the records and visualize only a few of them. Figure 18(b) highlights some of the areas that seem empty on System A while they do have data as shown on AID*.

7.8.2 System B

This system does not provide a detailed description of their index construction or tile generation method. However, they provide a web-hosted visualization of the Microsoft's Building Footprint data which covers the entire United States. We took the same dataset and generated AID* index and visualized it on our server. The first observation in this visualization is that System B does not produce any visualization until level 8. It is a plain blank map until then. We start seeing data points from level 9 onward and as Figure 19 suggests, AID* provides a more detailed visual points as compared to System B. We also noticed a significant delay



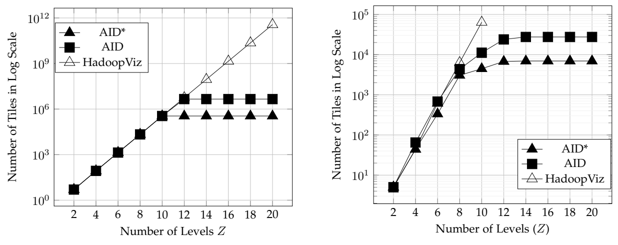
(a) AID* visualizes all records
(b) System B visualizes sample records

Figure 19: Comparison of exploring the Microsoft `Buildings` dataset on AID* and System B

in the response time of System B which goes up-to five seconds. However, we could not run a formal experiment on the response time due to the lack of access to the back-end server to run a controlled experiment.

7.9 Cost Model Verification

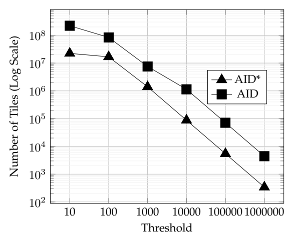
This section experimentally verifies the the accuracy of the proposed cost model. Figure 20 measures the index size in terms of number of files. We use equations 8, 7, and 9 to compute the number of image, data, and shallow tiles. Then, we compute the estimated size of the three indexes based on which tiles are materialized in each index. We further generated a synthetic uniform data as described in [26] to test the model. Since the results were exactly the same as figure 20(b), we do not show them as a separate figure. We also generate the actual index using the three method to compute the actual sizes of the three indexes. From the first glance, the trends of the three lines look very similar which verifies that the growth of the estimated index size is correct. While HadoopViz could not finish on time, the estimated cost model projects the exponential growth which explains why it fails.



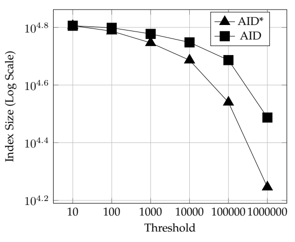
(a) Theoretical index size
(b) Actual index size

Figure 20: Verification of the cost model with Z

We still noticed two discrepancies between the estimated and actual sizes. First, the AID and AID* indexes stabilize at around 10^6 tiles based on the cost model in Figure 20(a) while they stabilize at around 10^4 tiles in the real experiment in Figure 20(b). This is a result of ignoring the empty tiles in our analysis which results in an over estimation of the index size. Second, the estimated cost model has an abrupt change at levels 10 and 12 for AID* and AID, respectively while in reality the change is gradual. This is a direct result of the uniformity assumption that causes all data tiles to appear at one level while in reality they appear gradually based on the data distribution and skewness. Despite these discrepancies, the cost model is still very effective in explaining the behavior of the experiments.



(a) Theoretical Number of Tiles



(b) Actual Number of Tiles

Figure 21: Verification of the cost model with θ

Figure 21 illustrates the effect of the threshold (θ) on the index size based on the cost model and reality. Both plots in Figure 21 show a steady decline in the number of tiles for both AID and AID* with increasing threshold as it is expected. Due to skewness of data at a very low threshold of 10, AID and AID* generate same amount of tiles in reality. But according to the cost model, the data tiles are all on level 12 for a threshold of 10. Hence the difference in index size of AID and AID* in Figure 21(a). But we see a similar trend in the decline of index size with increasing threshold in both the cost model and reality. As we increase θ, the level that contains most data tiles (z_D) decreases. Once z_D becomes smaller than Z, data tiles start to appear. As we increase θ more, the number of data tiles will start to decrease.

8 Conclusion

This paper addresses the problem of interactive exploration of big spatial data. It defines five requirements for a successful system for this problem and shows that existing systems do not support all five of them. Then, we introduced AID and AID* indexes which successfully satisfy the five requirements leading a successful interactive exploratory system. We also defined the visualization query interface that uses the proposed indexes to answer any visualization query in less than 500 milliseconds. Finally, we provided an extensive experimental evaluation on up-to a terabyte of data to show the scalability of the proposed solution as compared to baselines. This paper opens many future research directions in the area of exploratory analysis of big spatial data such as further tuning the index based on other user requirements such as the desired index size, indexing time, or the number of generated files.

Acknowledgements

This work is supported in part by the National Science Foundation (NSF) under grants IIS-1954644, IIS-1838222 and CNS-1924694 and by Agriculture and Food Research Initiative Competitive Grant no. 2019-67022-29696 from the USDA National Institute of Food and Agriculture

References

[1] L. Lins *et al.*, "Nanocubes for Real-Time Explorations of Spatiotemporal Datasets," in *TVCG*, 2013.
[2] "OpenStreetMap disk usage," 2019, https://wiki.openstreetmap.org/wiki/Tile_disk_usage.
[3] Z. Liu *et al.*, "The Effects of Interactive Latency on Exploratory Visual Analysis," *TVCG*, vol. 20, no. 12, pp. 2122–2131, 2014.
[4] A. Eldawy *et al.*, "HadoopViz: A MapReduce Framework for Extensible Visualization of Big Spatial Data," in *ICDE*, 2016.
[5] J. Yu *et al.*, "GeoSparkViz: A Scalable Geospatial Data Visualization Framework in the Apache Spark Ecosystem," in *SSDBM*, 2018.
[6] "OpenStreetMap," 2019, https://www.openstreetmap.org/.
[7] "MapBox," 2019, https://www.mapbox.com/.
[8] "OmniSci," https://www.omnisci.com/.
[9] T. Mostak, "An Overview of MapD (Massively Parallel Database). Harvard Technical Report," 2015, http://geops.cga.harvard.edu/docs/mapd_overview.pdf.
[10] Y. Park *et al.*, "Visualization-aware Sampling for Very Large Databases," in *ICDE*, 2016, pp. 755–766.
[11] Z. Liu *et al.*, "*imMens*: Real-time Visual Querying of Big Data," vol. 32, no. 3, 2013, pp. 421–430.
[12] M. El-Hindi *et al.*, "VisTrees: Fast Indexes for Interactive Data Exploration," in *HILDA@SIGMOD*, 2016, p. 5.
[13] H. Thorvaldsdóttir *et al.*, "Integrative Genomics Viewer (IGV): High-performance Genomics Data Visualization and Exploration," vol. 14, no. 2, 2013, pp. 178–192.
[14] "Mapzen," 2017, https://mapzen.com/.
[15] R. Michael *et al.*, "An Analytic Data Engine for Visualization in Tableau," in *SIGMOD*, 2011, pp. 1185–1194.
[16] A. Eldawy *et al.*, "SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data," in *ICDE*, 2015, pp. 1585–1596.
[17] J. Jia *et al.*, "Towards Interactive Analytics and Visualization on One Billion Tweets," in *SIGSPATIAL*, 2016, pp. 85:1–85:4.
[18] G. Planthaber *et al.*, "EarthDB: scalable analysis of MODIS data using SciDB," in *BigSpatial*, 2012, pp. 11–19.
[19] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, 1984.
[20] J. Yu *et al.*, "Geosparkviz in action: A data system with built-in support for geospatial visualization," in *ICDE*, Los Alamitos, CA, USA, apr 2019, pp. 1992–1995.
[21] "OpenLayers," 2019, https://openlayers.org/.
[22] S. Ghosh *et al.*, "AID: An Adaptive Image Data Index for Interactive Multilevel Visualization (Poster)," in *ICDE*, 2019.
[23] H. Chasparis *et al.*, "Experimental evaluation of selectivity estimation on big spatial data," in *GeoRich@SIGMOD*, 2017, pp. 8:1–8:6.
[24] A. Belussi *et al.*, "Estimating the selectivity of spatial queries using the 'correlation' fractal dimension," in *VLDB*, 1995, pp. 299–310.
[25] ——, "Skewness-based partitioning in spatialhadoop," *ISPRS Int. J. Geo-Information*, vol. 9, no. 4, 2020.
[26] T. Vu, S. Migliorini, A. Eldawy, and A. Bulussi, "Spatial data generators," in *SpatialGems@SIGSPATIAL*, 2019, p. 7.

Saheli Ghosh received her Masters in Technology (MTech) in Computer Science from WBUT, India in 2014. She is currently pursuing her PhD in University of California Riverside. Her research interests revolves around big data, primarily focusing on geo-spatial data and spatial query processing.

Ahmed Eldawy is an assistant professor at University of California, Riverside. He completed his PhD from University of Minnesota in 2015. He is the primary contributor and creator of SpatialHadoop. Big data management and spatial data processing are his primary research interests.