UNIVERSITY OF CALIFORNIA
RIVERSIDE

Interactive Visual Exploration of Big Spatial Data

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Saheli Ghosh

September 2020

The Dissertation of Saheli Ghosh is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

These years of my PhD at UCR have been one of the most formative experiences in my life. This wasn't easy or smooth. But this has been one of the greatest learning experience of my life. This experience has challenge me every step of the way and I am coming out as a completely different person, hopefully a better version of myself. Thankfully, I am surrounded by people who were there with me through my failures and success and supported me through it. This thesis, while my own work, would have been impossible without the support of these people around me.

I would like to thank my advisor, Prof. Ahmed Eldawy. He has been a great teacher, mentor and even saviour on many occasions. I have learnt things from him that is going to stay with me for the rest of my life. To the rest of my committee: Prof. Vassilis Tsotras, Prof. Vagelis Papalexakis and Prof. Sergio Rey who took time to listen to me, guide me and question me, so that I could find the right direction in my work, thank you.

I would love to thank my parents and Anubhab Sahin, who encouraged me to pursue this path, to begin with and for having faith in me that I could do it.

I would love to thank Geoffrey Logan, for putting up with me and my scary schedule, especially during the last few months before my defense. The global pandemic did not make it any easier and I am grateful that I had you to push me through this.

To my friends Unnati Rao and Priyanka Gandhi, who had to deal with all my disappointments, anger, excitement and tantrums for every failure or success in the past 5 years, thank you.

To Dr. Jason Ott, Dr. Pravallika Devineni, Bashar Romanous, Ravdeep Pasricha, Dr. Shailendra Singh, Dr. Ahmad Darki and Kittipat Patrick Apicharttrisorn, thank you for being such amazing friends and just being there whenever I needed you.

To Sudarshana Hore, Sayantika Mukherjee, Kuntal Chowdhury and Gouri Chakrabarty, thanks for reminding me of my roots and being my family away from home.

To Payas Rajan, thanks for teaching me how to read papers and how to organize my own code.

I would like to thank my labmates, Tin Vu, Samriddhi Singla and Akil Sevim, for encouraging me when I felt low, for listening to my n number of presentations and helping to make it better, find bugs in my code and just for making my lab time so much fun.

I would like to thank Uday Singh Saini, for answering all my machine-learning related queries, while being his sarcastic self all throughout.

UCR's CS department has been great. I cannot name every single person to whom I am thankful to, but from the day I stepped into the department, starting from different professors whose classes I took or whom I just know from departmental gatherings to the staffs in our department, everyone has been a delight.

Previous publications were used to create this thesis, portions of them are reprinted using the following permissions.

This dissertation is dedicated to:

- All the women in science, most of whom at some point in their life have probably questioned their decision of pursuing science and have continued anyway, while inspiring millions of others to get into science.
- My parents who had faith in me and who let me chase my dreams without any conditions.

ABSTRACT OF THE DISSERTATION

Interactive Visual Exploration of Big Spatial Data

by

Saheli Ghosh

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2020
Prof. Ahmed Eldawy, Chairperson

Visual exploration has become an integral part of big spatial data management. These datasets vary widely in their size, coverage, and accuracy. Therefore, users need to assess these aspects of the data to choose the right dataset to use in their analysis. Unfortunately, all the publicly available repositories for geospatial datasets provide a list of datasets with some information about them with no way to explore the datasets beforehand. 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. These indexes are either data indexes, which index data records, or image indexes, which index partially generated visualizations. Data indexes are fast to build but are only helpful for selecting a few records to visualize. On the other hand, image indexes are capable of visualizing big data in a short time but they are heavy on construction time and storage sizes. I introduced two new index structures, termed AID and AID*, that facilitate the visual exploration of an arbitrarily large number of big spatial datasets on a single machine. The indexes define multi-resolution fixed-size tiles on the input and classify 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. The 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.

Besides I also introduce UCR-Star, an application of the visualization engine powered by AID*, that is capable of hosting hundreds of thousands of geospatial datasets that a user can explore visually to judge their quality before even downloading them. It provides a web interface geared towards database researchers to understand how the index internally works. It provides a comparison interface where users can see side-by-side how two versions of the system work with the ability to customize each of them separately. Finally, the interface reports the response time of the indexes for a quantitative comparison.

On the other hand, the growth and rise of geospatial data, in the present world, with the ever increasing need to explore the geospatial data, many big data repositories were launched to host hundreds of thousands of datasets. In particular, UCR-Star focuses mainly on geospatial data and provides an interactive web interface to explore the metadata and contents of big geospatial data using a map interface. The primary target of UCR-

Star is the ease of visually exploring these spatial datasets without having to actually download terabytes of data. However, with hundreds of thousands of datasets in these repositories, users get lost and cannot find the datasets that might be useful to them. To address these problems, I worked on the first recommendation system for geospatial data exploration. We first define three recommendation tasks that could be helpful to users, namely, recommending i) a subset within a dataset, ii) a way to visualize the data, and iii) recommending an entire dataset. We then generalize them into one general visualization recommendation problem and we propose solutions based on collaborative filtering, tensor decomposition, and graph convolutional networks. We run experiments on real datasets from UCR-Star and show the effectiveness of the proposed solutions to solve the visualization recommendation problem.

Contents

List of Figures **xiv**

List of Tables **xvi**

1 Introduction **1**

2 Background & Related Work **8**
2.0.1 Geospatial Dataset 8
2.0.2 Visualization 9
2.0.3 Web-Map Visualization 9
2.0.4 Multilevel Image Index of Spatial Data 10
2.0.5 A data index for multilevel visualization 12
2.0.6 Single Machine Visual Exploration Systems 13
2.0.7 Distributed System Visualizations 14
2.0.8 Recommendation applications 16
2.0.9 Recommendation systems on Geospatial data 16
2.0.10 Recommendation system on visualization 17
2.0.11 Recommendation Algorithm 17
2.0.12 UCR-Star Logs 22
2.0.13 Precision Recall Curve 23

3 AID and AID*: A Spatial Index for Visual Exploration of Geo-Spatial Data **25**
3.1 Introduction 25
3.2 Overview 28
3.2.1 Offline Index Construction 29
3.2.2 Real-Time Query Processing 30
3.3 Preliminaries and Problem Definition 30
3.4 Index Construction 33
3.4.1 Index Layout 34
3.4.2 Index Building 37
3.4.3 Analysis of Index Construction 42

3.5 Visualization Query . . . 45
3.5.1 Query Definition . . . 46
3.5.2 Query Processing . . . 47
3.6 Experiments . . . 49
3.6.1 Experimental Setup . . . 50
3.6.2 Scalability and Platform Independence . . . 51
3.6.3 Index Construction Time . . . 52
3.6.4 Index size . . . 55
3.6.5 Visualization Query . . . 57
3.6.6 Threshold . . . 61
3.6.7 Tile Dimension . . . 64
3.6.8 Qualitative Comparison . . . 66
3.6.9 Cost Model Verification . . . 68
3.7 Conclusion . . . 71

4 An Application of UCR Star's Visualization Engine 73
4.1 Introduction . . . 73
4.2 Data Processing using AID and AID* . . . 75
4.3 Visual Exploration . . . 76
4.4 Web Interface . . . 77
4.4.1 Distribution of On-demand Tiles . . . 79
4.4.2 Threshold (θ) . . . 79
4.4.3 AID and AID* . . . 81

5 Recommendation on a Public Repository for Geospatial Exploration 82
5.1 Introduction . . . 82
5.2 Problem Definition . . . 86
5.2.1 Input . . . 87
5.2.2 Problem Formulation . . . 88
5.2.3 Output . . . 91
5.3 Proposed Visualization Recommendation Model . . . 91
5.3.1 Data Preprocessing . . . 92
5.3.2 Collaborative Filtering (CF) . . . 95
5.3.3 RESCAL . . . 97
5.3.4 Graph Convolutional Networks (GCN) . . . 99
5.4 Experiments . . . 101
5.4.1 Experimental Setup . . . 101
5.4.2 Evaluation Metrics . . . 102
5.4.3 Data Processing Phase . . . 104
5.4.4 Collaborative Filtering (CF) . . . 106
5.4.5 RESCAL . . . 107
5.4.6 Graph Convolution Network (GCN) . . . 108
5.4.7 Tuning the test cases . . . 109
5.5 Dataset Recommendation . . . 110
5.5.1 Experimental Results of the Models for Dataset Recommendation . 110

5.6 RESCAL: with and without definite negative relations 111
5.6.1 Tuning the test cases for RESCAL 112

6 Future Scope and Conclusion **114**

Bibliography **117**

List of Figures

1.1 Data.gov lists around 140,000 geospatial datasets in a textual form providing no visual insights 2
1.2 UCR-Star displaying Chicago Crime dataset 4
1.3 Scatter plot of the NYC Taxi data which reveals the noisy nature of this dataset 5
1.4 Scatter plot of the NYC Taxi data which reveals the noisy nature of this dataset 7

2.1 A traditional image index for multilevel visualization with three zoom levels 11

3.1 System overview 29
3.2 Examples of adaptive indexing with different threshold values 33
3.3 Tile classification example 34
3.4 Tuning of the parameter θ 35
3.5 Examples of the visualization query 48
3.6 Scalability of AID and AID* on Hadoop and Spark 52
3.7 Index construction time 53
3.8 Various system comparisons in terms of index construction time and index sizes. 54
3.9 Index size 56
3.10 Visualization query processing time 58
3.11 Visualization query time with parallel requests 59
3.12 Effect of changing threshold on index construction time and number of tiles generated 62
3.13 Effect of changing threshold on visualization query 63
3.14 Effect of tile dimension in index construction time and number of tiles generated 64
3.15 Visualization query performance with changing tile dimension 65
3.16 Comparison of exploring the `Tweets` dataset on AID* and System A 67
3.17 Comparison of exploring the `Buildings` dataset on AID* and System A . 68
3.18 Comparison of exploring the Microsoft `Buildings` dataset on AID* and System B 69

3.19 Effect of changing zoom levels on the tile numbers in proposed cost model Vs. actual number of tiles 70
3.20 Effect of changing threshold on tile numbers in the proposed cost model Vs the actual number of tiles 71

4.1 UCR-Star visualization engine showcasing the web interface and the effects of different threshold 78
4.2 UCR-Star showcasing static image tiles (black) and tiles generated on-the fly (red) 80

5.1 Figure A represents user's exploration pattern on a dataset, Figure B shows the recommendation based on Figure A where a user is given direction to find similar location for the dataset 83
5.2 Collaborative Filtering Recommending Tile-IDs 96
5.3 RESCAL Matrix Representation 98
5.4 Nodes (Tile-IDs) in GCN getting trained by neighboring nodes 100
5.5 Data processing time for CF, RESCAL and GCN 105
5.6 PR curve for Collaborative Filtering (CF) 106
5.7 PR curve for RESCAL 107
5.8 PR curve for GCN 108
5.9 PR curve showing results of increasing negative edges in test data for GCN 109
5.10 PR curve for CF, RESCAL and GCN on Multiple Dataset 110
5.11 PR curve for RESCAL's two versions 112
5.12 PR curve showing results of increasing negative relations in test cases for RESCAL 113

List of Tables

2.1 Related work based on the five requirements 15

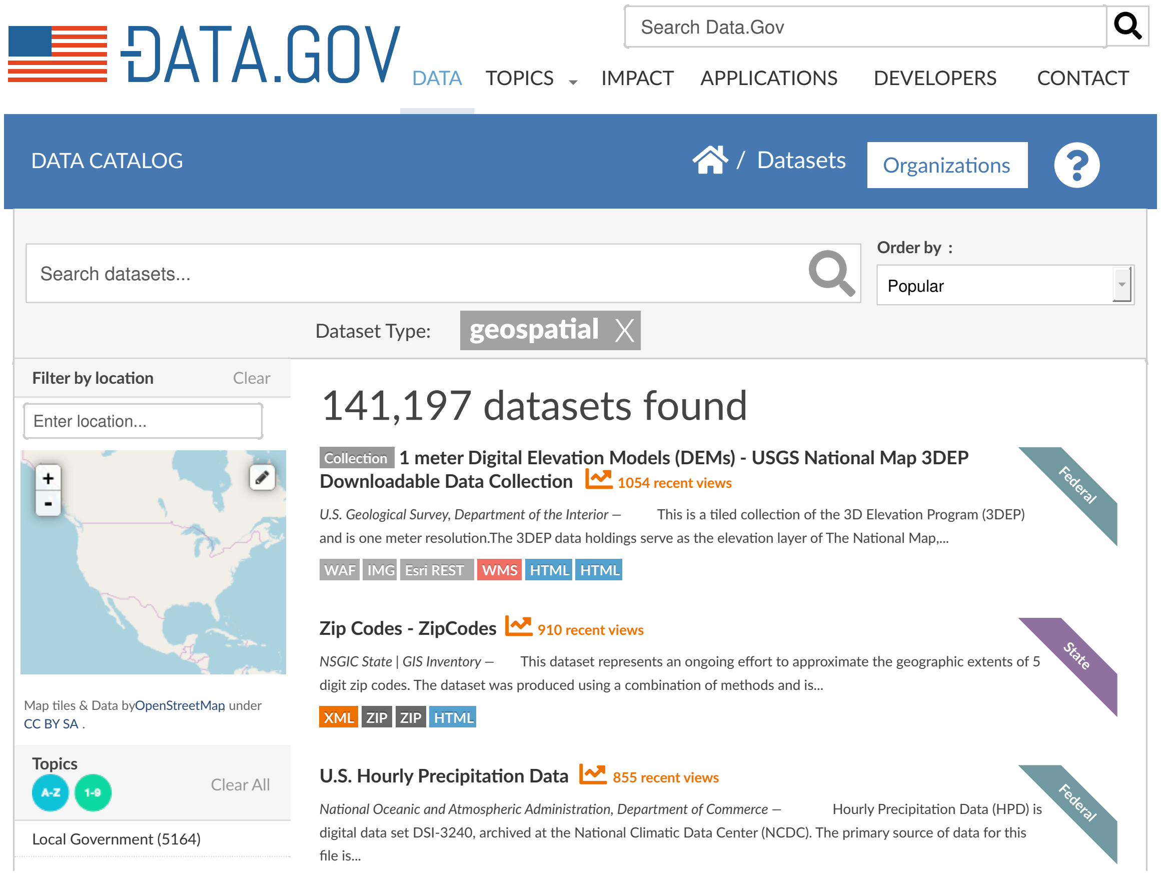
3.1 Input Datasets 50
3.2 Parameters and default values 50

5.1 Training and test datasets 102

Chapter 1

Introduction

Geospatial data has seen a tremendous growth in recent years. Data collected from satellites [28, 46], autonomous cars, IoT sensors, smart phones, 3D microscopes [24], social networks [49], etc all contribute to the growth of spatial data. On the other hand, there is a trend to make these spatial data open sourced. These open data often published by the governments, non-profits, public sectors, banks, etc constitute hundreds of thousands of datasets. For example, Data.gov [8] of US Federal government hosts more than 140,000 geospatial datasets. Other examples include data.gov.uk [7], World Bank Open Data [55], and United Nations Open Data [50]. In order to browse these repositories, existing repositories provide users with download links, giving no concrete information about the kind of data these datasets contain, or the coverage or quality of the datasets. As a result, users are flooded with thousands of such datasets, without an easy way of finding which dataset/s satisfy their requirements. For example, Figure 1.1 shows how Data.gov lists geospatial datasets in a mere textual form with insight about these datasets. Users often


DATA.GOV

Search Data.Gov

DATA TOPICS IMPACT APPLICATIONS DEVELOPERS CONTACT

DATA CATALOG / Datasets Organizations

Search datasets...

Order by : Popular

Dataset Type: geospatial X

Filter by location Clear

Enter location...

Map tiles & Data by OpenStreetMap under CC BY SA .

Topics A-Z 1-9 Clear All

Local Government (5164)

141,197 datasets found

Collection 1 meter Digital Elevation Models (DEMs) - USGS National Map 3DEP Downloadable Data Collection 1054 recent views

Federal

U.S. Geological Survey, Department of the Interior — This is a tiled collection of the 3D Elevation Program (3DEP) and is one meter resolution.The 3DEP data holdings serve as the elevation layer of The National Map,...

WAF IMG Esri REST WMS HTML HTML

Zip Codes - ZipCodes 910 recent views

State

NSGIC State | GIS Inventory — This dataset represents an ongoing effort to approximate the geographic extents of 5 digit zip codes. The dataset was produced using a combination of methods and is...

XML ZIP ZIP HTML

U.S. Hourly Precipitation Data 855 recent views

Federal

National Oceanic and Atmospheric Administration, Department of Commerce — Hourly Precipitation Data (HPD) is digital data set DSI-3240, archived at the National Climatic Data Center (NCDC). The primary source of data for this file is...

Figure 1.1: Data.gov lists around 140,000 geospatial datasets in a textual form providing no visual insights

end up guessing the usefulness of a dataset by their brief description in the repositories, downloading terabytes of data, finding a suitable tool to process these data and despite all these, they still have to discard most of the data, finding a very small portion or none of it useful for their purpose. This leads to hundreds of hours wasteful usage of precious work time.

However, it makes more sense to visualize geospatial data on a map for a better grip of the data. In the human brain, the visual cortex occupies around one third of the cerebral cortex [51] which shows the importance of visuals to the human brain. Visual exploration empowers a wide range of analytics by better utilizing that part of the brain which otherwise is not fully utilized. Sonoma Ecology Center once quoted "Maps are like campfires, everyone gathers around them, because they allow people to understand complex issues at a glance, and find agreement about how to help the land."

Following this logic, an alternative method that would be to able to visualize these geospatial data in an open repository, can clearly indicate the coverage, accuracy and quality of the data as portrayed in figure 1.2. Users are not only able to visualize the data, but are also able to interact with these data, by zooming into them for a detailed view or zooming out of them for a bird eyes view. Figure 1.3 depicts an 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 or, at least, be prepared to deal with the noise. Interested readers can refer to hundreds of additional examples at `https://star.cs.ucr.edu` which are all built by the method discussed in this article. While allowing users to browse through the metadata and contents of the datasets, it also opens new problems of finding not just a dataset, but a part of it that can be useful. For example, in figure 1.4, NYC Taxi pickup data and the road network, both are displaying the same location. A user who wants to work with the noisy NYC Taxi data might also be interested in the road network for the same location which could provide a means of cleaning the taxi data. It would be nice if a user who is exploring the NYC Taxi data is made aware of the availability of the road network dataset at the same location.

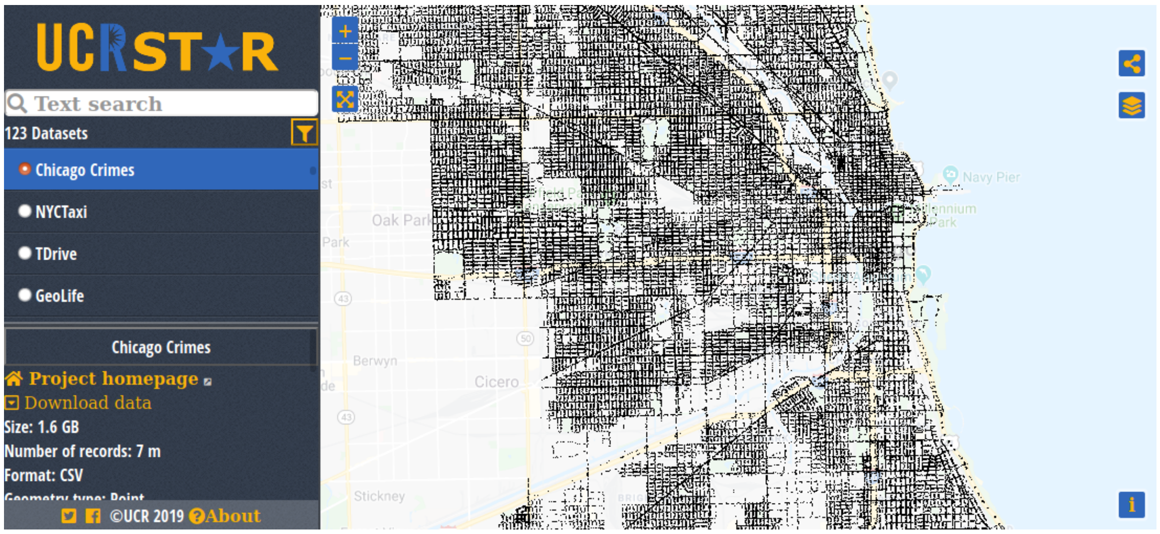
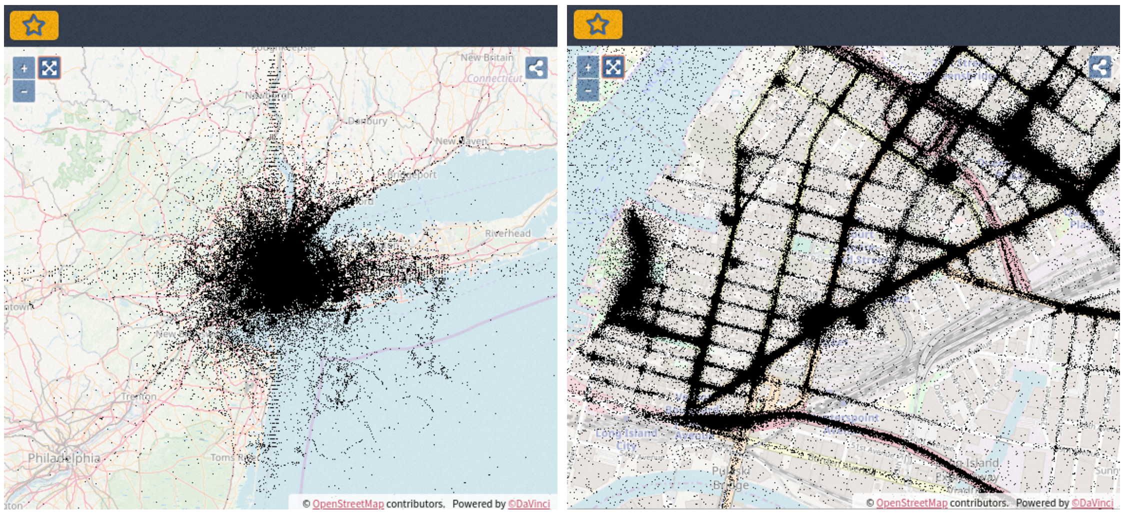


Figure 1.2: UCR-Star displaying Chicago Crime dataset

My thesis primarily revolves around enabling an open sourced visual repository with simple exploratory interface that allows users to quickly explore datasets before downloading them. It must satisfy the following few 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 added. (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. (3) **Individual records**: The users must be able to zoom in to see individual records to assess their quality and accuracy. (4) **Single machine**: The system must be hosted on a single machine for cost effectiveness. A cluster of machines 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 [27]. (6) **Recommenda-**



(a) Bird's eye view (b) Closer view

Figure 1.3: Scatter plot of the NYC Taxi data which reveals the noisy nature of this dataset

tion: The system must be able to guide a user through the visual exploration of the spatial datasets.

To satisfy the above six requirements, this thesis first 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).

In order to satisfy the requirement #6, I have built a recommendation system, that can guide a user to explore geospatial datasets. This recommendation is further divided into three subsections:

i) Recommending different datasets: when a user is visualizing dataset **A**, there might be a relevant dataset **B** in the same region. The system will suggest user to look into these datasets. For example in Figure 1.4 users can be recommended the road network dataset for that particular location in New York when they visualize NYC taxi data and vice versa.

ii) Recommending area of interest (AOI) within the same dataset: Sometimes the coverage of the dataset is so huge that users cannot really grasp the entire dataset.

iii) Recommending different visualizations for the same dataset: Typically, geospatial datasets can be visualized in different forms such as scatter plots, clustered points, choropleth, or heat maps. The most expressive visualization for a dataset could depend on which part of the dataset is visualized; for example, a heatmap could be preferred when the users zoom out while a scatter plot would be more helpful when users zoom in to see a few records. The system should be able to guide the user to switch between visualizations as they navigate the dataset.

In this dissertation we will discuss in details the algorithms and approaches adapted to achieve these above mentioned goals of our open sourced geospatial system. We start with a background and related work section, followed by different algorithms introduced in the system for an efficient processing of the data.

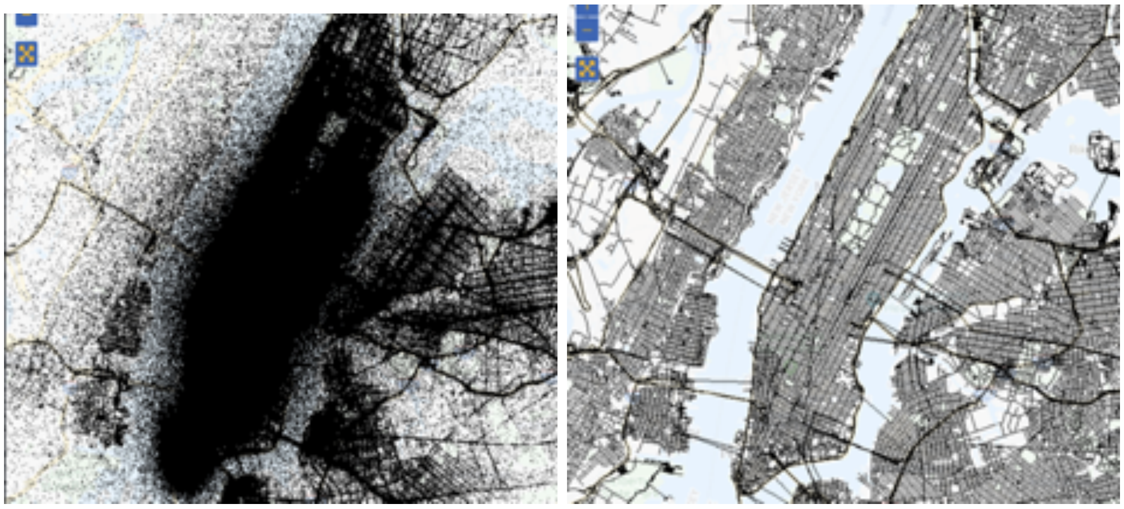

(a) NYC Taxi (b) Road Network

Figure 1.4: Scatter plot of the NYC Taxi data which reveals the noisy nature of this dataset

Chapter 2

Background & Related Work

The section provides some background knowledge on some existing spatial mechanism that is relevant to my work and helps in a clear understanding of my work. Additionally, it also focuses on some existing works on spatial data and their visualization. This provides more insights on my contribution as compared to other works and how it helps the community.

In the second part of this section we talk about the various recommendation systems, the existing algorithms, their benefits and how they can or cannot be used with our problem.

2.0.1 Geospatial Dataset

A geospatial dataset consists of a set of records where each record has a geospatial geometry attribute, e.g., point, line, or polygon. Examples of datasets include geo-tagged crimes, lake boundaries, or road segments.

2.0.2 Visualization

A visualization is a graphical representation of a geospatial dataset where each record is plotted according to its location on the map. One dataset can have multiple visualizations (views). For example, a simple scatter plot shows the distribution of crimes and a heatmap that shows their distribution.

2.0.3 Web-Map Visualization

A traditional multilevel visualization of geospatial data refers to the process of interactively exploring spatial datasets, that are plotted on top of a map. Users can zoom-in for a detailed view of these datasets, or zoom-out for a bird eye's view. The zoom levels are numbered like 0,1,2, ..., and most commercial systems provide a zoom level upto 20 where 0 denotes the top level and 25 being the deepest level. The images of the spatial data, that a user explore at different zoom levels, though often look like a single image, they are in fact multiple image tiles interleaved together to make it look like a single image. As we go deeper into the zoom levels the number of these image tiles increase.

Tile Based Visualization:

Scalable visualization of big geospatial data is done through tile-based visualization [11, 58, 12]. In this scheme, the whole world is covered with a quad-tree-like [44] pyramid structure which has one root tile at zoom level 0. As users zoom in, each tile is equally split into four tiles. Each tile has an ID (z, x, y) where $z \in [0, 19]$ is an integer zoom level, (x, y) is the position of the tile within that zoom level, $0 \leq x, y < 2^z$.

Non-Tile/Javascript Based Visualization

The standard and most widely used method for map visualization is through the various JavaScript map libraries including Google Maps and Open Layers. This technique relies on shipping the entire data to the web browser which handles all visualization. Unfortunately, this technique is not suitable for visualizing big data due to the limited capabilities of most web browsers.

2.0.4 Multilevel Image Index of Spatial Data

In order to create these tile-based visualization, a pyramidal image index is created that is based on a quad-tree [44] like structure. Following the quad-tree method, from one single image tile in a zoom-level, four more image tiles are generated in the subsequent level. In other words, an area that is captured in a single image tile at level z, is represented by 4 tiles in levels $z+1$. This contributes to an enhanced resolution with each deeper zoom level. It starts with a single image tile at level 0 which contains the entire image. Each of these tiles are recognized by a unique tile-id. A tile-id $< z - x - y >$ is often often a combination of zoom-level (z) and their x, y position on a 2-D grid system.

Figure 2.1 explains this even further with a multilevel visualization upto three levels. At level 0 we have a single image tile, which generates four tiles at level one, each of which generates four more image tiles at level two. The figure not only explains the pyramidal quad-tree like image index, it also suggests that with each zoom level the number of image tiles increase exponentially posing a threat for this mechanism while dealing with big spatial data.

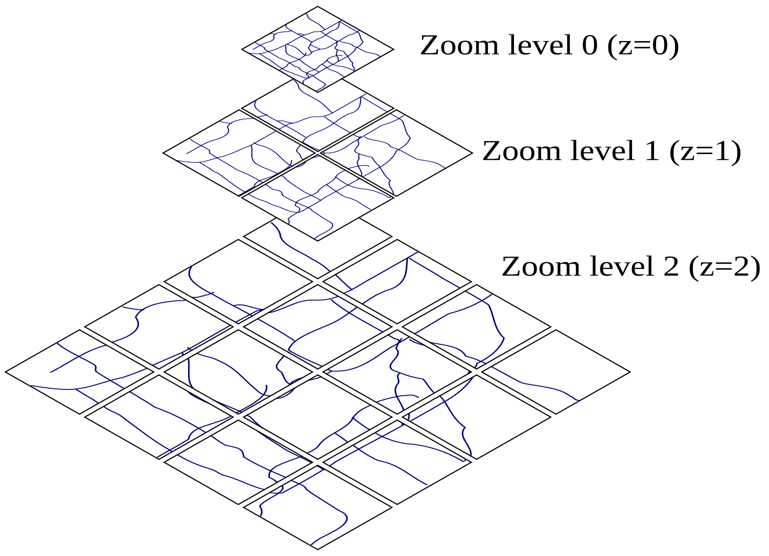


Figure 2.1: A traditional image index for multilevel visualization with three zoom levels

Usually, this entire pyramid is generated in an offline process and then loaded to a server. As a user browses through the map, a tile request defined by the tile-ids are sent to server, and the requested tiles are fetched to browser. This process of fetching these image tiles to the server is extremely fast and usually takes a few nano seconds to complete.

However, this method suffers from two main problems. i) The preprocessing time for generating the image index from vector data is extremely long. For a dataset of 100 GB, it can take hours to generate these indexes. ii) The other even bigger problem is the index sizes, or the number of files generated by this process. The number of files increases 4 times with each increasing zoom level. For example, the number of tiles at level 4 is $4^0 + 4^1 + 4^2 + 4^3 = 85$. But this quickly escalates to over 10 million tiles by the time we reach level 10.

2.0.5 A data index for multilevel visualization

A data index is a spatial index for accessing spatial data faster. It is a data structure that facilitates efficient access of spatial objects. It mainly relies on constructing a minimum bounding rectangle (MBR) around the object.

Definition 1 *A* ***minimum bounding rectangle (MBR)*** *is rectangular area on a geometric plane defined by* $\{x_{min}, y_{min}, x_{max}, y_{max}\}$ *which represents the minimum area needed to enclose another geometry (O) in the plane.*

- x_{min}, y_{min} is usually the lower left corner of the box
- x_{max}, y_{max} is usually the upper right corner of the box

MBR is one of the key components of creating these spatial indexes, which help in various spatial operations like range queries, K-nearest neighbours, etc. Various kinds of spatial indexes like R-tree [15], R-grove [53], etc are popular and useful for various use cases. One of the popular uses of these spatial data indexes is also creating a runtime multilevel visualization(as explained in Javascript based visualization earlier). However, not all javascript based visualization uses these data indexes, but the ones that do, are more capable of efficient handling of the data. Unlike the image indexes, the data indexes do not create any image tiles offline. On the contrary, they load the entire indexed data into the server. Each time a user requests a tile, the images are created at the very moment for visualization. The indexed data is extremely useful for making this process interactive.

This process is however limited by the amount of data in each tile. Processing too much data in runtime, can slow down the process resulting in slower response and prolonged loading of the image on the browser.

2.0.6 Single Machine Visual Exploration Systems

Single machine visual exploration can be defined as single machine implementation of visual exploration. As the name suggests, the work under this category aims at using single-machine algorithms and indexes to speed up the visual exploration of big spatial data. Most of these single machine systems use data indexes for faster processing. 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 [41], imMens [26], Nanocubes [25], VisTrees [9], and IGV [48] 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 defined in sec 1.

Another group of systems like Mapzen [30] and Tableau [31] focus on plotting data on top of an existing map. This visualization is done on the browser using JavaScript libraries, e.g., Google Maps. Due to browser limitations, these systems cannot scale to big datasets, resulting in poor interactivity and hence violating requirements #2 and #5. OmniSci [36], 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 in a system. To be able to use such systems efficiently for big data will not be very cost effective.

2.0.7 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 [11] and Shahed [10] employ Hadoop map-reduce and GeoSparkViz [58] uses Spark systems respectively, to preprocess the datasets to produce the desired visualization. They use a pyramidal quad-tree [44] image index to generate a multilevel visualization. However, the amount of image tiles increase exponentially with each level, creating about a million tiles in level 10. As a result of which HadoopViz does not scale above 10 levels and GeoSparkViz fails to scale above level 7. GeoSparkViz and Cloudberry [19] 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 [42] 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.

Table 2.1 summarizes, how the existing work satisfy the five requirements of interactive exploration. As shown in the table, only the proposed indexes (AID/AID*) satisfy the five requirements.

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

Work	Requirements				
	#1	#2	#3	#4	#5
VAS [41]			✓	✓	✓
imMens [26]			✓	✓	✓
Nanocubes [25]			✓	✓	✓
VisTrees [9]			✓	✓	✓
IGV [48]			✓	✓	✓
Mapzen [30]	✓			✓	
Tableau [31]	✓			✓	
OmniSci [36]			✓	✓	✓
HadoopViz [11]		✓		✓	✓
Shahed [10]		✓			
Cloudberry [19]		✓			✓
GeoSparkViz [58]		✓			✓
EarthDB [42]		✓			
AID & AID*	✓	✓	✓	✓	✓

Table 2.1: Related work based on the five requirements

(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. A detailed description of this process is described in chapter 3.

Moving on to our next problem of geo-spatial recommendation, the upcoming section provides some insights on the various recommendations techniques, algorithms, machine learning models as well as evaluation metric for the models.

2.0.8 Recommendation applications

There have been a lot of applications that use recommendation systems. Companies like Amazon [?], Netflix [?], and Spotify [?] have been mining user behaviors and recommending them products, movies, shows, and songs for years. Besides, some systems like Geolife [?], Facebook [?], Twitter [22] also use location-based social networks, to recommend users a variety of things including movies, products, and restaurants. This paper addresses a different problem of recommending datasets, or subsets of them, while lacking any personal user information.

2.0.9 Recommendation systems on Geospatial data

Most recommendation systems on geospatial data are inspired from their ancestors like Netflix or Amazon. LARS [23, 45] categorizes location-based recommendation into spatial users, spatial items, or spatial both. In [4] user's interests and preferences are used to recommend tourist hotspots on Greece. The work in [16], uses Twitter data to mine user's music choices to predict artist similarity and local trends. Other-location based systems [40, 43] use content-based recommendations where user choices and preferences are combined with location-based features like tags and features for recommendations. Another method is the Hyper Texts Induced Topic Search (HITS) [2] which identifies local experts and use their information to recommend places based on their area of expertise. Other work [6, 47] also combines spatial filtering with collaborative filtering to provide user recommendation. Unlike all these systems that recommend points-of-interest or other well-defined items, our system deals with recommending parts of dataset where the location can be anywhere in the space and not tied to defined points on the map.

2.0.10 Recommendation system on visualization

Another type of recommendation that is related to our work is visualization recommendation for datasets. Systems like VizRec [52], VizML [18] and others [14] emphasizes on patterns and trends to enable visual analysis. These can be generally classified as content-based systems since they rely primarily on the dataset patterns while our technique uses user behavior to make dataset and visualization recommendation.

2.0.11 Recommendation Algorithm

In general, recommendation algorithms can be divided into *content-based* and *collaborative-based* techniques. Content-based techniques rely on item information, e.g., genre or cast members, to recommend items based on their similarity. On the other hand, collaborative-based techniques rely on mining user behavior to find similarity between users or items.

Collaborative Filtering

Collaborative Filtering (CF) is a widely used and extremely popular recommendation algorithm. Like the name suggests, it is a method of filtering out items that might be liked by a user based on similarities of the user with other users or other items that a user likes. Though there are various types of collaborative filtering and the algorithm has evolved over time, on a very high level it is reliant on rating items based on users' historical preferences on collection of items. It can be broadly categorized into two types.

User Based Collaborative Filtering This is based on the idea that if two users have similar taste in items, an item liked by user A can be recommended to User B. For example, if user A and user B both like reading books, and both has rated Hamlet, Pride and Prejudice and Twilight highly. However user A also rated Harry Potter highly, which user B hasn't read yet. CF will recommend the Harry Potter to user B.

Item Based Collaborative Filtering Item based collaborative filtering [1] was proposed by Amazon in 1998. As the name suggests it is based on similarity between items than users. Here the model looks at how a user has rated various items and tries to draw similarities between items by their ratings. For example, if the model sees, the user usually rates comic books highly, it will recommend the user a comic book that is highly rated but not read by the user.

However, CF has some limitations such as the cold-start with fresh items and inclusion of side information from items which makes it not effective for some applications. Cold Start refers to the problem of learning when initially too few users/items are present as training data. In such cases, some recommendation algorithms might act poorly and some systems rely on content-based recommendation instead. In order to solve the problem we moved to RESCAL

RESCAL

RESCAL [34] , [35] is a three-way tensor factorization technique designed for knowledge graphs. It is very powerful in finding hidden relationship between entities. It was proposed as recommendation system for knowledge graphs to find relations between entities and then to recommend and predict links between entities [33, 21]. The input to RESCAL is a set of tuples in the form $\langle i, j, k\rangle$ that represents the relation k between two entities i and j. This makes it similar to the common RDF schema used in knowledge graphs. Assuming n entities and m relations, RESCAL represents these triples as a tensor (3D matrix) $\mathbf{X} \in \{0,1\}^{n\times n\times m}$, where $x_{ijk} = 1$ if the relation k holds for the entities i and j, otherwise, it is zero. RESCAL decomposes the tensor $\mathbf{X}$ into three components $\mathbf{E}$, $\mathbf{W}$, and $\mathbf{E}^T$ such that:

$$\mathbf{X} \approx \mathbf{E} \times \mathbf{W} \times \mathbf{E}^T \tag{2.1}$$

Where $\approx$ indicates the approximation with the tensor decomposition technique, $E \in \mathbb{R}^{k\times r}$ is a matrix that holds the relationship between n entities and r latent variables. r is a model parameter that controls the complexity of the model. $\mathbf{W} \in \mathbb{R}^{r\times r\times n}$ is a tensor that holds the relationship between latent variables for each relation. Finally, $\mathbf{E}^T$ is the matrix transpose of $\mathbf{E}$. Typically, r is much smaller than m which makes $\mathbf{W}$ much more condensed than the input $\mathbf{X}$.

Using this decomposition method, we can rewrite x_{ijk} as follows:

$$x_{ijk} = \mathbf{e}_i^T \times \mathbf{W}_k \times \mathbf{e}_j \tag{2.2}$$

$$= \sum_{a=1}^{r}\sum_{b=1}^{r} w_{abk} e_{ia} e_{jb} \tag{2.3}$$

where $\mathbf{e}_i^T$ is a row vector of size r that represents the latent variables of entity i, $\mathbf{W}_k$ is an $r \times r$ matrix that represents the relationship between latent variables for relation r, and $\mathbf{e}_j$ is a column vector that represents the latent variables for entity j. This allows us to estimate the relationship between any pair of entities i and j in a relation k.

However, RESCAL is limited to linear relationships and it fails to establish long range relationships. To address these issues, neural network plays a key role where a historical, complicated relationship between entities can be stored and used for recommendation. In particular, graph convoluational networks (GCN) have been used in recommendation for knowledge graphs and social networks [17, 56, 57].

Graph Convolution Network

Graph Convolutional Networks [20] is a powerful machine learning tool that uses a neural network architecture to find relationships between nodes in a graph. The primary idea behind GCN is to learn how an edge in a graph can be predicted from its two end-points (vertices). The idea is to introduce a latent representation to vertices such that tightly connected vertices will have similar latent variables. The learning process runs in several iterations where connected vertices exchange information at each iteration to update the latent variable representation until the model converges. For some applications of GCN [56], this knowledge involves features that describe the nodes alongside the connectivity of the nodes with other nodes. However in cases where the nodes are devoid of any features, each node is trained by the connectivity features from adjacent nodes.

Formally for a given graph $G = (V, E)$, the goal here is to learn a function of features (node embedding) which takes two inputs. 1) a matrix $\mathbf{X}$ of dimension $n \times d$ where

n is the number of nodes and d is the number of features per node; d is a configurable parameter. The value x_{ij} of $\mathbf{X}$ represents the feature j for the node i. These features can be an identity matrix in absence of any specific feature. The output is an $n \times d$ matrix which represents the features for the n nodes in the graph. 2) A matrix representation of the computational graph named A The output of GCN is an $n \times f$ feature matrix named $\mathbf{Z}$, where f is the number of features for each node i. Like other neural networks, GCN too can have hidden layers. If we use one layer, then the information is exchanged between direct neighbors at each iteration. The more layers we use, the more hops the information can travel at each iteration, i.e., exchange data with neighbors of neighbors and their neighbors as we go deeper. It is for us to decide how many layers to use based on the complexity of the application. Each layer acquires the knowledge from the output of the previous layer. Formally this can be defined as:

$$H^{(l+1)} = f\left(H^{(l)}, A\right), \tag{2.4}$$

where $H^{(0)} = X$ and $H^{(L)} = Z$ and Z is the graph level output and L is the total number of layers in the graph.

None of the above mentioned systems satisfy my requirements, because I have a repository that hosts multiple geospatial data from various sources. Users from various spheres of science and commerce visualize these datasets and each one of them has their own specific requirements. It is the datasets that are the most important factor here. Intuitively a dataset is mostly explored in a very specific way by multiple users.

What we propose in this dissertation is a recommendation system that tracks anonymous user behavior of exploring various geospatial datasets and recommends users various areas to explore based on previous user behavior. We basically try to mine different users pattern of exploring a specific dataset, their choices of switching from one dataset to other or from one visualization to other and use that information to guide a new user to explore UCR-Star. We have used UCR-Star logs to build this system, however this technique can be applied to any other system which has similar pattern of capturing user behavior. The next section talks briefly about UCR-Star logs.

2.0.12 UCR-Star Logs

UCR-Star [13] [`https://ucrstar.com`], is a public web-based repository for geospatial data which provides interactive tile-based visualization for hundreds of datasets. While users visualize the data, the browser sends requests for tiles to update the visualization [12]. These requests are stored in a log $L = \{r_i\}$. A request r_i is a tuple in the form $\langle t_i, d_i, v_i, z_i, x_i, y_i \rangle$, where $1 \leq i \leq |N|$, t_i is the request timestamp, d_i is the dataset ID, v_i is the visualization ID, (z_i, x_i, y_i) is the tile ID. Notice that these logs are naturally anonymous since UCR-Star is public and does not have access to any user information.

While I try different models for my recommendation, it becomes pertinent to find a way to evaluate these systems. The following section provides some insights on my evaluation method and why it is relevant to use this technique for our evaluation.

2.0.13 Precision Recall Curve

Precision-Recall Curve (PR-curve) has been used as our evaluation metric for testing the implemented model. Precision can be defined as the number of correct predictions by a model, while recall measures the relevance. Mathematically it can be defined as:

$$Precision = TP/(TP + FP) \tag{2.5}$$

where TP (FP) is the number of True (False) Positives.

Recall is used to measure the correct number of positive predictions out of all the positive predictions. Mathematically recall can be defined as:

$$Recall = TP/(TP + FN) \tag{2.6}$$

where FN is the number of False Negatives. When the model makes a prediction (or a score) above a certain threshold, we consider that to be a positive prediction, and below that to be a negative prediction. To generate PR-Curves multiple thresholds are selected (based on the score output by the model) and Precision and Recall are calculated for model predictions based on that threshold. Assuming the model outputs probabilities, selecting a threshold of say 0.9, might lead to a high precision score as most True positives will be labelled with a high scores than True Negatives, but this won't cover the entire spectrum of all Positive examples, just the ones that our model predicted with a high score, thus the recall will be low. Thus, lowering the threshold for positive prediction will increase recall, but can be at expense of precision. An ideal model will output a precision recall curve which is as high as possible while moving from left to right (increasing recall). The added benefit of Precision-Recall testing with a ratio of positive examples and negative examples is that if the PR curve drops as we test with an increasing number of negative examples when compared to positive examples, this generally is an indication that the model has a tendency to recommend negative results over positive results, and each positive prediction by the model at a lower threshold score might not be a good recommendation for the user. As long as the PR curve is high, this implies that model is learning latent representations for tiles that facilitate better prediction scores, thus using nearest neighbour embedding for making recommendations is a justified choice. The values of precision and recall both range from 0 to 1. In a precision-recall curve, precision is plotted on the x-axis and recall on the y-axis. A PR-curve of a model that bows towards the coordinates $(1, 1)$ eventually is regarded as an effective model. A straight horizontal line depicts precision proportional to the number of positive cases, denoting the model to be not useful for the cause. The values of precision and recall both range from 0 to 1. In a precision-recall curve, precision is plotted on the x-axis and recall on the y-axis. A PR-curve of a model that bows towards the coordinates $(1, 1)$ eventually is regarded as an effective model. A straight horizontal line depicts precision proportional to the number of positive cases, denoting the model to be not useful for the cause.

More details of this part of the work can be found in 5.

Chapter 3

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

3.1 Introduction

A usual approach for interactive big spatial data visualization is via multilevel images which put fixed-sized tiles into a pyramidal structure to provide visualizations at different zoom levels, e.g., Google Maps, Bing Maps and HadoopViz [11]. Since the number of tiles increases exponentially with each zoom level, an efficient indexing technique is required to store these tiles. For example, most web maps provide 18 zoom levels with up-to 90 billion tiles. Our aim through this chapter is to satisfy the first five requirements described in chapter 1.

Limitations of existing work: There are many existing systems for big spatial data visualization and exploration, however, all of them fail to satisfy one or more of the five requirements. HadoopViz [11] 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 [58] preloads data in the distributed memory of the Spark cluster which violates requirements #1 and #3. Web maps such as OpenStreetMap [38] rely on MapBox [29] to build and visualize data but they need to produce billions of image tiles that consume terabytes of disk space [39] which does not scale to accommodate a large number of datasets (fails requirement #1).OmniSci [36, 32] preloads the dataset to a GPU to provide fast access which makes it limited to the size of the GPU (fails requirement #4). This makes it expensive as it is dependent on GPU prices, even though it can be hosted in a single machine. Some systems rely on sampling as in VAS [41] and aggregation as in NanoCubes [25] to reduce the data size, thus, they fail requirement #3.

To satisfy the five requirements, this chapter 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.

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 chapter is organized as follows Section 3.2 gives a high-level overview of the index. Section 3.4 details the index construction process. Section 3.5 describes the interactive visualization query processing. Section 3.6 provides an extensive experimental evaluation of the proposed work.

3.2 Overview

Figure 3.1 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

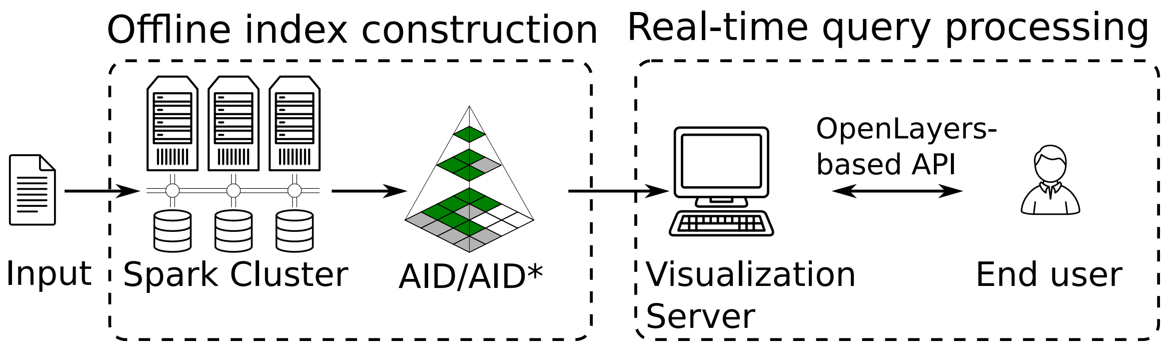


Figure 3.1: System overview

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.

3.2.1 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 3.2 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*).

3.2.2 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 [37] 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 3.5 will show the details on how the visualization query handles all these cases.

3.3 Preliminaries and Problem Definition

This section gives some preliminary definitions for the multilevel visualization problem. Figure 3.2 illustrates an example of the pyramid structure that we define below.

Definition 2 *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 3 *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 4 *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 $z \in [0, Z]$ 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 20 zoom-levels (Z), z will range from 0 to 19. In other words, the number of tiles increases* exponentially *with the number of zoom levels.*

Definition 5 *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:*

$$\begin{aligned}
t.BB.w &= \mathcal{P}.BB.w/2^z \\
t.BB.h &= \mathcal{P}.BB.h/2^z \\
t.BB.x_{min} &= \mathcal{P}.BB.x_{min} + t.BB.w \cdot x \\
t.BB.y_{min} &= \mathcal{P}.BB.y_{min} + t.BB.h \cdot y
\end{aligned} \tag{3.1}$$

Definition 6 *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 7 *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 8 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 [11] 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 [11] solves the multilevel visualization problem by pre-generating 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.

Conversely, this chapter proposes an alternative approach that builds an index, AID or AID*, that is much smaller in size and can still answer any visualization query within the specified time limit.

3.4 Index Construction

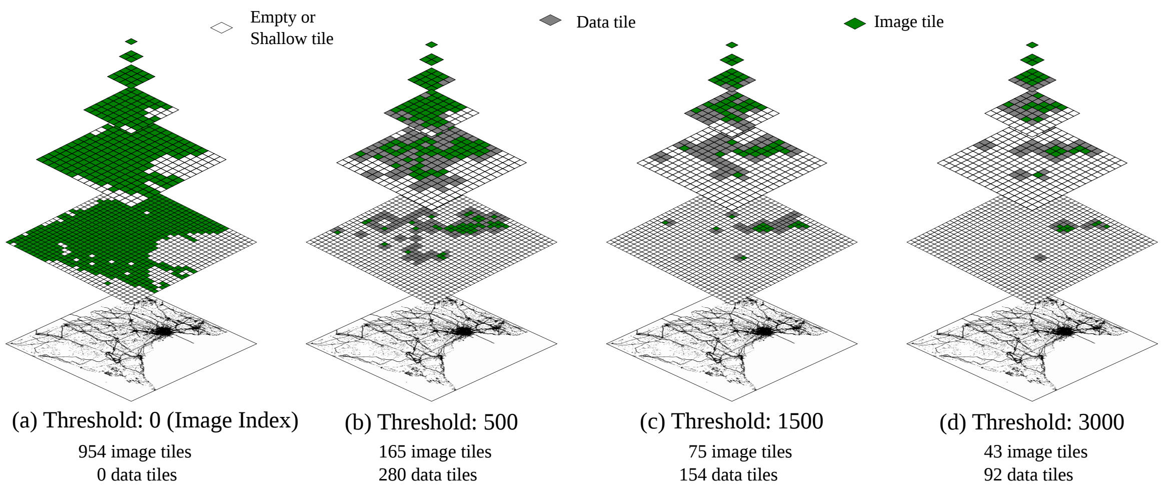


Figure 3.2: Examples of adaptive indexing with different threshold values

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. After the index is constructed, users can interactively visualize the data using the query processing described in Section 3.5.

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** [12], two types of tiles are materialized to disk, image and data. Image tiles contain prerendered im-

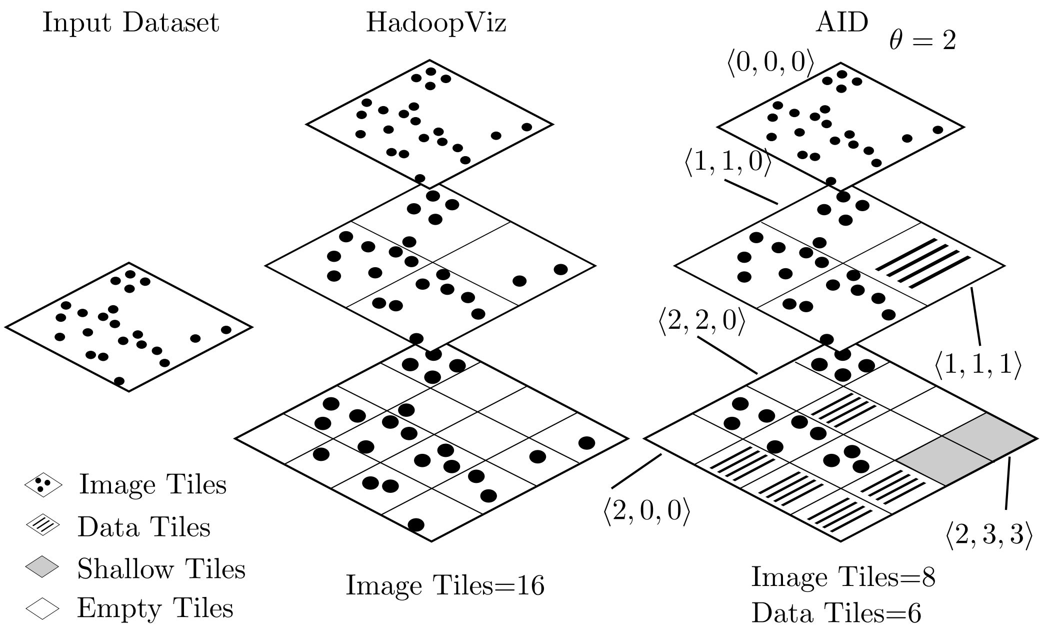


Figure 3.3: Tile classification example

ages 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 are not materialized. The rest of this section describes the index layout in Section 3.4.1. Section 3.4.2 details the process of building the index. Section 3.4.3 derives a theoretical analysis for the index construction phase.

3.4.1 Index Layout

The proposed index follows the multilevel pyramid layout similar to the one illustrated in Figure 3.2.

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 3.3 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 $\langle \ldots \rangle$ 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.

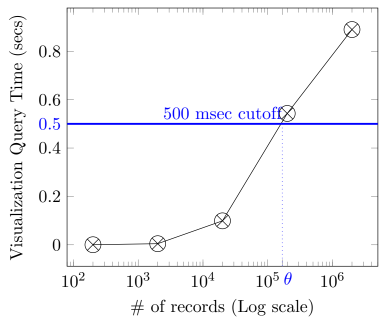


Figure 3.4: Tuning of the parameter θ

Definition 9 *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 8). 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 3.4 shows the results of this experiment on a point dataset where the 500 millisecond cutoff is reached at around 160,000 records. Normally, this tuning step would take around 2θ time, i.e., one second.

Definition 10 *A tile t is an* **image tile** *if it has a size $|t.F| > \theta$. In Figure 3.3, tile $\langle 1, 1, 0 \rangle$ is an image tile because it has five records while $\theta = 2$.*

Definition 11 *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 3.3, 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 12 *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 3.3, tile $\langle 2, 3, 3 \rangle$ is a shallow tile as it has one record while its parent is a data tile.

Definition 13 *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 [11], 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, $|\mathcal{P}| = (4^{Z-1} - 1)/3$, 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 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 of having no θ. The results of such low θ are explained in more details in section 3.6.6. The next part explains how the AID and AID* indexes are constructed.

3.4.2 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.

Data summarization

This first preprocessing phase summarizes the data to allow classifying the tiles based on their visualization cost. Based on the work in [5], 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

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
```

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 [5].

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 3.3, 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
11:                    and tile-class is data then
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 $\mathcal{P}$
```

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 [11], 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. If AID index is being constructed, a data tile is created. Otherwise, if AID* index is being constructed, no data tiles are created.

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 [11]. 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.

3.4.3 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:

$$N_z = N/4^z, z \in [0, Z] \tag{3.2}$$

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:

$$z_D = \lceil \log_4 (N/\theta) \rceil \approx \log_4 (N/\theta) \tag{3.3}$$

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:

$$D = \begin{cases} 0 & ; Z \leq z_D \\ 4^{z_D} & ; Z > z_D \end{cases} \tag{3.4}$$

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.

$$I = \sum_{z \in [0, Z_I]} 4^z = \frac{4^{Z_I} - 1}{4 - 1} \tag{3.5}$$

$$S = \sum_{z \in (z_D, Z)} 4^z = \frac{4^Z - 1}{4 - 1} - D - I \tag{3.6}$$

We approximate Equations 3.4, 3.5, and 3.6, by assuming $Z > z_D$ and removing the ceiling and floor functions. This results in the following approximations.

$$D \approx 4^{z_D} \approx N/\theta \tag{3.7}$$

$$I \approx \frac{4^{z_D} - 1}{4 - 1} \approx N/3\theta \tag{3.8}$$

$$S \approx 4^Z/3 - N/\theta - N/3\theta = \frac{1}{3}(4^Z - 4N/\theta) \tag{3.9}$$

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^2N/θ. (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 $T^2N/\theta + N$. (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 $T^2 4^Z/3$.

Among the above three indexes, the existing image index (HadoopViz) is the one that increases exponentially in terms of number of files and total size. 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 while in AID the shallow tiles are not stored in the index and AID* does not store data tiles alongside the shallow tiles making it even smaller.

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 [3] which is beyond the scope of this paper. In such a case, the biggest change in the cost model would be introduction of empty tiles(E). Since we assume, data is uniformly distributed, there are no empty tiles, however, for a non-uniform distribution, many regions will not have any data at all, which will contribute to creation of empty tiles. An extremely skewed data can generate a large number of such empty tiles. Hence in our algorithm, like shallow tiles, we make sure that empty tiles have no physical storage.

Non-uniform data also results 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. Overall due to the skewed nature of the data, which leads to the introduction of empty tiles, a non-uniform input reduces the total number of tiles, which in turn reduces the index size. However, the trend of the tile growth for the different algorithms remains similar as seen later in figure 3.19

3.5 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. While image tiles are pregenerated and materialized, data and shallow tiles need to be processed to visualize the other tiles that were not generated during the index construction phase. In the next part, we first define the visualization query and then we show how to answer it using the AID and AID* indexes.

Algorithm 3 Get a tile from the index

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

3.5.1 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 8). Notice that while the index might contain image or data tiles, the return value of the GetTile query is always an image to be displayed to the user. 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.

3.5.2 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 3.5. First, if the user requests a tile which is stored as an image tile (e.g., tile A in Figure 3.5), 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 5.

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 3.5) 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

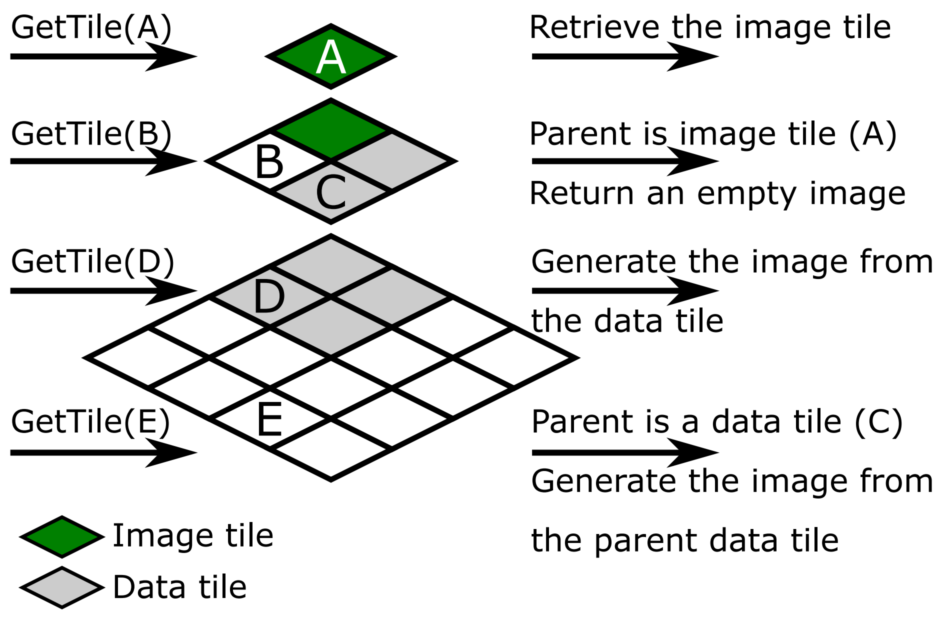


Figure 3.5: Examples of the visualization query

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 3.3 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 (from a data tile in AID or from the R*-tree in AID*) 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.

3.6 Experiments

This section aims at providing an extensive experimental evaluation of our proposed work under different circumstances and parameters. 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 **blue**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.

3.6.1 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 [37] while the back-end is implemented as a Jetty server in Java. Table 5.1 lists the datasets that we use in this work. The `All-Nodes` dataset is a point dataset extracted from OpenStreetMap and the `Buildings` dataset is the Microsoft's US Building Footprints. Unless otherwise mentioned, the experiments use the default parameter values listed in Table 3.2. We denote the threshold θ in terms of bytes by multiplying with the average record size.

Table 3.1: 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

Table 3.2: Parameters and default values

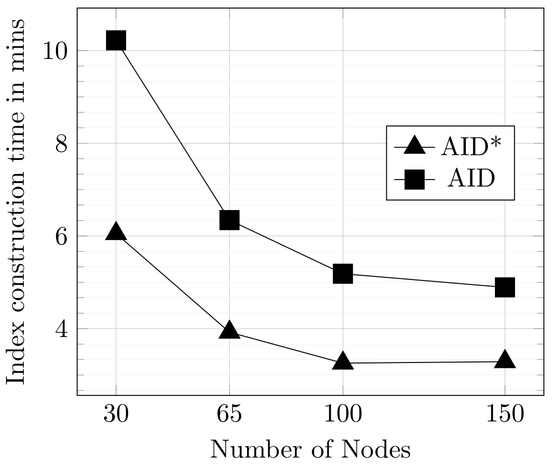
Parameter	Values (default)
Input size	1.6 GB – 1 TB
Number of levels (Z)	1–(20)
Threshold (θ)	10 KB $\cdots$ (10 MB) $\cdots$ 1 GB
Tile dimensions (T)	(256), 512, 1024 pixels
Cluster size	12, 30, 65, (100), 150 (nodes)

We use the following notation for the algorithms that we use in this part. AID (without a star) refers to AID index [12] when constructed on a non-indexed data while AID* indicates the proposed index constructed on an R*-tree indexed file. The suffix

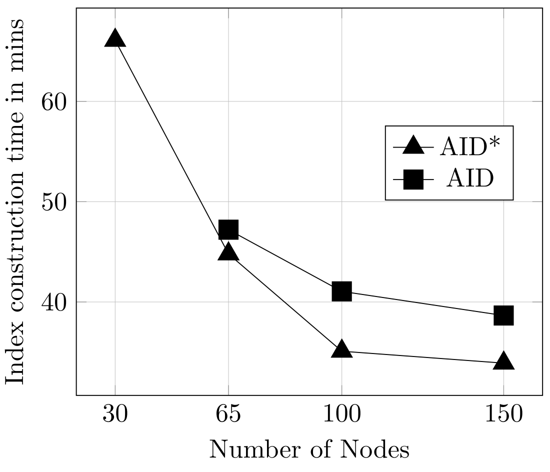
'/Hadoop' and '/Spark' indicates which implementation is tested and on which platform. Finally, HadoopViz indicates the HadoopViz [11] baseline which only runs on Hadoop. We report both running times on Hadoop and Spark to differentiate the improvement gained by the proposed index as opposed to using Spark which is generally more efficient than Hadoop.

3.6.2 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 vary the cluster size from 30 to 150 nodes and show the index construction time on both Hadoop and Spark. Figure 3.6 shows the index construction time in minutes as the cluster size increases. This experiment propagates three 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.



(a) Spark



(b) Hadoop

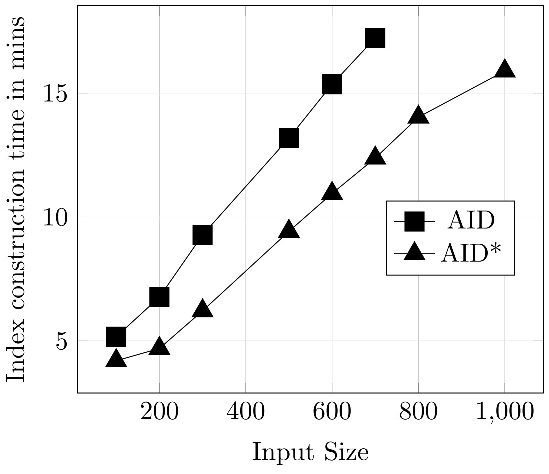
Figure 3.6: Scalability of AID and AID* on Hadoop and Spark

While both Spark and Hadoop implementations for AID* scale well, Spark is up-to an order of magnitude faster than Hadoop. Furthermore, 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.

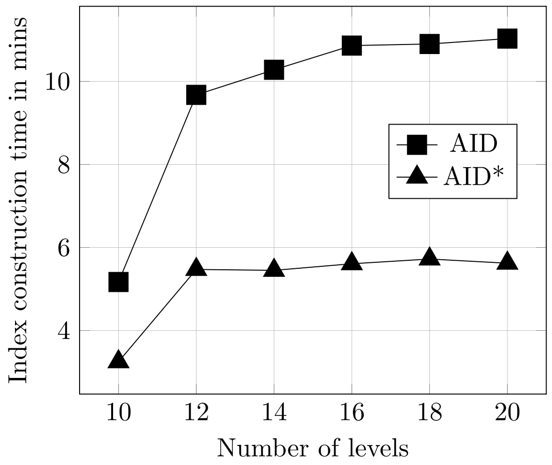
3.6.3 Index Construction Time

This section studies the index construction time for both AID and AID*. In Figure 3.7(a), we vary the input size from 100 GB to 1 TB and report the overall index construction time. This experiment only considers the proposed AID* and AID but not HadoopViz as the latter failed for more than 10 zoom levels. 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.



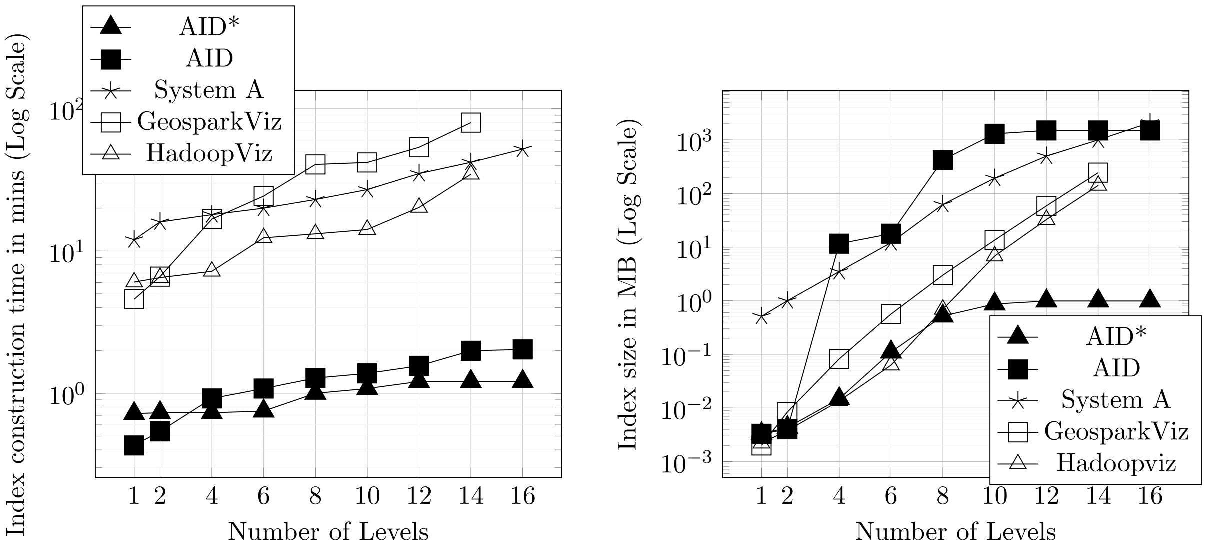
(a) Effect of input size



(b) Effect of number of levels

Figure 3.7: Index construction time

In Figure 3.7(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 while HadoopViz is not reported here as it did not scale beyond 10 levels in this experiment. 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 **??**, $z_D = \lceil \log(100\ GB/1\ MB)/2 \rceil = 9$ 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

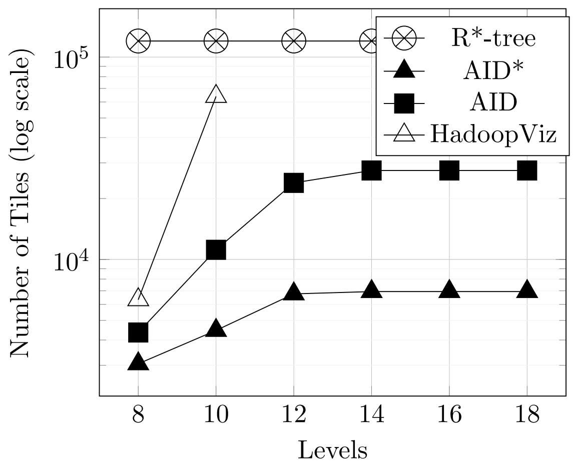
Figure 3.8: Various system comparisons in terms of index construction time and index sizes.

We conduct a similar experiment on other systems with Twitter data and a cluster of 12 nodes as shown in figure 3.8(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 that option, System A failed to generate indexes beyond level 6. We have varied the index construction level from 1 to 16. It should also be noted according to the latest implementation of GeosparkViz [?], the 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.

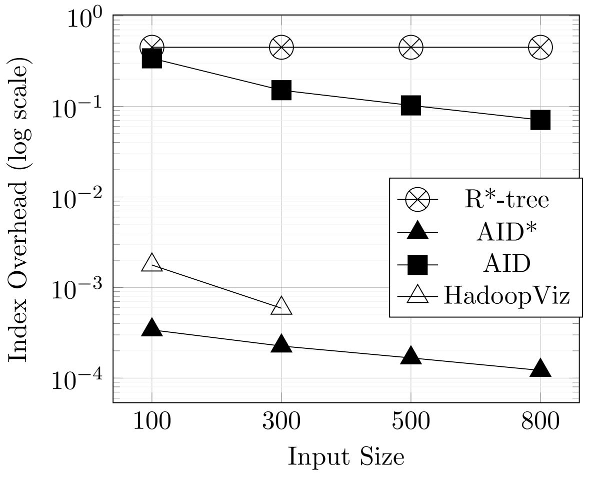
3.6.4 Index size

In this section we measure the index size of AID and AID* as compared to two baselines, HadoopViz image index and R*-tree data index. 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 3.9(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

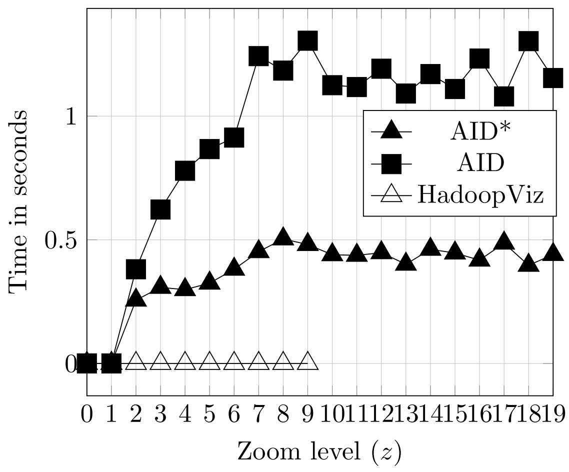
Figure 3.9: Index size

In Figure 3.9(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 in 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) As expected 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 3.7 and 3.8 where the number of images tiles is roughly one third the number of data tiles.

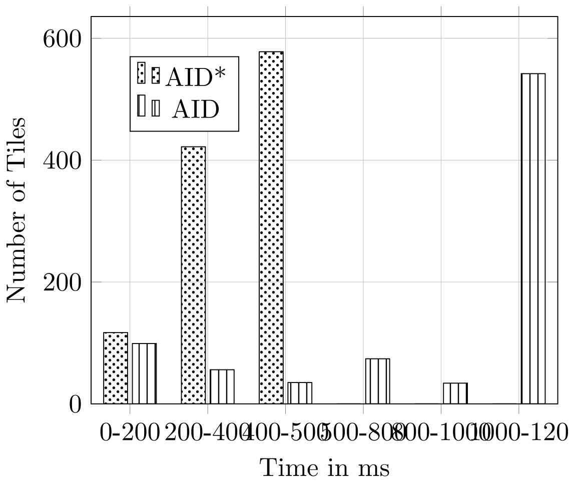
Figure 3.8(b) shows how index size increases for different systems with increasing zoom levels for Twitter data. The index size is measured in Megabytes. The figure provides the following observations: 1) AID* takes least indexing space, which makes it obvious why it can support multiple datasets unlike 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 increasing zoom levels.

3.6.5 Visualization Query

This part evaluates the performance of the visualization query described in Section 3.5 which retrieves the image of a single tile. 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.



(a) Average visualization time per zoom level

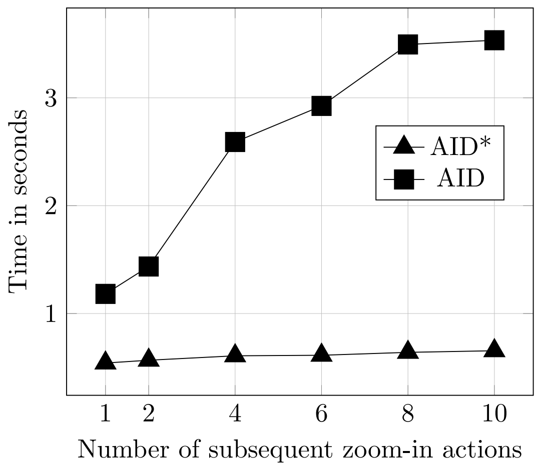


(b) Number of generated tiles grouped by visualization time

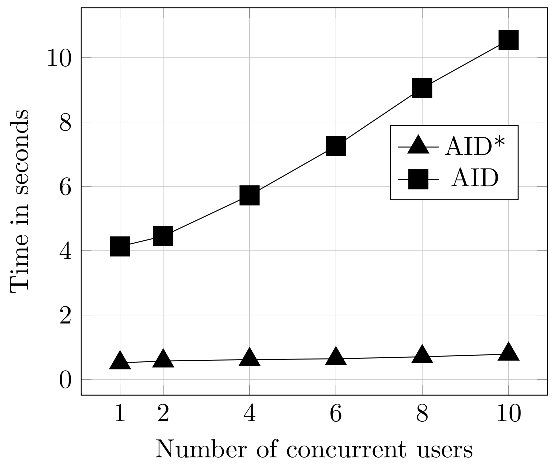
Figure 3.10: Visualization query processing time

In this experiment, we measure the performance of the visualization query which runs on a single machine and operates on the indexes generated by either HadoopViz, AID, or AID*. In HadoopViz, the query processing time is the time to retrieve a pregenerated image from disk. In AID, it is the time to either retrieve an image or select and visualize the records in the data tile. In AID*, it is the time to either return a pregenerated image or search the R*-tree index and visualize the query results.

In Figure 3.10(a), we vary the zoom level from $z = 0$ to $z = 19$ and compute the average visualization query time at each level. 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,



(a) Visualization query time for zooming through multiple zoom levels simultaneously



(b) Visualization query time when multiple tiles are requested together and visualized through multiple zoom levels

Figure 3.11: Visualization query time with parallel requests

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 [27] 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 as the figure suggests. 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).

Figure 3.10(b) projects the number of tiles that are computed within a specific time to give an idea of the number of tiles that need more than 500 milliseconds for a real time query processing. As we can see, for AID* the entire set of tiles up-to level 20 can be queried within 500 milliseconds. Since AID* has no data tiles, 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, not only for a shallow tile requests, the server needs to find its parent data tile by climbing up the pyramid, but also each of this 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 3.9(a) shows us the results of users zooming in through 10 levels together, to zooming in to just one level. In all these cases we deliberately 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.

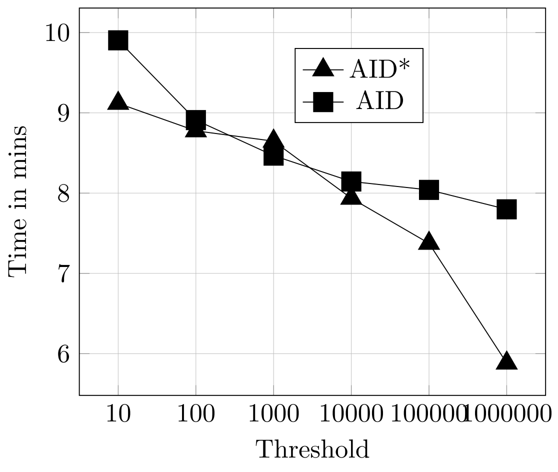
Another scenario could be when multiple users want to visualize a dataset simultaneously. In order to create that scenario, we chose 10 random tiles that represents 10 users requesting each of these tiles at the same time. To address more complex operation, we expect each of these users to zoom-in upto the deepest level. Figure 3.11(b) shows the number of tiles requests 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.

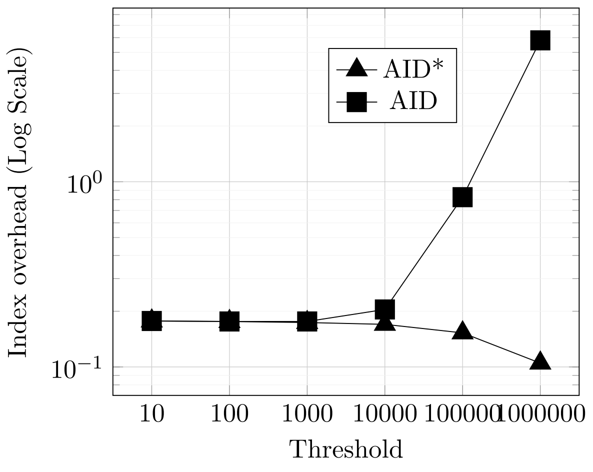
3.6.6 Threshold

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 4.1 shows the results of this experiment on the ALL NODES dataset. Since HadoopViz does not use a parameter θ, its results should be constant and hence we don't show it in our figure.

From Figure 4.1 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 4.1 b).As we increase the threshold, we are making image tile generation more restrictive. This results in less image tile generation, which also results



(a) Time Vs Threshold

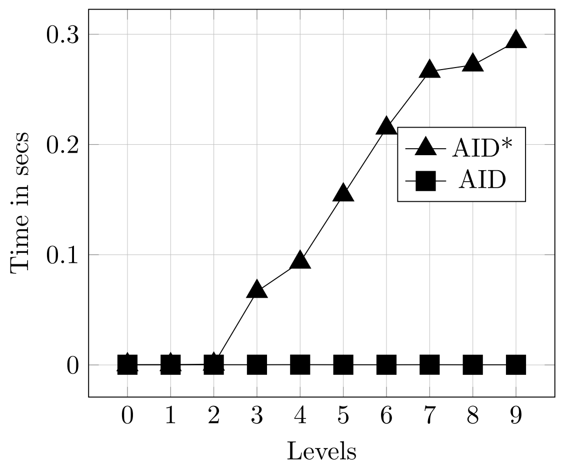


(b) Index Overhead Vs Input Size

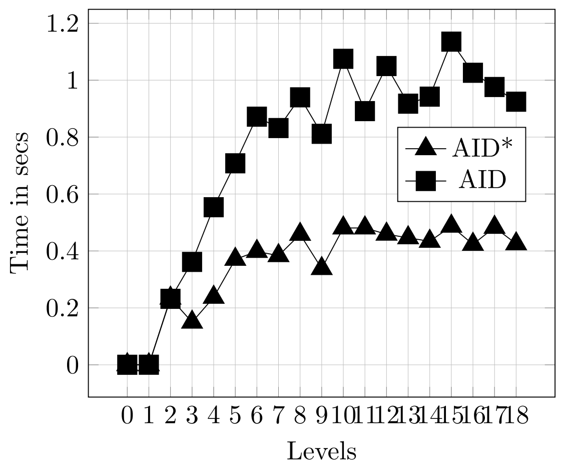
Figure 3.12: Effect of changing threshold on index construction time and number of tiles generated

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. For lower threshold, with very small number of data tiles, the index size is almost same for both AID* and AID. However, as the threshold increase the difference between the index size becomes more prominent between AID and AID*. Finding the optimal value of θ depends on how the users weigh the indexing time and query processing time.

In figure 3.13 we provide the results of querying AID, AID* for threshold 10 KB and 1GB. In figure 3.13 a), the visualization query is run on a threshold of 10KB. Such 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, AID* has



(a) Vis query performance with θ 10



(b) Vis query performance with θ 1000000

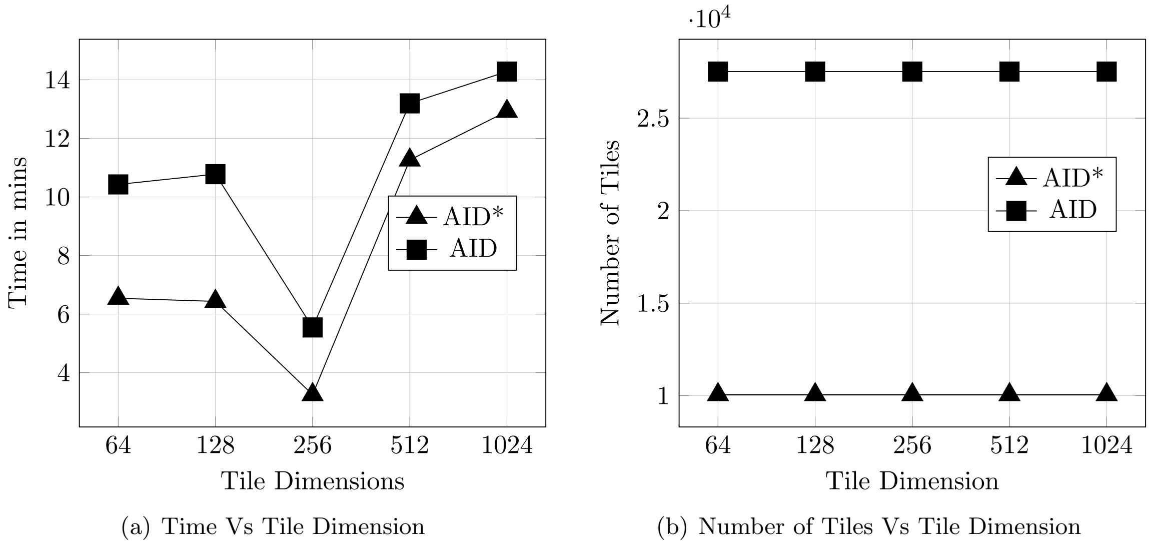
Figure 3.13: Effect of changing threshold on visualization query

no information about empty tiles. For each empty, data or shallow tile requested by the user, AID* needs to refer to its R* index to return the appropriate tiles. This phenomenon is clear in figure 3.13 a) where the reference to empty tiles caused the visualization query time to grow. 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 3.13 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.

3.6.7 Tile Dimension

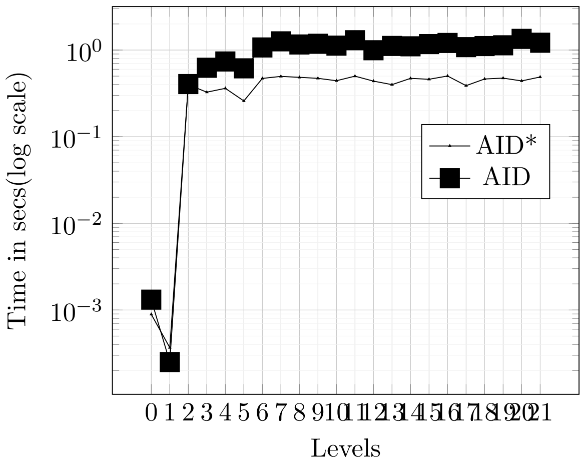
This section focuses on the effect of tile dimensions of the image tiles on the performance of the AID index. It emphasizes on finding an optimal tile dimension that 1) does not take too long in index construction 2) is not too big to be optimally queried by the visualization server

In this experiment, we used five tile dimensions 64×64, 128×128, 256×256, 512×512, and 1024×1024. To get comparable results, we generate a multilevel image with 22, 21, 20, 19, and 18 levels for the tile dimensions of64, 128, 256, 512, and 1024, respectively. This ensures that the level of details in the deepest level is exactly the same in the three images because one tile of dimensions 512×512 has the same number of pixels as four tiles of dimensions 256×256.

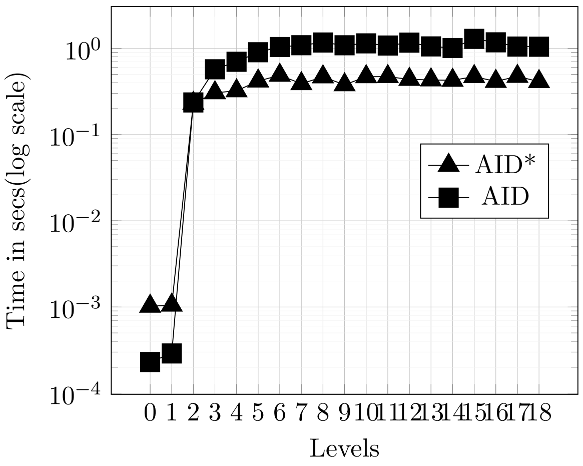


(a) Time Vs Tile Dimension (b) Number of Tiles Vs Tile Dimension

Figure 3.14: 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 3.15: Visualization query performance with changing tile dimension

In Figure 3.14 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 3.14 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 3.14 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 /reffig:TileDimensionVizQuery.

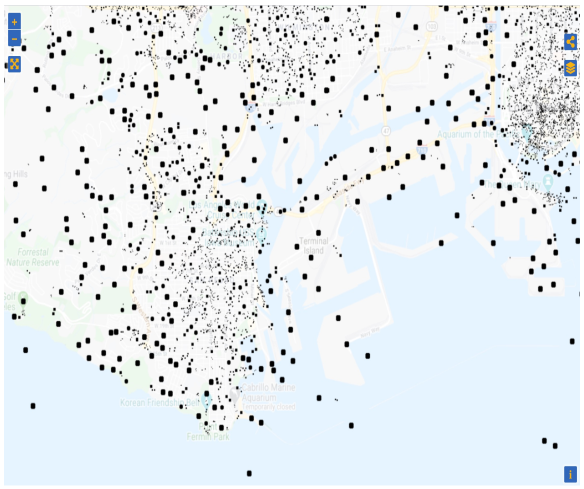
3.6.8 Qualitative Comparison

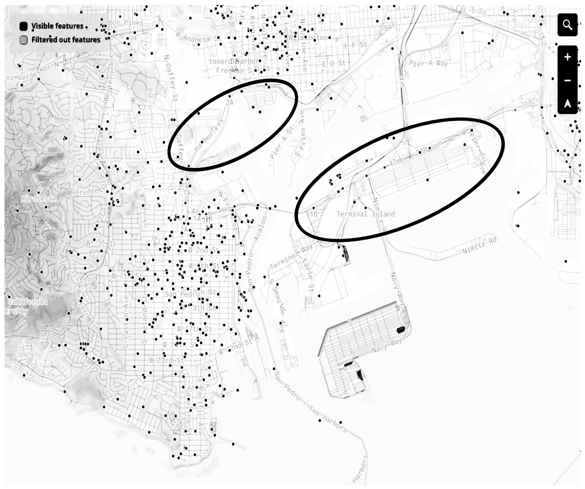
This section compares the visual exploration provided by AID* with two different commercial systems referred in the paper as System A and System B. This part shows the negative effect of downsizing the data through sampling on the end-user experience as opposed to visualizing *all* the records in AID and AID*.

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 3.8(a) and 3.8(b).

Figure 3.16 provides a comparison between the two visualizations. First, AID* took 11 minutes to build the index while System A took 42 mins because it is a single-machine system. Second, 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


(a) AID* - Index construction in 11 minutes. Index size is 989.4 K. All records are visualized.



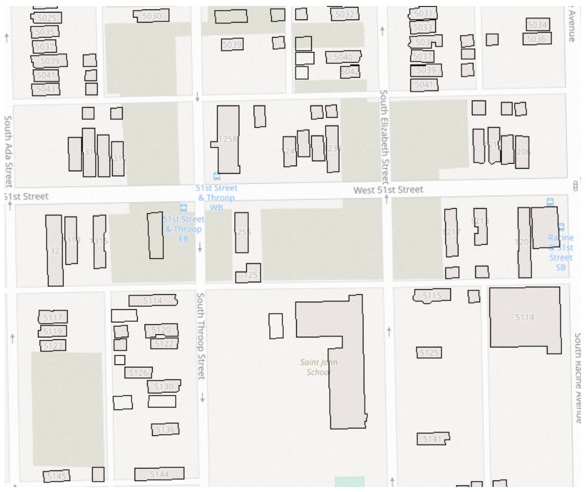
(b) System A - Index construction in 42 mins. Index size is 1 GB. Only a sample of the records is visualized

Figure 3.16: Comparison of exploring the `Tweets` dataset on AID* and System A

them. Figure 3.17(b) highlights some of the areas that seem empty on System A while they do have data as shown on AID*. Another example is provided where `Buildings` is compared System A and AID* in figure 3.18. This dataset has a different shape (polygon). Similar to `Tweets`, System A took way longer than AID*, and yet it needed to sample the dataset. As we can see, that even at a zoom level of 12, System A has many empty spots as compared to the visualization produced by AID*.

System B

This system does not provide a detailed description of their index construction or tile generation method. However, they provide a web-hosed 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.


(a) AID* - Index construction in 40 minutes. Index size is 120 Mb. All records are visualized.

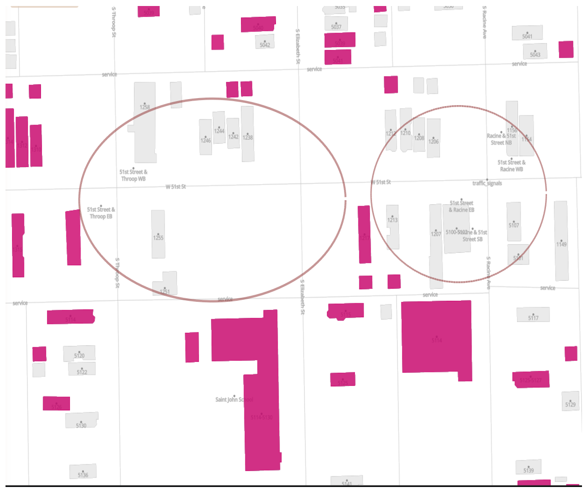

(b) System A - Index construction in 2.4. Index size is 22 GB. Only a sample of the records is visualized

Figure 3.17: Comparison of exploring the `Buildings` dataset on AID* and System A

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 3.18 suggests, AID* provides a more detailed visual points as compared to System B. We also noticed a significant delay 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.

3.6.9 Cost Model Verification

This section experimentally verifies the the accuracy of the proposed cost model. Figure 3.19 measures the index size in terms of number of files as the number of zoom levels Z changes from 1 to 20. We use equations 3.8, 3.7, and 3.9 to compute the number of image, data, and shallow tiles. Then, we compute the estimated size of the three indexes based on which tiles

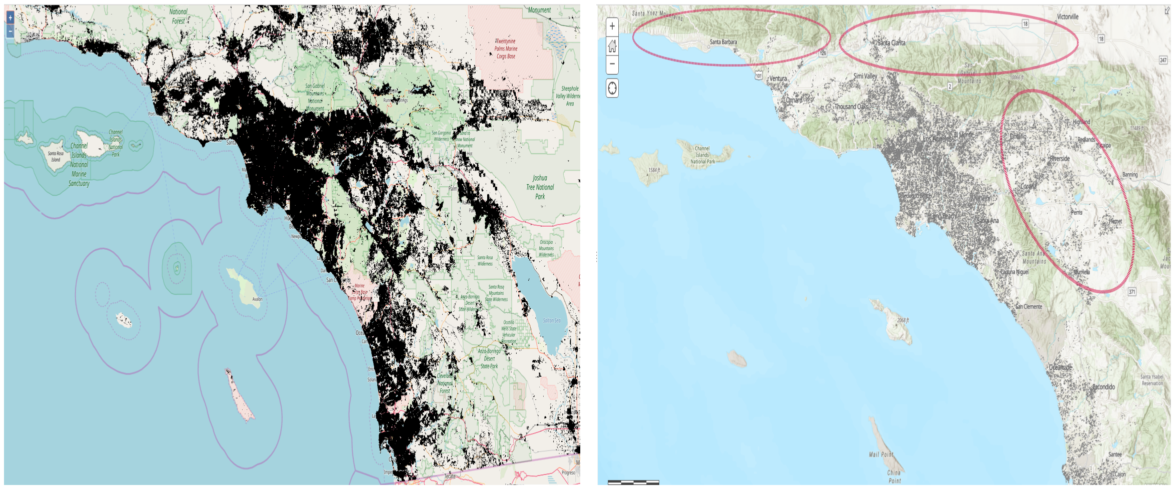
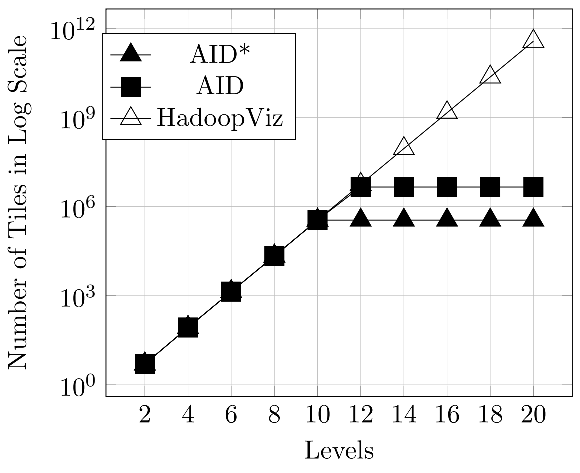

(a) AID* visualizes all records (b) System B visualizes sample records

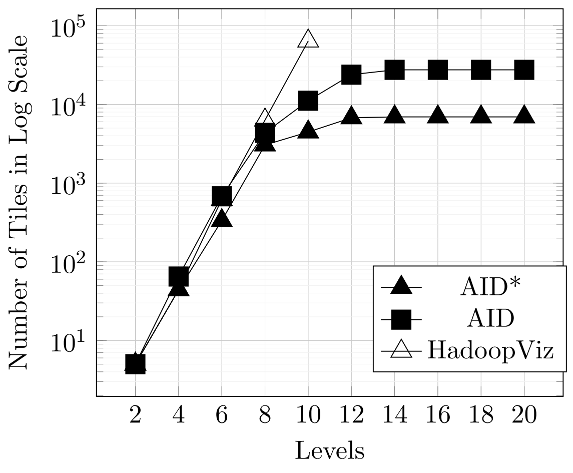
Figure 3.18: Comparison of exploring the Microsoft `Buildings` dataset on AID* and System B

are materialized in each index. We further generated a synthetic uniform data as described in [54] to test the model. Since the results were exactly the same as figure 3.19(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.

We still noticed two discrepancies between the estimated and actual sizes. First, the AID and AID* indexes stabilize at around 10^6 based on the cost model in Figure 3.19(a) while they stabilize around 10^4 in the real experiment in Figure 3.19(b). This is a result of ignoring the empty tiles in our analysis which over estimates the index sizes. In the future, we can extend our cost model using the box counting technique [3] to account for empty



(a) Theoretical index size

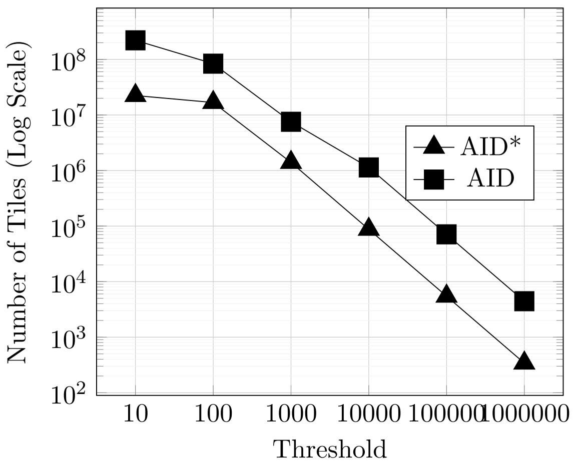


(b) Actual index size

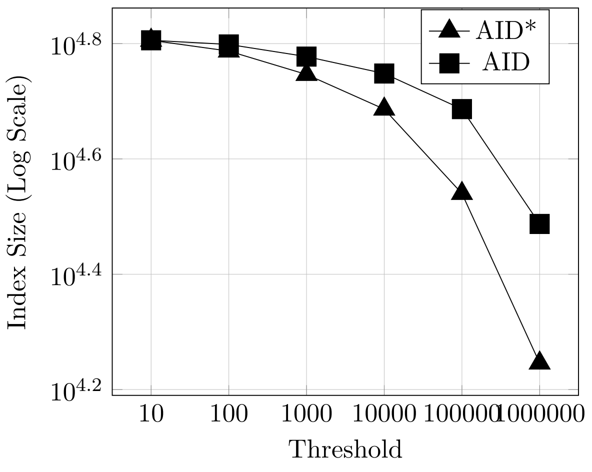
Figure 3.19: Effect of changing zoom levels on the tile numbers in proposed cost model Vs. actual number of tiles

tiles. 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.

Figure 3.20 illustrates the effect of the threshold (θ) on the index size based on the cost model and reality. HadoopViz is not included as it does not use the threshold. Both plots in Figure 3.20 show a steady decline in the number of tiles for both AID and AID* with increasing threshold as it is expected. Again due to our assumption of uniformity of data the number of tiles are way more in the cost model as compared to reality. Moreover 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



(a) Theoretical Number of Tiles Vs Threshold



(b) Actual Number of Tiles Vs Threshold

Figure 3.20: Effect of changing threshold on tile numbers in the proposed cost model Vs the actual number of tiles

threshold of 10. Hence the difference in index size of AID and AID* in Figure 3.20(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.

3.7 Conclusion

This paper addresses the problem of interactive exploration of big spatial data. The paper 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. AID is based on the idea of defining a pyramid of tiles on the input and it introduces a new model to classify these tiles based on their processing time. This classification is then plugged into a Spark-based program that constructs either AID or AID* efficiently for very large datasets. Additionally, we derives a theoretical cost model for the index constructions process to understand the behavior of the proposed indexes and the baseline solution. 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. We showed that the proposed solution is the only one that can provide an arbitrary number of zoom levels for extremely large datasets without compromising the end-user experience. 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.

Chapter 4

An Application of UCR Star's Visualization Engine

4.1 Introduction

UCR-Star, that is capable of hosting hundreds of thousands of geospatial datasets that a user can explore visually to judge their quality before even downloading them. The AID and AID* played an important role in building this system. This following section provides a deeper dive into the core engine behind UCR-Star. It provides a web interface geared towards database researchers to understand how the index internally works. It provides a comparison interface where the attendees can see side-by-side how two versions of the system work with the ability to customize each of them separately. Finally, the interface reports the response time of the indexes for a quantitative comparison.

At UCR, we have built the Spatio-Temporal Active Repository, **UCR-Star** [`https://star.cs.ucr.edu`] which assists users in the dataset selection problem. Figure 1.2 provides a high-level overview of UCR-Star. The data portrayed on the map is the bird's eye view of Chicago crime dataset. UCR-Star currently hosts **152 datasets** with nearly a **terabyte** in size and it is capable of hosting hundreds of thousands of such datasets. This work demonstrates the **visualization engine behind UCR-Star**. Internally, UCR-Star indexed each dataset using the adaptive image-data index (AID) [12] which arranges pregenerated image and data tiles in a pyramid structure that enables the web server of UCR-Star to provide the desired real-time visualization.

It should be noted that the main focus of this chapter is *not* on the graphical interface of the repository. The novelty lies in building **one single open sourced system that can host hundreds and thousands of big geospatial data on a single machine for interactive exploration**. This serves the database communities, geospatial communities, and data science communities, who are in a constant search for datasets, to process and analyze these spatial datasets without downloading them.

The contribution of this chapter can be summarized as follows:

1. It introduces the open sourced repository, UCR-Star, that allows users to visually explore various publicly available big geospatial datasets.

2. Demonstrates the internal design of the visualization engine behind UCR-Star which provides extremely interactive visualization of terabytes of data on a single machine.

3. It provides a comparison interface between AID and AID* indexes, which are the key components of this visualization engine. This interface allows the audience to change the

system parameters and observe the performance.

4.2 Data Processing using AID and AID*

In order to prepare the dataset for visualization, UCR-Star uses the partial pyramid structure of **AID** or **AID***. As we have learnt from the previous chapter, these indexes store way fewer tiles as compared to a regular tile-based visualization. These indexes leverages the sparsity of geospatial data in order to pregenerate and materialize selected tiles beforehand. The remaining tiles are generated on-demand upon user request. This results in a very small-sized disk-resident index which makes it possible to host hundreds of thousands of such indexes in a single-machine system for the final visualization.

AID and AID* control the number of tiles to generate using a **threshold** (θ) which represents the largest size of data that can be generated on-demand without sacrificing the interactivity of the system. This threshold is used to classify tiles into four classes:

***Image*:** If a tile size is larger than θ, it is classified as an image tile.

***Data*:** If a tile size less than or equal to θ and has an image tile as a parent, then it is classified as a data tile.

***Shallow*:** If a tile size is less than θ, but the parent is a data tile, then it is classified as a shallow tile.

***Empty*:** When a tile size is zero, it is classified as an empty tile.

The details of how the tiles are classified and generated at scale can be found in chapter 3.

AID: The AID index significantly reduces the index size by generating and storing only image and data tiles. Image tiles are stored as pregenerated image files. Data tiles are stored in a geospatial data format, e.g., CSV or Geojson. When a user requests a data or a shallow tile, it can be generated on-the-fly by reading and processing exactly one data tile which sets an upper limit on the time needed to generate any tile.

AID*: The AID* index further improves over the AID index by storing only image tiles. Instead of storing data tiles, the AID* index uses a separate data index, e.g., R-tree or R*-tree, that can be utilized with a range query to generate data and shallow tiles on the fly.

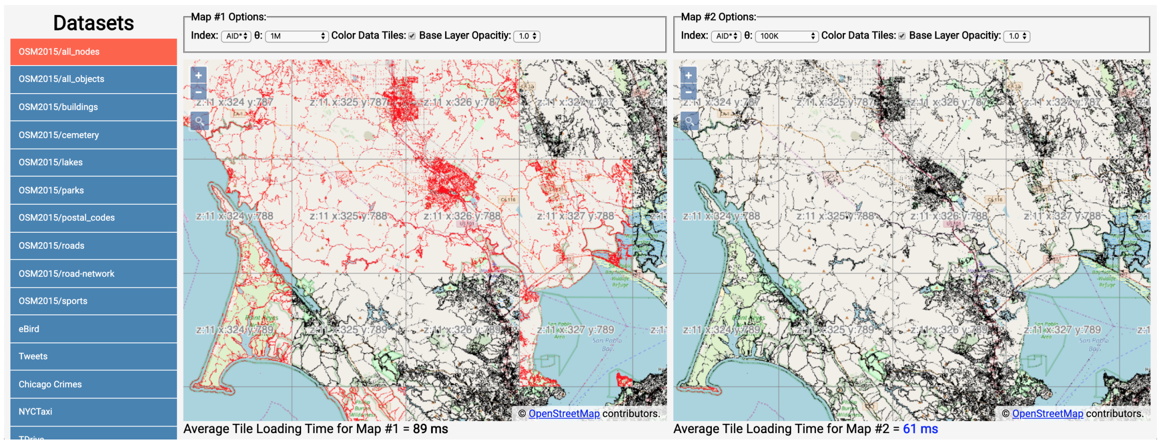
The example in Figure 3.2 illustrates the saving in the storage of the index which also reflects in the time needed to construct the index. For this example, if all tiles are stored, 954 tiles will need to be generated and stored. If an AID index is built with $\theta = 1500$, only 229 tiles will be stored. In AID*, only 75 image tiles will be generated and stored. If AID is built with $\theta = 3000$, then 135 tiles are stored. For AID* that reduces to 43. This example is for only six levels. In the reality, we generate 20 levels and the reduction in index size and construction time for AID and AID* is several orders of magnitude

4.3 Visual Exploration

Visual exploration consists of a single visualization query that retrieves an image for a single tile in the pyramid structure. This visualization query is integrated into OpenLayers to provide the interactive visual interface for end users. This visualization query consists of a dataset ID and a tile ID (z, x, y), where z is the zoom level, and (x, y) is the position of the tile in the grid at level z. The result of the query is an image that represents the requested tile. If the requested tile is pregenerated (Image tile), then it can be fetched and returned immediately. Otherwise, it needs to be generated on-the-fly using the AID or AID* index. For AID, the tiles generated on-the-fly are either data tiles or shallow tiles. For the data tiles, the images are simply generated from the information within the data tiles. However for the shallow tiles, the parent of the tile needs to be traced up the pyramid to generate the image for the tile. In case of AID*, a simple range query on the pre-indexed input data is used to generate the tiles on-the-fly. It should be noted that the tiles generated on-the-fly should be small enough to not affect the interactivity of UCR-Star. The threshold (θ) is hence an important component in the design and making it too big or too small can affect the real-time interaction of UCR-Star.

4.4 Web Interface

Figure 4.1 represents **the web interface of this demonstration.** On the left side of the page all the datasets are listed. Users can choose the dataset they want explore from the list. Users are able to zoom-in deeper into the dataset for detailed view or zoom-out for bird's eyes view, similar to the interaction of a standard webmap. UCR-Star provides 20 zoom levels for all generated visualization. It should also be noted that UCR-Star does not aggregate or sample the data which means that all individual records for any dataset are visible at deeper zoom levels. The figure also shows that the datasets are realized on top of OpenStreetMap (OSM), but there are options of switching the base layer from OSM to Google Satellite or Google Maps. The page is divided into two panels, so that users can compare and



Map #1 Options:
Index: AID* θ: 1M Color Data Tiles: ✓ Base Layer Opacitiy: 1.0

Map #2 Options:
Index: AID* θ: 100K Color Data Tiles: ✓ Base Layer Opacitiy: 1.0

Average Tile Loading Time for Map #1 = 89 ms

Average Tile Loading Time for Map #2 = 61 ms

Figure 4.1: UCR-Star visualization engine showcasing the web interface and the effects of different threshold

contrast various features side-by-side. On each of the two sides, they have the option of choosing between the two indexing techniques (AID and AID*), different thresholds (θ), base layer opacity, and an option of distinguishing between pregenerated image tiles and tiles generated on-the-fly by color coding them. Both maps are geosynchronized which means that an interaction with one map will always reflect on the other so that they show the same region. Users can also search for a specific location on the map by entering any textual query, e.g., country or city name. At the bottom left corner of both maps, the average tile loading time is provided for the users to identify the efficiency of the features chosen by them. The average time is updated as users interact with the map. The back-end is hosted on two identical AWS servers, one for each map. This ensures a fair comparison between the two techniques since there will be no contention between them on the same set of resources.

4.4.1 Distribution of On-demand Tiles

The two proposed indexes, AID and AID*, pregenerate and materialize only a few image tiles while the remaining tiles are generated on-demand. The first demo scenario **helps attendees to understand the ratio of pregenerated image tiles and tiles on-the-fly in UCR-Star at different levels and how non-static tiles increase with increasing zoom levels.**

When the users select the 'Color Data Tiles' checkbox, tiles that are generated on the fly will be colored in red, as seen in Figure 4.2. As they zoom in deeper, the users will see a surge in the red tiles denoting an increase of tiles generated on-the-fly. This is because with deeper zoom levels data coverage by each tile decreases causing the size of the tiles to fall below the threshold (θ). From the figure itself we can see that the black tiles are more dense, containing more records which explains the reason for them being bigger in size. Whereas the red ones are comparatively sparse and contains lesser records. The tile classifications are controlled by threshold (θ), defined in the index construction phase and the details of this can be obtained from [12].

4.4.2 Threshold (θ)

The key parameter for configuring both AID and AID* is the threshold θ. In this part of the demo, the users can see the effect of changing the threshold θ on both AID and AID*. The threshold θ controls the trade-off between the index size and the response time of the visualization query. As θ increases, fewer tiles will be generated by both AID and AID* while the number of on-demand tiles will increase and the server will take more time

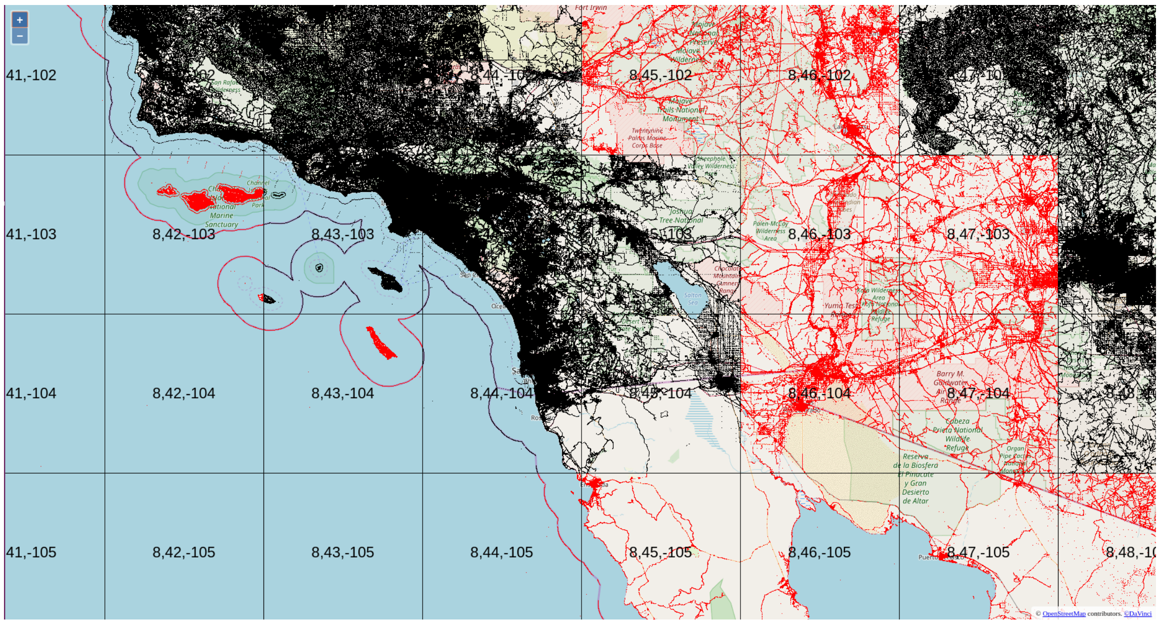


Figure 4.2: UCR-Star showcasing static image tiles (black) and tiles generated on-the fly (red)

to generate them. The attendees **can see how increasing threshold θ hinders the interactivity of the system and takes more time in generating the tiles on-the-fly.** Similarly extremely small threshold results in the indexes having almost all static tiles, increasing the index overhead, which makes it impossible to host multiple datasets within UCR-Star.

In Figure 4.1, we see two different thresholds (θ=1 Mb and θ=100 KB) have been used for visualizing the same dataset. To represent how much tiles are generated on-the-fly, we have kept the color distinction on. As we can see higher threshold (left panel), results in more tiles to be generated on-the-fly, resulting in longer time for average tile loading.

4.4.3 AID and AID*

As we had mentioned earlier, we developed two different indexing techniques, AID and AID*, to use in UCR-Star. Through this demonstration, users **will be able to see a side by side comparison of AID and AID* in UCR-Star's visualisation engine.** For the same threshold both AID and AID* have the same number of static image tiles (tiles that are bigger than the threshold (θ) and are pregenerated as image tiles). However, AID* works on a previously indexed dataset when generating images on the fly, while AID generates it from data files (e.g., CSV). This affects the time taken to generate the image tiles on-the-fly. For a data tile, only one file needs to be read from disk and processed but the entire file needs to be read since it is not internally indexed. For example in AID, if a shallow tile is processed, only a small portion of the data tile is needed but since it is stored as a non-indexed file, it need to be processed in its entirety. On the other hand in AID*, the associated R*-tree index needs to be used which means that the index structure needs to be traversed before the data is located. However, since the R*-tree node is much smaller than a data file, the amount of data that needs to be processed can be further limited than AID.

This phenomenon can be experienced by the attendees, by selecting AID on one panel and AID* on the other panel of UCR-Star. As the attendees will zoom in or out or pan across their desired dataset they can see the average tile loading time for both AID and AID* to verify the efficiency of both methods.

Chapter 5

Recommendation on a Public Repository for Geospatial Exploration

5.1 Introduction

For geospatial data, all records are associated with a geolocation which makes it natural to visualize these data on a map. UCR-Star [13] [`https://ucrstar.com`] is an interactive geosaptial data repository which allows users to visualize big geospatial datasets on a map. Through an intuitive map interface, users can explore both the metadata and the *contents* of terabytes of geospatial data. UCR-Star is a tool that allows users to assess the *coverage*, *accuracy*, and *quality* of these datasets and download either the entire dataset or a subset of it. This gives users a lot of flexibility.

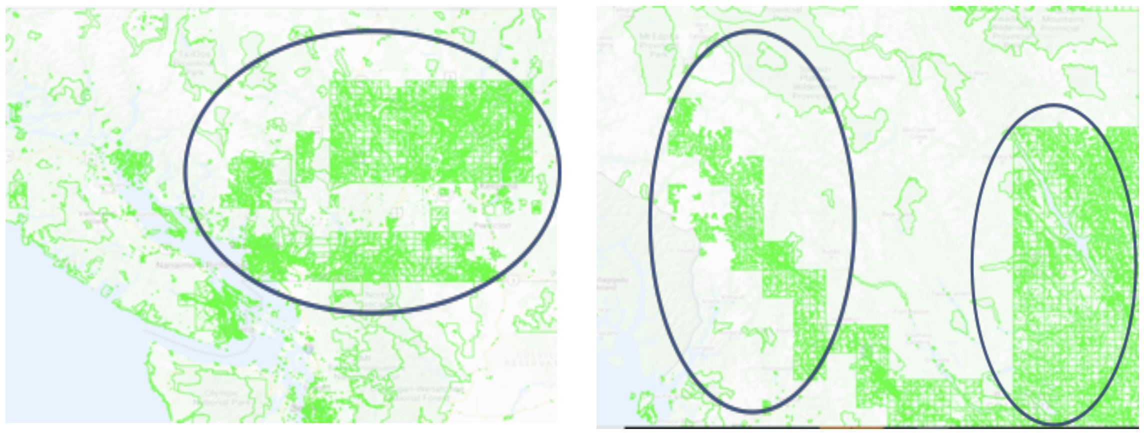


Figure A:User's usual exploration pattern
Figure B:Recommendation based on user's exploration pattern

Figure 5.1: Figure A represents user's exploration pattern on a dataset, Figure B shows the recommendation based on Figure A where a user is given direction to find similar location for the dataset

While allowing users to browse through the metadata and contents of the datasets, it also opens new problems of finding not just a dataset, but a part of it that can be useful. For example, figure 5.1 shows another example of one dataset that has some regions that could be of interest to some users. So, a user who showed interest in the highlighted region on Figure 5.1A might also be interested in the similar areas highlighted in Figure 5.1B.

The solution that this chapter proposes to the aforesaid problems is to build a recommendation system for spatial exploration. The system's main motto is to guide a user through the visual exploration of the spatial datasets. As mentioned in 1, we divide this problem into three parts.

i) *recommending different datasets*

ii) *recommending area of interest (AOI)*

iii) *recommending different visualizations for the same dataset*

Challenges: The recommendation problem has been well studied and deployed in popular systems such as MovieLens [**?**], Netflix, Amazon, and Spotify. However, we highlight two main challenges that make existing techniques inapplicable to our problem. First, all these systems maintain personal user records that allow them to track user activity over a long time. For example, when a user opens Amazon, the recommendations can pop up right away based on past user interest tracked through login information or cookies silently collected in the background. On the other hand, UCR-Star is publicly available and does not keep any user information or cookies. This design becomes also more appealing due to the recent data breaches from several systems and the enforcement of GDPR restrictions from the European Union which make it more challenging to collect and maintain user data. To sum it up, without user information each new visit to UCR-Star is treated as a new user.

The second challenge that we have to face is the lack of *item* information. In existing systems there is a well-defined set of items, e.g., movies or products. However, in the problem that we address in this paper, there is virtually unlimited number of items we can recommend. For example, any subset of the data can be an item to recommend. Some recent systems [18] showed that visualization recommendation can help with data exploration but it focused on non-spatial visualizations, i.e., charts, and relied on analyzing the data itself rather than mining user behavior.

To address the above challenges, we propose a visualization-centric recommendation system that takes as input the logs collected by UCR-Star and builds a model that can provide the three types of recommendations. To match with how existing visualization systems work [12], we divide the space using a quad-tree-like pyramid structure and assign a unique ID for each tile. Furthermore, we augment this ID with the dataset and visualization type. This becomes our pool of items that we recommend from. We use UCR-Star logs to construct a model that catches the relationship between these items and use this model to recommend an item which can represent a dataset, a visualization, or a different location in the data. This allows us to solve the three recommendation categories using one recommendation model.

To build the recommendation model, we explore three solutions. First, we try the standard item-based collaborative filtering (CF) which relies on a similarity matrix between tiles based on user behavior. Second, we adapt RESCAL, a tensor decomposition technique for knowledge bases, to build a relationship between items. Third, we use a graph convolutional network (GCN) method, which is popular in recommending friends in social networks, to find relationships between these tiles through a network constructed from user exploration history. We apply the three methods on a real dataset collected from UCR-Star over a period of six months and show that the traditional CF technique does not operate well for this problem while RESCAL and GCN methods are comparatively better. A background for these models are provides in section 2.0.11.

The rest of the chapter is organized as follows. Section **??** talks about the existing recommendation systems and how they align with our problem. Section 5.2 formulates the problem. Section 5.3 describes the proposed solution. Section 5.4 gives the results of our experiments.

5.2 Problem Definition

This section describes the problem that this paper addresses and gives a formal definition to it. In visual exploratory systems, like UCR-Star, users explore the datasets through three main interactions. (1) Users can move the map via zoom in/out, pan, and fly-to location. (2) Users can switch to a different dataset. (3) Users can switch between different visualizations for the same dataset. The goal of this paper is to analyze users behavior. For example, for map interaction part, users who explore taxi pickup datasets in New York might tend to zoom in to Manhattan or Central Park. Second, for the switching between datasets, users who are interested in a road network dataset might tend to switch to a public transit datasets in specific cities. Third, for the switching between visualizations, users might prefer to use heatmap visualization for point datasets at a high level, but switch to scatter plot when they zoom in deep enough. The main challenge here is how to abstract the users interaction in a way that allows us to provide these recommendations.

To highlight how this recommendation problem is new, we make an analogy to the popular movie recommendation systems, like MovieLens [?] or Netflix. Those systems recommend movies based on a user's taste and its similarity to other users [?]. In contrast, the proposed system would recommend *movie clips* in movies based on which parts of the same movie or other movies the users watched. This might not be what the movie industry wants, but it would be very helpful for data exploration. In addition, unlike services like YouTube where users explicitly extract and upload movie clips, the proposed system needs to automatically infer the interesting parts of the data without users explicitly extracting and uploading smaller subsets. In other words, YouTube still addresses the traditional user-item recommendation where the items are videos while our system needs to recommend parts of items not the whole items.

5.2.1 Input

The input to our problem is extracted from the logs generated by UCR-Star. The log consists of a sequence of tuples in the form $\langle t, d, v, z, x, y\rangle$ where t is the requested timestamp, d is the dataset ID, v is the visualization ID, $z \in [0, 19]$ is the zoom level of the tile, $x \in [0, 2^z)$, $y \in [0, 2^z)$ are the column and row of the tile in zoom level z. Interested readers can refer to [12] and Appendix **??** for further information. These logs are incredibly useful in inferring user behavior since they record each tile request from users. However, they also impose the following challenges that we need to deal with.

1. The logs are naturally anonymous since UCR-Star is public and does not require login.
2. The logs do not explicitly store user interactions, e.g., zoom in or out, but it only stores the tile requests made by the browser.
3. The front-end browser uses caching in JavaScript to reduce server requests, hence, not all user interactions result in a server request. For example, if the user zooms in and then zooms out, only the zoom in will result in server requests.

Out of these logs, the input consists of two parts, *historical logs* and *active logs* as defined below.

Definition 14 (Historical log) *A historical (offline) log H consists of user requests for users who are no longer online. These logs are used to build the recommendation model.*

Definition 15 (Active (Online) log) *An online log O consists of user requests for one user who is actively exploring the system. An online log is used to provide new recommendations for this user as long as he/she is online. We also use the term* visualization trajectory *to denote the online log since it resembles navigating through a dataset.*

5.2.2 Problem Formulation

Given the two inputs described above, i.e., offline and online logs, we define three recommendation tasks for three common use cases, i.e., recommending an area of interest (AOI), recommending a dataset, or recommending a visualization. After that, we generalize them into one problem that this paper addresses in the next section.

Task 1: AOI Recommendation. The first task is when a user is exploring a dataset and the system recommends a new location based on the parts that the user visualized in the visualization trajectory. In other words, given a trajectory that consists of a sequence of tile IDs $O_T = (z_i, x_i, y_i)$, where $1 \leq i \leq N$, and N is the length of the trajectory, the system recommends the next set of tile IDs $R_{POI} = \{(z_j, x_j, y_j)\}$, where $1 \leq j \leq k$ and $k = |R_{POI}|$ is the number of recommendations. The recommended tiles should be based on how the tiles in the trajectory are similar to other users in the offline logs. We assume here that the trajectory belongs to one dataset and one visualization and that the system recommends tile IDs within the same dataset and visualization.

Task 2: Dataset Recommendation. In this task, a user is exploring one or more datasets and the system recommends a new dataset that is relevant to the previous datasets that the user explored. So, given a sequence of datasets $O_D = \{d_1, d_2, \ldots, d_N\}$ in a user trajectory, the system recommends the next set of datasets $R_D = \{d_1, \ldots, d_k\}$ that are relevant to the datasets in the trajectory. For example, if the user explored the datasets `taxi-pickups`, `road-network`, and `bus stops`, the system might recommend `railroads` and `airports`. However,it should be noted that, since we depend on user behavior for the recommendation, if a user switches from one dataset to another randomly, that behavior will be logged and affect the recommendation accordingly.

Task 3: Visualization Recommendation. In this third task, a user is exploring one dataset but switches between visualizations for that dataset and the system recommends the next set of visualizations that the user might find interesting. Given a sequence of visualizations $O_V = \{v_1, v_2, \ldots, v_N\}$ in a user trajectory, the system recommends the next set of visualizations $R_V = \{v_1, \ldots, v_k\}$ for the same dataset. For example, if the user explored a point dataset in the following visualization order *heatmap* $\rightarrow$ *choropleth map for states* $\rightarrow$ *choropleth map for counties*, as the user keeps zooming in, the system can recommend *choropleth map for cities* $\rightarrow$ *scatter plot.*

Visualization Recommendation Problem. Instead of addressing each of the above three tasks as separate problems, we propose one generalized problem that can be used to solve the above three tasks. First, we make the following definitions.

Definition 16 (Request Pattern) *A request pattern p is a request that has some of its values replaced with '_' which means* any value*. For example, the pattern $p_1 = (d, \_, z, x, y)$ means a request for a specific dataset d, a specific tile ID (z, x, y) and any visualization.*

Definition 17 (Pattern Match) *We say that a pattern p matches a request r, denoted $p \approx r$, if $\forall a \in \{d, v, z, x, y\} p.a = \_ \vee p.a = r.a$. In other words, for each of the five attributes,* *either the pattern contains a wildcard (_) or is equal to the request. We also define $p \in R$, where R is a set of requests, as $\exists r \in R : p \approx r$. In other words, a pattern p belongs to a request set R if it matches at least one request in it.*

Definition 18 (Generalized Recommendation Problem) *Given an online log O that consists of a sequence of requests, the problem is to recommend a set of recommended requests R where $O \cap R = \emptyset$ and $r_i \in O \cup R = (d_i, v_i, z_i, x_i, y_i)$. In other words, given a sequence of online requests O, the system recommends a set of new requests R that the user did not visit. In this general form, both the online requests and recommendations can contain any dataset, visualization, or tile ID.*

To solve Task 1 where we recommend a new location for a specific dataset d and visualization v, we define $R_{POI} = \{(z_j, x_j, y_j) : (d, v, z_j, x_j, y_j) \in R\}$. This means that out of the set of general recommendations R, we select the ones that belong to a specific dataset d and visualization v.

To solve Task 2 where we recommend a new dataset, given a sequence O_D of previously visited datasets, we define $R_D = \{d_j : (d_j, \_, \_, \_, \_) \in R \wedge d_j O_D\}$. This means that out of the set of general recommendations R, we select the ones that recommend a new dataset d_j not visited earlier.

Finally, in Task 3 we need to recommend a new visualization for a specific dataset d given a set of previous visualizations O_V for this dataset. Thus, we define R_V as the set of recommended visualization as $R_V = \{v_j : (d, v_j, \_, \_, \_) \in R \wedge v_j O_V\}$. In other words, out of the general set of recommendations, we keep the ones that recommend a new visualization v_j for a specific dataset d.

5.2.3 Output

The output of the proposed recommendation system can take one of two representations, a server-side representation and a client-side representation. The server represents each recommendation as a request in the form (d, v, z, x, y). The client translates this recommendation to a form that the user can understand. For example, if the recommendation belongs to the same dataset and visualization that are currently visible, the client-side would translate that to the appropriate instruction, e.g., zoom in. Alternatively, if the recommendation belongs to a different dataset, the client-side would translate it to a suggestion of switching to another dataset. For the rest of this paper, we will focus on the server-side representation, i.e., recommending a set of requests.

5.3 Proposed Visualization Recommendation Model

This section describes three proposed models for the visualization recommendation problem. To the best of our knowledge, a recommendation system on *geospatial visualization* is an unexplored area. Unlike most popular recommendation systems, the relationship between users and products do not exist in this problem for the following two reasons. First, we do not have users in the system since it is public with no tracking of user information which means that we have the cold-start problem with *every* user. Second, since we would like to recommend areas-of-interest (AOI) within a dataset, we technically have an infinite number of items that we can recommend. These challenges call for a novel design for recommendation systems which we describe in this paper.

In the following part, Section 5.3.1 describes a preprocessing step that transforms the UCR-Star logs by introducing some structure that will help when applying recommendation models. Then, we propose three visualization recommendation models in Section 5.3.2-5.3.4 that are all adapted to work for the proposed problem.

5.3.1 Data Preprocessing

UCR-Star [13] [`https://ucrstar.com`], a public repository for geospatial data, records each user request in the form (t, d, v, z, x, y) which contains the timestamp of the request and tile that was requested. Unfortunately, this format does not explicitly contain any user information which makes all existing recommendation models inapplicable. Therefore, we introduce a preprocessing step that adds some structure to the data that are later used by the proposed models. First, we introduce some definitions and then we describe the preprocessing algorithm.

Definition 19 (User Action) *A user action is an event initiated by the user that results in requesting different tiles from the server. User actions are pan, zoom-in, zoom-out, fly-to, change visualization, and change dataset.*

Definition 20 (Request Sequence) *Any user action results in requesting multiple tiles from the server which is denoted* request sequence. *Since all these requests are initiated at the same time, their request timestamps* $(t_1 \leq t_2 \leq ... \leq t_k)$ *are all bounded within a small range, i.e.,* $t_k - t_1 \leq \delta$ *and* δ *is a small number, e.g., 100 milliseconds.*

Definition 21 (Visualization Session) *A visualization session is a sequence of user actions performed by the same user in the same browser window. Closing the browser tab or refreshing the page would terminate the session.*

We can summarize the definitions above as follows: one user performs a series of actions within one session. Each action results in a sequence of requests. While these definitions would be very helpful in building a recommendation system, the input logs do not explicitly record any of these. Therefore, the goal of the preprocessing step is to reconstruct sessions, actions, and sequences from one large log of requests.

Reconstructing Request Sequences: We make two observations to reconstruct sequences. First, all requests within the same sequence are performed in a very short timeframe since they are all generated automatically by the browser. Second, all requests in a sequence must belong to a set tiles that could appear together on the screen, i.e., contiguous tiles in one level z for one dataset d and one visualization v.

Based on these two observations, we propose Algorithm 4 to reconstruct sequences. It starts by initializing an empty set of sequences S. Then, it scans requests ordered by timestamp t. For each request $r_i = (t_i, d_i, v_i, z_i, x_i, y_i)$, it scans all sequences in S and checks which ones the request can be added to. A request r_i can be added to a sequence s if there exist at least one request $r_j \in s$ such that: (1) $(d_i, v_i, z_i) = (d_j, v_j, z_j)$, i.e., belong to the same dataset, visualization, and zoom level, (2) $|x_i - x_j| + |y_i - y_j| = 1$, i.e., the two tiles are adjacent, and (3) $t_i - t_j \leq \delta$ and we set $\delta = 100$ milliseconds in this case. If the request r_i matches one or more sequences, all these sequences are merged into one sequence and r_i is added to it. This ensures that a sequence will all be merged together regardless of

```
1: function ExtractSequences($L = \{r_i\}$)
2:     $S = \emptyset$
3:     for $r_i \in L$ do
4:         $s_{new} = \{r_i\}$
5:         for $s \in S$ do
6:             $t_{max} = \max_{r_j \in s} t_j$
7:             if $t_i - t_{max} > \delta$ then
8:                 Remove $s$ from $S$ and write to the output
9:             else
10:                if $\exists r_j \in s$ S.T. $(d_i, v_i, z_i) = (d_j, v_j, z_j)$ and
11:                $|x_i - x_j| + |y_i - y_j| = 1$ and
12:                $|t_i - t_j| \leq \delta$ then
13:                    Merge $s$ into $s_{new}$ and remove from $S$
14:        $S = S \cup \{s_{new}\}$
15:    Write all $s \in S$ to the output
```

the time order of the requests. If r_i does not match any sequence, then a new sequence $s_{new} = \langle r_i \rangle$ is created. As the sequences are scanned, all sequences that satisfy the condition $t_i - t_{max} > \delta$, where $t_max = \max_{r_j \in s} t_j$ are considered final and they are written to the output and removed from the set of sequences S.

Reconstructing User Sessions: After we extract the sequences, the next step is to group them together into sessions. To do that, we simply merge sequences that come from the same IP address [1] and with a time difference of at most Δ where we set $\Delta =$ 4 minutes.

Reconstructing User Actions: Once sequences are grouped into sessions, we can further reconstruct user actions by comparing each pair of successive sequences in the same session. If a user switches between datasets or visualizations, this will be easily captured since each sequence contains a single dataset and visualization. If a user pans

[1] In absence of cookies and user information, IP addresses are not considered private information by most regulations

around, this will result in two sequences at the same zoom level with adjacent tiles. If a user zooms in or out, it will result in a change of one zoom level between two sequences. Finally, two sequences of the same dataset and visualization but at majorly different locations, i.e., more than one zoom level difference or no adjacent tiles, would indicate a fly-to action. The models used in this paper do not explicitly use user actions but they could be useful in the future.

5.3.2 Collaborative Filtering (CF)

This section describes how we use collaborative filtering (CF), a widely used recommendation algorithm [1], to solve our problem. A detailed description of collaborative filtering is provided in 2.0.11. In broader sense, CF is meant to build a model based on item-item similarity or user-user similarity. The key feature is that the similarity is calculated based on user ratings rather than the contents of the items. For example, let us assume that users X and Y both like items A and B; and user X also likes item C. CF would then recommend item C to user Y. We use CF to determine the relationships between tile-IDs, where each tile-ID is treated as an item chosen by a user.

In our problem, we map each session to a user and each tile (d, v, z, x, y) to an item. Since each session is considered a new user, we use an item-based CF model that tracks the similarity between items (tiles in our case). For each tile requested in one session, we consider that this tile is *liked* by the user session. By tracking all tile requests in each session, we can infer tile-tile similarity based on user behavior. As explained in Figure 5.2,

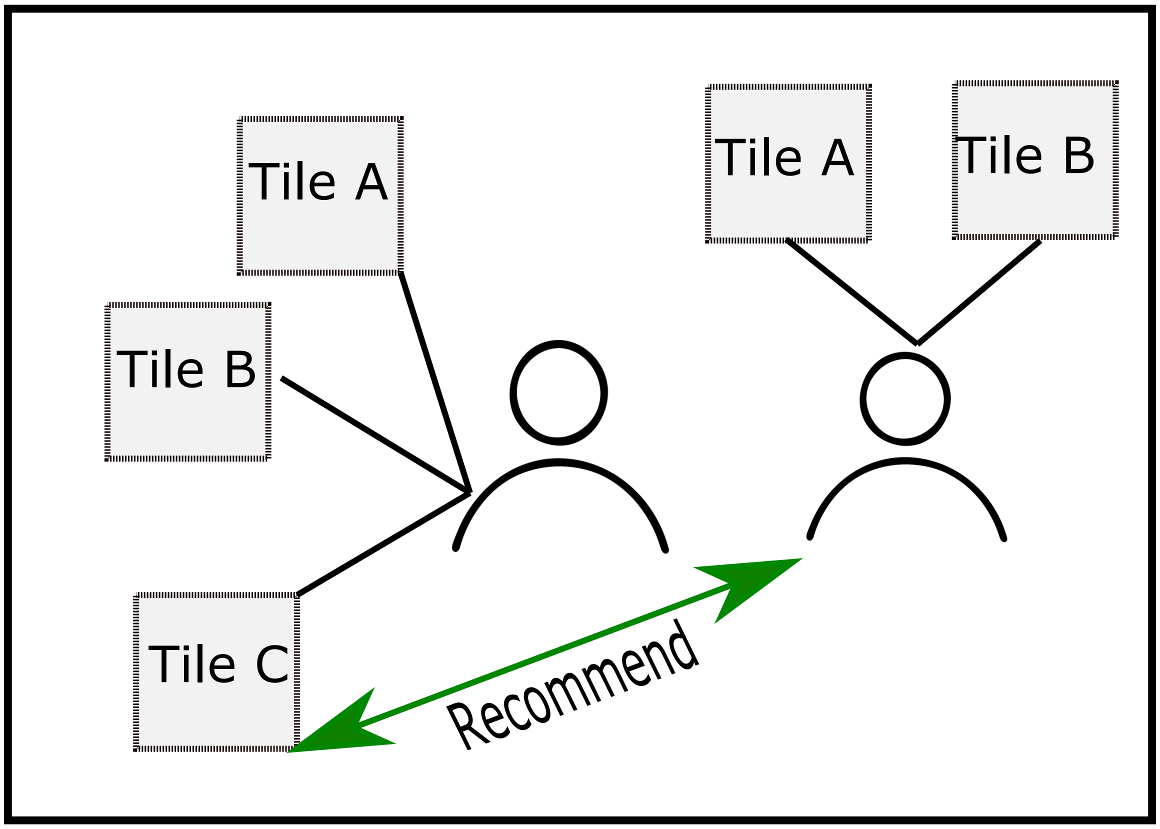


Figure 5.2: Collaborative Filtering Recommending Tile-IDs

if one user views tiles A, B, and C and a new user exploring tile A and B can then be recommended tile C.

Our proposed model works in two phases, *model building* and *recommendation*. In model building, we use complete user sessions from the historical logs to train an item-based CF model. Assuming n user sessions and m tiles, the input is represented as a matrix $\mathbf{M}$ of size $n \times m$. The entry m_{ij} is one if the user session i requested tile j. This matrix $\mathbf{M}$ is factorized to learn a User representation a $n \times d$ matrix $\mathbf{U}$, and a Tile representation a $m \times d$ matrix $\mathbf{T}$ where d is the number of features per user and tile ID..

The *recommendation* phase runs when a new session starts (a new user visits the system). It uses the initial activity of the user (e.g., tiles requested after a few user actions) and the matrix **T** obtained earlier to learn a latent representation for the user and predict his/her next activity and recommend tiles accordingly. This representation vector for the user (or matrix in case of multiple users) is then multiplied by the tile embedding matrix **T** learned in the training phase to find the similarity with all other tiles. Finally, the tiles are sorted by similarity and top-k results are recommended. This method effectively finds the top-k similar tiles to the ones that the user visited.

While CF is very popular for recommendation, it has some drawbacks specific to the proposed problem. First, all users have a very short life-span, e.g., a few minutes which makes it challenging to find interesting patterns in user behavior. Second, CF does not take the time order into account. Thus, user X visiting tiles A then B then C is treated exactly the same as another user Y visiting tiles C then B then A. It also suffers from cold start and scalability issues. The skewness of the data makes it even more difficult to work with CF.

5.3.3 RESCAL

The model of RESCAL has been described in details in section 2.0.11. Following the tensor design of RESCAL, we format our data to fit into this model.

Figure 5.3 illustrates the RESCAL decomposition method. A value x_{ijk} in the input tensor is estimated by multiplying the i^{th} row-vector of matrix **E** by the k^{th} slice of the tensor W by the j^{th} column-vector of the matrix **E**.

Following the tensor design, we format our data to fit into this model. We start with the UCR-Star logs which are decomposed into sequences and sessions. Our goal is to predict a relationship between tile IDs i and j in a sequence k. Therefore, we make an analogy by mapping *tiles* to *entities* and *sequences* to *relations*. To construct the relationship between tiles, we make the following two connections. First, for each sequence, we add a relationship between every pair of tiles within this sequence to indicate that they have been visualized simultaneously by the user. Second, for each pair of consecutive sequences s_1, s_2, we add a relationship between two tile $i \in s_1$ and $j \in s_2$ if one of the tiles is parent, grandparent, or sibling to the other according to the tile structure.

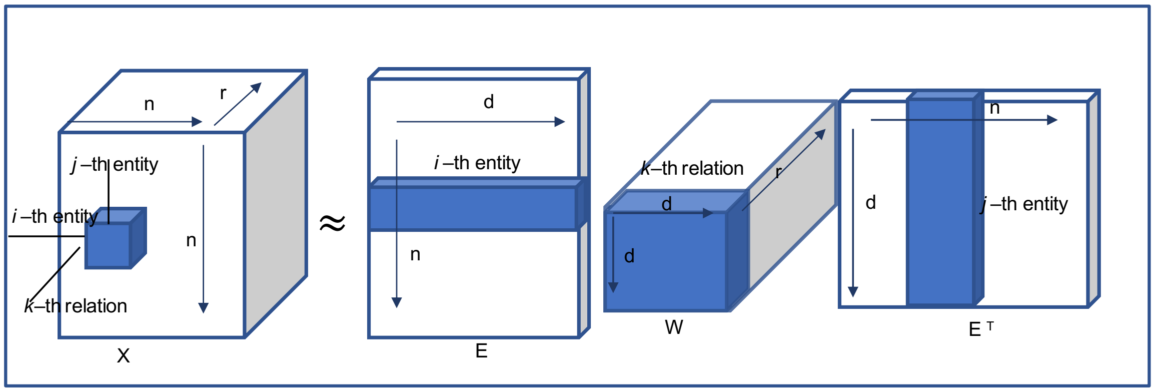


Figure 5.3: RESCAL Matrix Representation

During recommendation, after each user action we collect the set of tiles that are currently visible on the screen and try to predict the next set of tiles based on our model. To do so, we use the latent representation of the tiles in **E**, similar to CF, to find the top-k Nearest Neighbor tiles to the ones that are currently visible. Notice that the latent variables represent the collective similarity according to all previous sequences in the training set. This method has been extensively used in knowledge graphs to find groups of related entities [34] and we propose to use it to find related tiles. This approach is slightly different than just using the scores of a pair of nodes to predict an edge between the node, but given the optimization objective, the latent representation space embeds similar nodes close by and dissimilar nodes far apart, thereby making a K-Nearest Neighbor approach feasible. Another approach just as valid, but slightly more computationally demanding will be to compute pairwise scores for all pairs of nodes and sort the score per node in descending order to obtain some recommendation for each given node.

5.3.4 Graph Convolutional Networks (GCN)

A detailed description of GCN is provided in section 2.0.11. To prepare our data to fit into a GCN, we create a computation graph consisting of all the tile-IDs that are obtained from UCR-Star logs. Each tile-ID $(d_i, v_i, z_i, x_i, y_i)$ is represented as a node in the graph. Each set of tile-IDs explored by the user at each sequence k are represented in different layers of this model. The construction of the computation graph, can be summarized as:
a) All the tile-IDs are represented as nodes in the graph; b) All the tile-IDs visited by the user at a given sequence s_1 have a relation to each other; and c) For a pair of successive sequences s_1 and s_2, we add a relationship in s_2 to establish the connection between two tiles $i \in s_1$ and $j \in s_2$, given if one of them is a parent, child, or grand child or sibling of any of the nodes. In essence, this is similar to how we construct the relationships in RESCAL model in Section 5.3.3 except that we do not keep any information about the sequence IDs,

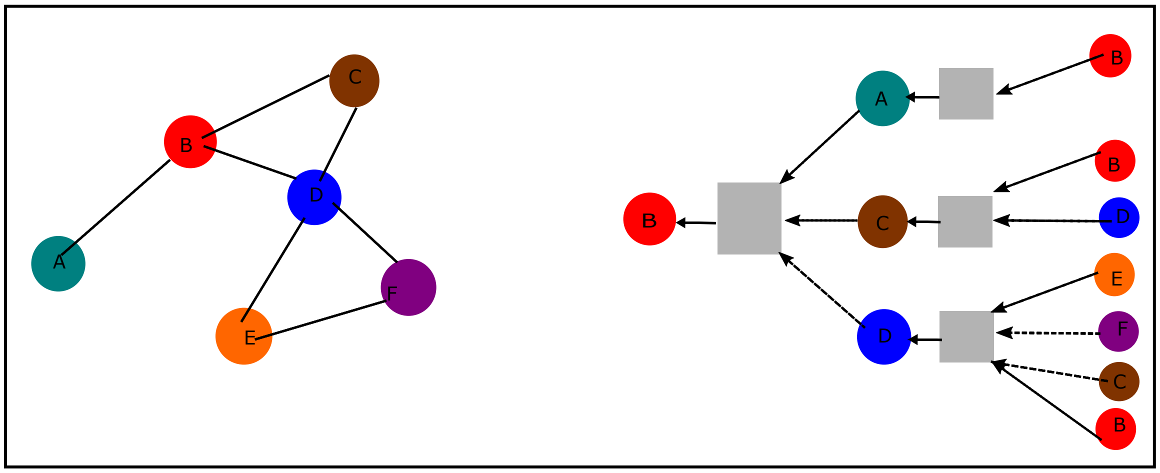


Figure 5.4: Nodes (Tile-IDs) in GCN getting trained by neighboring nodes

i.e., all connections are finally merged into one graph. To further contrast them, the input to GCN is an $n \times n$ adjacency matrix while for RESCAL the input is $n \times n \times m$ tensor.

Given a node i of the computational graph G, we can use the features of the neighbors and neighbors of neighbors of the nodes to propagate and aggregate information on each of these nodes. This creates an output matrix $\mathbf{Z}$ of dimension $n \times f$.

Figure 5.4 shows an example of GCN. The computational graph is constructed based on the relationship of the tile-IDs. The figure to the right represents the graph in the GCN network. Node B of an input graph, embeds itself from its neighbors and neighbors of neighbors. The small boxes in the figure represents hidden layers of neural network that aggregate information. Though there is no strict rule regarding the depth of the neighbors that should be considered for building feature of nodes, we should not go too deep such that the entire graph becomes a part of embedding of a single node. So we have kept the number of layers to two in our model.

When a new user visits UCR-Star, based on the current location of the user we derive a set of tile-IDs that are currently visible. For each of these tile-IDs i, we can compute the similarity vector from Z, for any other node j. This in turn helps us predicting the next best location for the new user.

5.4 Experiments

This section provides a detailed experimental evaluation of our work using different models, datasets and parameters. We primarily highlight the following points in this section.

- How the different models suit our work.
- How the logs are used for different recommendations tasks.
- Evaluate the accuracy of the models using standard methods.
- Measure the scalability of the models with increase in data size as well as data variation.

5.4.1 Experimental Setup

All the experiments are conducted in a deep learning Amazon Machine Image (AMI) of Amazon Web Service (AWS) with Ubuntu 18.04 for scalability and reproducibility. It provides Python 3, PyTorch 1.5 and nVIDIA CUDA alongside other deep learning features. We also used PyTorch Geometric for deep learning extensions within the PyTorch. The data processing as well as the models are implemented using Python 3.5 in an Ubuntu 16.04 machine with a 31GB RAM. Table 5.1 lists the datasets and their descriptions that

Table 5.1: Training and test datasets

Dataset	# records (training)	# records (test)	Description
All-Nodes	10102	1917	All points on the map from OSM
Buildings	65276	16320	Building footprint from OSM
Multiple	100629	30787	Combination of multiple datasets traversed by different users

are used in this work for training and testing the models. The number of records in the table denotes the number of tile requests in the logs. However, we note that the preprocessing step that prepares the data for each model might considerably change these numbers. We will talk about this more in the following subsections. We use the acronyms CF and GCN to refer to Collaborative Filtering and Graph Convolutional Network, respectively. For the performance measurement, we have primarily used precision-recall curve which is often referred as PR-Curve. In this section we focus on the AOI recommendation task and move dataset recommendation.

5.4.2 Evaluation Metrics

It is important to measure the effectiveness of the proposed model for recommendation. In our case, the models are expected to predict the next set of tile requests that the user makes which might refer to a different area in the dataset or a different dataset. We want to make sure that the recommendations by the models are meaningful. However, accuracy is not always the best tool for such measurement as it depends on the distribution of the labels. For example, amongst a group of 1,000 people, if we tag everyone as disease free, including the 20 people who are sick, the model has an accuracy of 98%. It is indeed a high number, but yet not a good model. Similarly, if the model recommends all tiles to the user, the real visited tiles will all be included but the result would be irrelevant. In order to resolve this, we use PR-curve for an evaluation metric. The details of the PR-curve can be found in section 2.0.13.

For definitive and robust working of the models, we introduce **negative relations** in the test set. Negative relations can be defined as a no-relationship condition. In case of CF when sessions and tile-IDs are not connected, we can call them a negative relation. Similarly, for RESCAL and GCN when there are no connections between two tile-IDs we can call them negative relations. Hence **positive relations** are connected tile-IDs for GCN and RESCAL, and existing relations between sessions and tile-IDs. Varying the negative relations in the test datasets can be useful in proving the sturdiness of the models. If the models can classify negative and positive relations correctly, as we vary the number of negative edges in the test data, that will be considered a strong component of the models. Such models would make more relevant recommendations

In CF, the negative relations are all the edges that are not positive. But in RESCAL and GCN, we generate random edges, that are not listed as positive, and mark them as negatives. This is also called an *open world* model. RESCAL and GCN do not assume all the non-positive edges to be negatives. Hence it is important for us to define the negative relations, so that the model trains not only for similarities but also dissimilarities between nodes. By default we keep the ratio of positive to negative edges as 1:1. The number of negative relations in all three models can be varied in the test cases. While testing the model, our aim is to detect the percentage of correct predictions and also to determine how correctly we can classify positive and negative relations. PR-curve helps us with this evaluation.

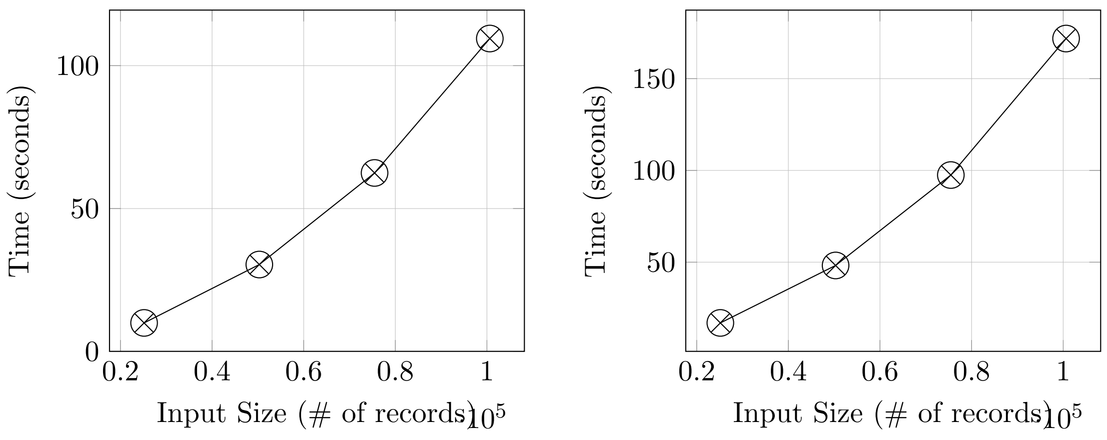
Each model gives out a prediction (score) as an output that establishes a relation between the entities of these models (session-tile-ID for CF, tile-ID-tile-ID for RESCAL and GCN). Based on this we create a threshold (a number between 0 and 1), which decides if a prediction (score) is positive or negative. In our experiments, we have used multiple thresholds from 0 to 1, to generate the PR-Curves. Interested readers can refer to Appendix 2.0.13 for further explanation of the PR-Curve.

The added benefit of PR testing with a ratio of positive examples and negative examples is that if the PR curve drops as we test with an increasing number of negative examples when compared to positive, it generally is an indication that the model has a tendency to recommend negative results over positive results and each positive prediction by the model at a lower threshold score might not be a good recommendation for the user.

As long as the PR curve is high, this implies that model is learning latent representations for tiles that facilitate better prediction scores, thus using nearest neighbour embeddings for making recommendations is a justified choice.

5.4.3 Data Processing Phase

This phase transforms the UCR-Star logs to feed them into the models. It takes the raw logs and adds structure to it by extracting sequences, sessions, and tile-tile relationships. Figure 5.5 shows the computation time for preparing these data. CF only requires breaking



(a) CF

(b) RESCAL and GCN

Figure 5.5: Data processing time for CF, RESCAL and GCN

the input into sessions so it is relatively faster. However, RESCAL and GCN need an extra step that adds tile-tile relationships between sequences as explained in Sections 5.3.3 and 5.3.4.

In Figure 5.5, we increase the input size on the x-axis and measure the total processing time on the y-axis. The input is obtained by taking subsets of the *multiple* dataset from 25k records to 100k records. We show the running times of the two preprocessing methods for CF and RESCAL+GCN. As expected, the processing time increases steadily with the increase of input size. The time goes up almost linearly for both methods since it requires one scan over the data. However, the computation of an adjacency list in RESCAL and GCN is about 150% slower than dividing the input into sessions in CF.

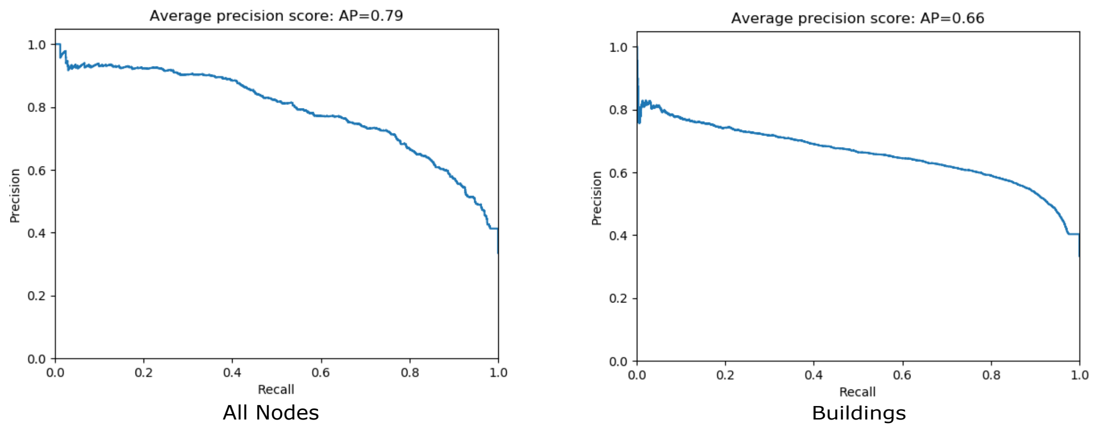


Figure 5.6: PR curve for Collaborative Filtering (CF)

5.4.4 Collaborative Filtering (CF)

This experiment measures the accuracy of the CF algorithm for AOI recommendation. We leave the dataset recommendation on the *multiple* dataset to Appendix 5.5. For each case, we plot the PR-curve to evaluate the model performance. As shown in Figure 5.6, CF does not show a very promising result with less than 80% average precision score. CF has several drawbacks for the visualization recommendation problems that cause this poor results. First, we recall from Section 5.3.2 that we do not have user information and we treat each session as a new user. This results in a very sparse user-item relationship that causes CF to act poorly. Second, the CF model does not keep any information about the time order of the requests. In other words, it does not know which tiles came first and which tiles came next. This means that all user actions are simply merged together. This results in poor recommendation since CF might recommend tiles that are not close to the set of tiles currently visualized by the user.

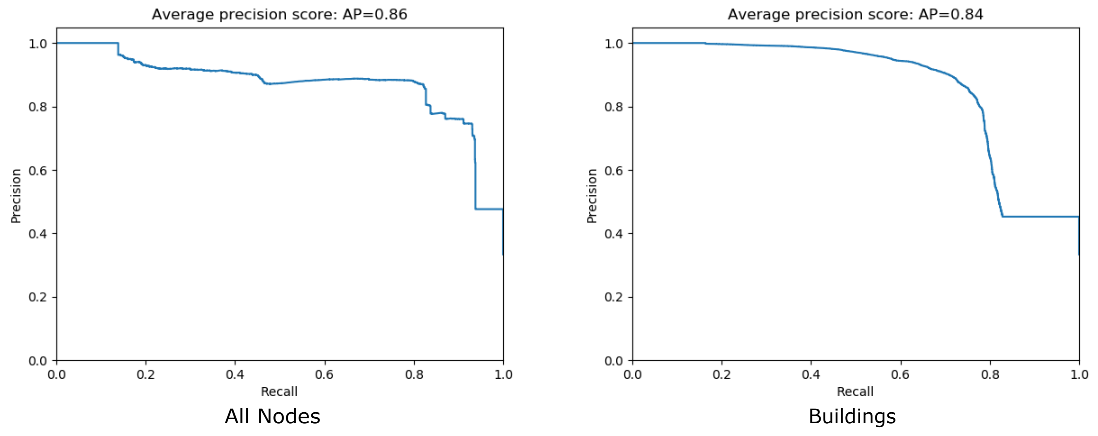


Figure 5.7: PR curve for RESCAL

5.4.5 RESCAL

Figure 5.7 provides the PR-Curve of RESCAL with various inputs from the small (*all-nodes*) to the large (*building*) datasets (for *multiple datasets* see Appendix 5.5). The average precision score ranges from 75% to 80%. For the large dataset, the precision starts to fall as the problem becomes harder due to the increasing number of tiles in the dataset. Unlike CF, this method captures the relative ordering of requested tiles by building tile-tile relationships based on the sequences. This additional relationship significantly improves the performance as shown in this experiment. In addition, the tile-tile relationship increases the size of the dataset significantly which helps in the training phase as it provides a good opportunity of extracting interesting patterns.

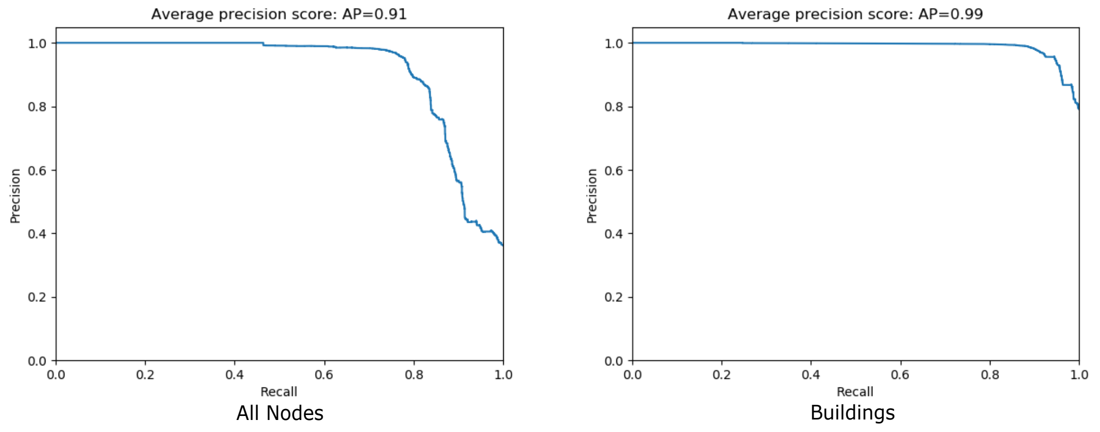


Figure 5.8: PR curve for GCN

5.4.6 Graph Convolution Network (GCN)

Figure 5.8 represents the PR-curve of GCN. The results of GCN are significantly better than both CF and RESCAL. In addition, it performs much better with the large dataset whereas the two other methods could not handle the big dataset well. This can be attributed to the way that GCN builds its model. GCN constructs its model based on the tile-tile graph that we feed as input. This allows the model to keep track of the sequencing information and recommends tiles that are close to the requested tiles. Moreover, as the test data grows, and more number of negative edges are introduced to the system, the system continues to classify the correct positive and negative edges with proper relevance.

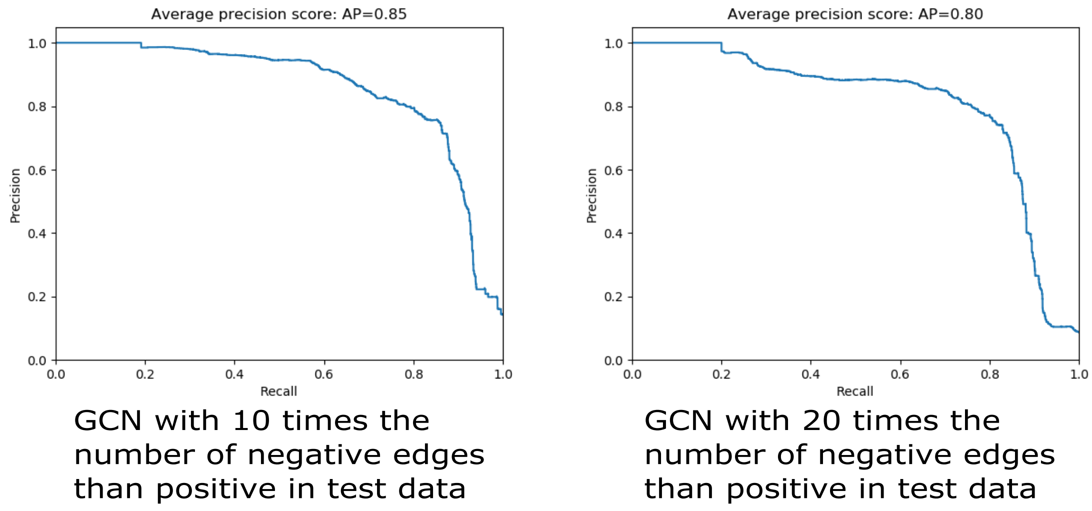


Figure 5.9: PR curve showing results of increasing negative edges in test data for GCN

5.4.7 Tuning the test cases

As we had explained in Section 5.4.2, introducing the negative relations plays an important role in determining the strength of the models. In this experiment, we vary the amount of negative relations that we introduce and measure the performance using the PR-Curve. In particular, we vary the ratio between negative and positive weights between 10:1 and 20:1. Such a test can help us determine the robustness of a method. Due to the limited space, we only show the results of GCN as the best performing algorithm so far but we present more results in the appendix.

Figure 5.9 plots the PR-curve GCN for 10:1 and 20:1 negative-to-positive edge ratios. As we can observe, the performance gets lower as the ratio of negative edges increase. Despite losing some precision, GCN still manages to hold the curve characteristics. Besides the average precision of GCN falls to around 80%, which is still a sign of a good model.

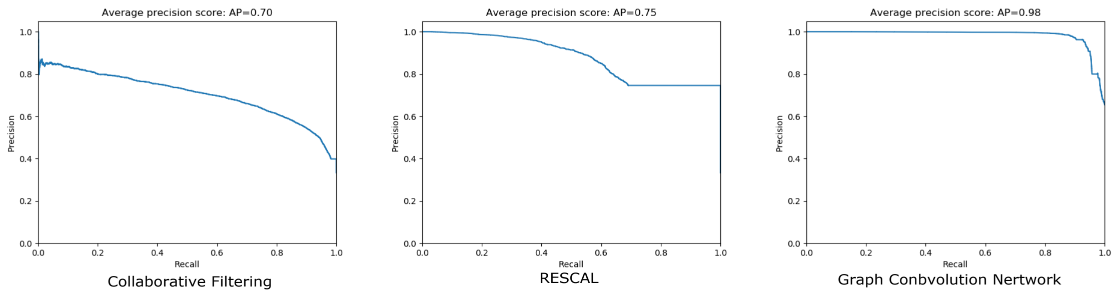


Figure 5.10: PR curve for CF, RESCAL and GCN on Multiple Dataset

These results resonate with our expectations that GCNs tend to learn the similarities and dissimilarities between nodes that were explored by the user in a better manner, which is why they are not as susceptible other models. We can further conclude, that GCN is more suited for our recommendation system.

5.5 Dataset Recommendation

This section is primarily used to provide the results of some important experiments, that we could not provide in the main paper due to lack of space.

5.5.1 Experimental Results of the Models for Dataset Recommendation

As explained in section 5.2.2, recommending different dataset is a part of this problem. We provide some experiments to show how our implemented models are capable of dataset recommendation alongside POI recommendation. We have used UCR-star logs to capture the user behavior of switching from one dataset to another. The dataset is referred as Multiple in 5.1. Figure 5.10 represents PR-curve of Multiple dataset logs into Collaborative

Filtering, RESCAL and GCN. As seen in case of other datasets, CF does not show a very promising result. RESCAL on the other hand, performs significantly well with an average precison of 75%, but starts deteriorating with increase in data size. Finally, as anticipated GCN performs the best out of the three, with an average precision of 98%.

5.6 RESCAL: with and without definite negative relations

We have discussed about negative relations in section 5.4.2. We have also described that in RESCAL the number of negative relations in the training set are generated by a default ratio of 1:1. However this ratio is modifiable. It should be noted, that RESCAL can be designed in a *closed world* model, as well as an *open world* model. In a **closed world** model, all the relations that are not positive are considered negative, whereas in **open world**, the negative relations are sampled, and only the sampled ones are considered negative and the non sampled edges are considered as missing. Figure 5.11 shows PR-curve for closed (v1) and open (v2) world implementation of RESCAL for All Nodes. As we can see that v1 has an average precision of 67%, whereas v2 has an average precision of 84%. Besides v2 can classify the Tile-IDs with a high precision for more than 90% of the data in v2 while in v1 it stops classifying beyond 60% of the records. This is an expected trait. Instead of learning the dissimilarities between tile-IDs, v1 trains on such relations as *unknown*. Whereas in v2, the model explicitly trains itself with the dissimilarities between the same number of tile-IDs as it trains for similarities. This enhances the quality of the training, as a result producing better results. This is the primary reason for which we stuck to v2 of RESCAL for the recommendation.

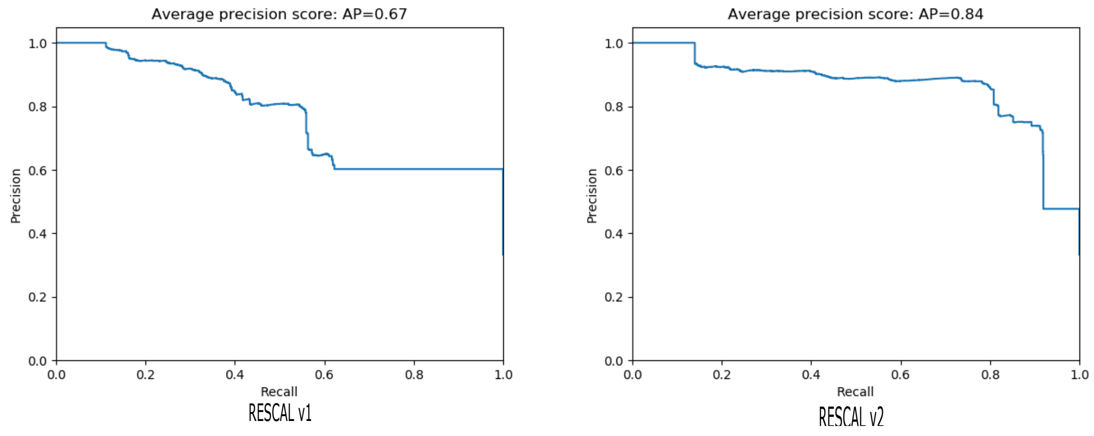
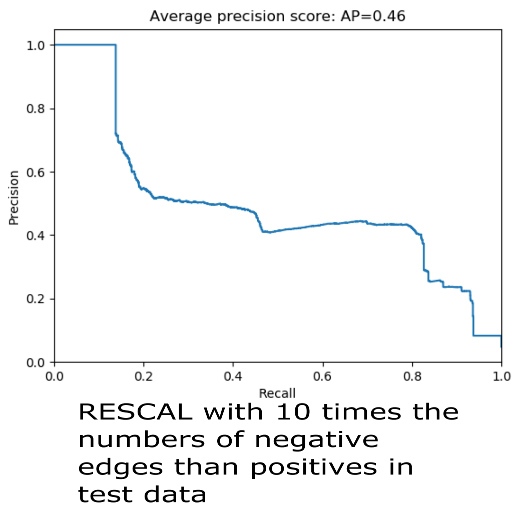


Figure 5.11: PR curve for RESCAL's two versions

5.6.1 Tuning the test cases for RESCAL

As we have seen in section 5.4.7 that increasing negative relations in the test case affects the average precision of GCN adversely.

Figure 5.12 plots the PR-curve RESCAL with negative to positive test ratio being 10:1 and 20:1. Unlike GCN, RESCAL loses a considerable amount of it average precision (46% for 10 and 31% for 20) and almost completely loses the curve structure. This proves the shortcomings of RESCAL. Models like RESCAL tend to excessively predict the negative relations as false positives, thereby making the recommendation performance poor.



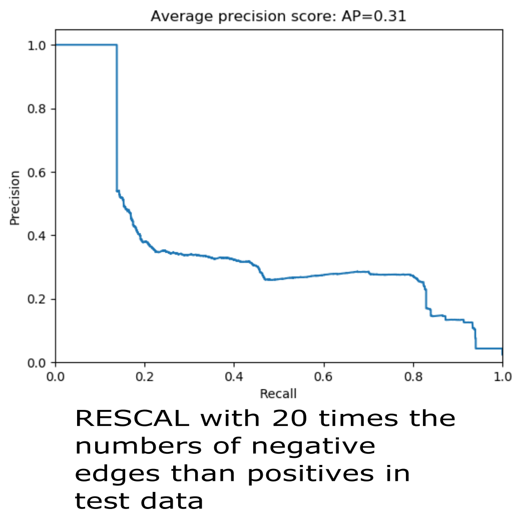


Figure 5.12: PR curve showing results of increasing negative relations in test cases for RESCAL

Chapter 6

Future Scope and Conclusion

This work on Geospatial data opens up a number of future research options. From the visualization perspective to the computational back-end, this system has versatile open ends.

- **Incorporating multiple visualization in the recommendation system:** Despite doing early work on different kinds of visualization for the same dataset like heat map, scatter plot etc, it has not been introduced in UCR-Star. As a result, the logs in UCR-Star never captured users changing visualization, which resulted in me never testing it on our recommendation models.

- **Enhancing UCR-Star with an online recommendation system:** It will also be nice to implement the recommendation system on UCR-Star, where users can experience how the recommendation will work. It will also help in creating online recommendation, where a user accepting a recommendation by the system gets higher points, and users neglecting a recommendation give it a negative score.

- **Refine GCN, with evolve GCN in recommendation:** Adding more information to these recommendation models might increase the quality of these recommendations. So adding additional information like records counts, image pattern, geotagged information to the existing models can be a good test for the system.

- **Add additional visual analysis to user specific queries:** Geospatial analysis is an upcoming demand. Many people not only wants to visualize spatial data, but also wants to analyse its properties. Adding the feature to UCR-Star will be exciting.

- **Making the visualization more attractive:** The main visualization can be enhanced many folds by smaller improvements like adding tags to the location, adding some more attractive looking layer, introducing 3-D visualization etc.

- **Integrating user behavior in the tile classification:** User behavior that can be captured via UCR-Star logs can be a good parameter for measuring which locations are commonly visited by users. This information can be used to define the threshold in AID/AID* instead of using record counts.

- **Visual exploration of data stream :** So far all our works are based on static data. Working on live data stream can be an excellent idea. It will be interesting to see how our present work scales with data stream and what new components might be necessary.

In conclusion, my work on creating a user friendly, open-sourced geospatial repository can be of immense help to the entire scientific and non-scientific community. The aim is to make it a go-to repository for anyone trying to explore any geospatial data. We started with 123 datasets and now we have 152 and this number will continue to increase. It is possible to store so many of these datasets due to the extremely small sized indexes of AID*. With the recommendation system up and running, this can be a visual dictionary for geospatial data.

Bibliography

[1] Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295, 2001.

[2] Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of International conference on World Wide Web 2009*, April 2009. WWW 2009.

[3] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 299–310, 1995.

[4] Charou et al. *Integrating Multimedia GIS Technologies in a Recommendation System for Geotourism*, pages 63–74. Berlin, Heidelberg, 2010.

[5] Harry Chasparis et al. Experimental evaluation of selectivity estimation on big spatial data. In *GeoRich@SIGMOD*, pages 8:1–8:6, 2017.

[6] Chi Yin Chow et al. Towards location-based social networking services. In *SIGSPATIAL*, pages 31–38, December 2010.

[7] datauk. `https://www.data.gov.uk/`.

[8] dataus. `https://www.data.gov/`.

[9] Muhammad El-Hindi et al. VisTrees: Fast Indexes for Interactive Data Exploration. In *HILDA@SIGMOD*, page 5, 2016.

[10] Ahmed Eldawy et al. SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data. In *ICDE*, pages 1585–1596, 2015.

[11] Ahmed Eldawy et al. HadoopViz: A MapReduce Framework for Extensible Visualization of Big Spatial Data. In *ICDE*, pages 601–612, 2016.

[12] Saheli Ghosh et al. AID: An Adaptive Image Data Index for Interactive Multilevel Visualization (Poster). In *ICDE*, 2019.

[13] Saheli Ghosh et al. UCR-STAR: The UCR Spatio-Temporal Active Repository. *SIGSPATIAL Special*, 11(2):34–40, December 2019.

[14] Gotz et al. Behavior-driven visualization recommendation. page 315–324, Sanibel Island, FL, 2009.

[15] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, 1984.

[16] Hauger et al. Exploring geospatial music listening patterns in microblog data. In Andreas Nürnberger, Sebastian Stober, Birger Larsen, and Marcin Detyniecki, editors, *Adaptive Multimedia Retrieval: Semantics, Context, and Adaptation*, pages 133–146, Cham, 2014. Springer International Publishing.

[17] He et al. Lightgcn: Simplifying and powering graph convolution network for recommendation. *arXiv preprint arXiv:2002.02126*, 2020.

[18] Kevin Zeng Hu et al. Vizml: A machine learning approach to visualization recommendation. page 128, Glashow, Scotland, UK, May 2019. ACM.

[19] Jianfeng Jia et al. Towards Interactive Analytics and Visualization on One Billion Tweets. In *SIGSPATIAL*, pages 85:1–85:4, 2016.

[20] Thomas N. Kipf et al. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[21] Kong et al. Decompressing knowledge graph representations for link prediction. *arXiv preprint arXiv:1911.04053*, 2019.

[22] Kywe et al. On recommending hashtags in twitter networks. In *International conference on social informatics*, pages 337–350. Springer, 2012.

[23] Justin J. Levandoski et al. LARS: A Location-Aware Recommender System. In *ICDE*, pages 450–461, Arlington, VA, April 2012.

[24] Jeff W Lichtman et al. The Big Data Challenges of Connectomics. *Nature Neuroscience*, 2014.

[25] Lauro Lins et al. Nanocubes for Real-Time Explorations of Spatiotemporal Datasets. In *TVCG*, 2013.

[26] Zhicheng Liu et al. *imMens*: Real-time Visual Querying of Big Data. volume 32, pages 421–430, 2013.

[27] Zhicheng Liu and Jeffrey Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *TVCG*, 20(12):2122–2131, 2014.

[28] Land Process Distributed Active Archive Center, 2015. `https://lpdaac.usgs.gov/about`.

[29] MapBox, 2019. `https://www.mapbox.com/`.

[30] Mapzen, 2017. `https://mapzen.com/`.

[31] Richard Michael et al. An Analytic Data Engine for Visualization in Tableau. In *SIGMOD*, pages 1185–1194, 2011.

[32] Todd Mostak. An Overview of MapD (Massively Parallel Database). Harvard Technical Report. 2015. `http://geops.cga.harvard.edu/docs/mapd_overview.pdf`.

[33] Nickel et al. Factorizing yago: scalable machine learning for linked data.

[34] M. Nickel et al. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

[35] Maximilian Nickel et al. A Three-Way Model for Collective Learning on Multi-Relational Data. pages 809–816, Bellevue, WA, July 2011.

[36] OmniSci. `https://www.omnisci.com/`.

[37] OpenLayers, 2019. `https://openlayers.org/`.

[38] OpenStreetMap, 2019. `https://www.openstreetmap.org/`.

[39] OpenStreetMap disk usage, 2019. `https://wiki.openstreetmap.org/wiki/Tile_disk_usage`.

[40] Moon-Hee Park et al. Location-Based Recommendation System Using Bayesian User's Preference Model in Mobile Devices. In *Ubiquitous Intelligence and Computing*, pages 1130–1139, Berlin, Heidelberg, 2007.

[41] Yongjoo Park et al. Visualization-aware Sampling for Very Large Databases. In *ICDE*, pages 755–766, 2016.

[42] Gary Planthaber et al. EarthDB: scalable analysis of MODIS data using SciDB. In *BigSpatial*, pages 11–19, 2012.

[43] Lakshmish Ramaswamy et al. CAESAR: A context-aware, social recommender system for low-end mobile devices. In *MDM*, pages 338 – 347, 2009.

[44] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[45] Mohamed Sarwat et al. LARS*: An Efficient and Scalable Location-Aware Recommender System. 26(6):1384–1399.

[46] Sentinel-2 - European Space Agency, 2016. `http://www.esa.int/Our_Activities/Observing_the_Earth/Copernicus/Sentinel-2`.

[47] T.Horozov et al. Using location for personalized POI recommendations in mobile environments. pages 6 pp.–129, 2006.

[48] Helga Thorvaldsdóttir et al. Integrative Genomics Viewer (IGV): High-performance Genomics Data Visualization and Exploration. volume 14, pages 178–192, 2013.

[49] Twitter. The About Webpage. Available at `https://about.twitter.com/company`, 2015.

[50] United nations open data, 2019. `http://data.un.org/`.

[51] D. C. Van Essen and H. A. Drury. Structural and functional analyses of human cerebral cortex using a surface-based atlas. *Journal of Neuroscience*, 17(18):7079–7102, 1997.

[52] Vartak et al. Towards visualization recommendation systems. *SIGMOD Rec.*, 45(4):34–39, May 2017.

[53] Tin Vu and Ahmed Eldawy. R-Grove: Growing a Family of R-trees in the Big Data Forest. In *SIGSPATIAL*, pages 532–535, 2018.

[54] Tin Vu, Sara Migliorini, Ahmed Eldawy, and Alberto Bulussi. Spatial data generators. In *1st ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2019)*, page 7, 2019.

[55] World bank open data, 2019. `https://data.worldbank.org/`.

[56] wu et al. Socialgcn: An efficient graph convolutional network based model for social recommendation. *arXiv preprint arXiv:1811.02815*, 2018.

[57] Ying et al. Graph convolutional neural networks for web-scale recommender systems. pages 974–983, 2018.

[58] Jia Yu et al. GeoSparkViz: A Scalable Geospatial Data Visualization Framework in the Apache Spark Ecosystem. In *SSDBM*, pages 15:1–15:12, 2018.