UNIVERSITY OF CALIFORNIA
RIVERSIDE

Geospatial Computing from Data Lakes to Deep Learning Applications

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

Doctor of Philosophy

in

Computer Science

by

Majid Saeedan

June 2025

Dissertation Committee:
Dr. Ahmed Eldawy, Chairperson
Dr. Amr Magdy
Dr. Evangelos Christidis
Dr. Zhijia Zhao

The Dissertation of Majid Saeedan is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First, I extend my sincere gratitude to my advisor, Dr. Ahmed Eldawy, for his guidance, critical insights, and unwavering support throughout my research. His mentorship has been instrumental throughout this journey.

I also want to acknowledge the members of my dissertation committee: Dr. Amr Magdy, Dr. Evangelos Christidis, and Dr. Zhijia Zhao. Their valuable feedback and encouragement were essential for the completion of this work.

I am indebted to my collaborators who contributed to various aspects of this research. Dr. Zhijia Zhao, for his collaboration on the work presented in Chapter 3. Dr. Alberto Belussi and Dr. Sara Migliorini, from the University of Verona, for their collaboration on the work presented in Chapter 4. Dr. Evangelos Christidis and Dr. Muhammad Shihab Rashid for their collaboration on the work presented in Chapter 5.

I would also like to acknowledge the Department of Computer Science and Engineering for providing a supportive and welcoming environment throughout my graduate studies. Additionally, I thank all the staff of the department whose efforts behind the scenes ensured that I could focus on my research.

Finally, I'd like to acknowledge that this work was supported in part by the National Science Foundation (NSF) under grants IIS-1954644, CNS-1924694, and IIS-2046236.

To my parents, my family, and my friends, for their unwavering support, encouragement, and belief throughout this journey.

ABSTRACT OF THE DISSERTATION

Geospatial Computing from Data Lakes to Deep Learning Applications

by

Majid Saeedan

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2025
Dr. Ahmed Eldawy, Chairperson

This thesis explores geospatial vector data, including geometric shapes such as points, lines, and polygons. This data is crucial in navigation, urban planning, and many more applications. Geospatial computing is a multidisciplinary field that focuses on creating techniques and tools to handle large geospatial datasets.

Given the reliance on data lakes to store large data sets in their raw formats, it is critical to have full support for geospatial datasets to enable scalable processing. To address this, we make two contributions in this area. First, we propose a column-oriented binary format called Spatial Parquet, which integrates geospatial vector data into Apache Parquet that enables significant data compression and efficient querying. Second, to improve support for semi-structured data, we introduce a distributed JSON processor for scalable SQL queries on large JSON datasets, including GeoJSON. It processes complex datasets like Open Street Map with features such as projection and filter push-down.

Advances in Deep Learning (DL), including foundation models and Large Language Models (LLMs), offer opportunities for geospatial data analysis. We make three main

contributions in this area. First, we study how to design DL models that can express a wide range of geospatial functions. We explore three representations: an image-based representation using geo-referenced histograms (GeoImg), a graph-based point-set representation (GeoGraph), and a vector-based representation using a Fourier encoder (GeoVec). We formalize these representations and design corresponding models: ResNet and UNet for the first, PointNet++ for the second, and Poly2Vec with Transformers for the third. We evaluate all approaches on four spatial problems, showing the accuracy and effectiveness of the three approaches. Second, we create a benchmark called GS-QA for evaluating spatial question-answering with LLMs. A semi-automated process generates diverse question-answer pairs that cover various spatial objects, predicates, and complexities. An evaluation methodology is suggested with some experiments. Finally, a prototype for generating geospatial vector data from text prompts, called GeoGen I, is proposed. It has potential for applications such as spatial interpolation, data augmentation, and change analysis. We adapt diffusion models, traditionally used for generating realistic images, as geospatial data generators. We also explore their use for similarity search through geospatial data embeddings, highlighting the potential of vector databases in this domain.

This thesis advances geospatial data processing, storage, analysis, and generation, opening new research pathways in geospatial computing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis explores geospatial vector data, which includes geometric shapes such as points, lines, and polygons related to positions on Earth. Points can represent precise locations, such as a specific address. Lines are used to denote paths, such as roads, or rivers. When a line starts and ends at the same point, it is called line ring. Polygons are made up of one or multiple line rings and are used to define the outlines of geographic areas, such as city boundaries, lakes, or even small buildings. Geospatial vector data are crucial in diverse areas such as navigation; urban planning; environmental science; and many more. Geospatial computing is a multidisciplinary field that combines elements from geography and computer science to develop techniques and tools for handling and analyzing large geospatial datasets. In this thesis, we explore two aspects for this data. We start by exploring the research topic of data lakes and how we can improve their support for geospatial data. Secondly, we explore deep learning applications in relation to geospatial computing to take advantage of the most recent advancements in this field.

Data lakes are data repositories designed to store, manage, and analyze large volumes of raw data. Various types of data are stored in data lakes, including structured, semi-structured, and unstructured data, which allows organizations to consolidate information from various sources. Common formats used in data lakes include CSV, JSON, and Parquet. Each format has its own advantages and disadvantages, from human readability, flexibility in writing, and efficient processing. Due to the importance of data lakes for handling large data sizes, it is crucial to have full support for geospatial datasets in them. We make two contributions in this area. First, we propose Spatial Parquet which proposes a structure and encoding for geospatial data, making it natively supported in Parquet. Second, we propose dsJSON a system for distributed processing of JSON datasets, including GeoJSON, that makes it possible to support running SQL queries using Spark on complex datasets, like Open Street Map (OSM).

Recent advances in deep learning have profoundly influenced many fields. With the development of foundation models and Large Language Models (LLMs), researchers are now exploring the various ways that they can be used to address challenging problems. As such, it is crucial to also explore how these advances can be utilized for geospatial problems. We make three main contributions in this area. In the first contribution, we performed a study to learn how to best represent spatial problems that involve a large number of points using existing deep learning architectures. Second, we created a benchmark for evaluating LLMs on geospatial question-answering. Finally, we explore the are of generative models and propose an initial prototype for generating geospatial data.

The second chapter discusses Spatial Parquet, which is introduced because for modern data analytics applications column-storage formats are preferred, due to their improved storage efficiency through encoding and compression. Parquet is the most popular file format for column data storage that provides several of these benefits out of the box. However, geospatial data are not readily supported by Parquet. Spatial Parquet is a Parquet extension that efficiently supports geospatial data. It inherits all the advantages of Parquet for non-spatial data, such as rich data types, compression, and column/row filtering. Additionally, it adds three new features to accommodate geospatial data. First, it introduces a geospatial data type that can encode all standard spatial data types in a column format compatible with Parquet. Second, it adds a new lossless and efficient encoding method, termed FP-delta, that is customized to efficiently store geospatial coordinates stored in floating-point format. Third, it adds a light-weight spatial index that allows the reader to skip non-relevant parts of the file for increased read efficiency. Experiments on large-scale real data showed that it can reduce the data size by a factor of three even without compression. Compression can further reduce the storage size. Additionally, Spatial Parquet can reduce the reading time by two orders of magnitude when the light-weight index is applied. This initial prototype can open up new research directions to further improve geospatial data storage in column format.

In the third chapter, we propose a system called dsJSON. The popularity of JSON as a data-interchange format resulted in big amounts of datasets available for processing. Users would like to analyze this data using SQL queries, but existing distributed systems limit their users to only two specific formats, JSONLine and GeoJSON. The complexity of

JSON schema makes it challenging to parse arbitrary files in a modern distributed system while producing records with unified schema that can be processed with SQL. To address these challenges, we introduced dsJSON, a state-of-the-art distributed JSON processor that overcomes limitations in existing systems and scales to big and complex data. dsJSON introduces the projection tree, a novel data structure that applies selective parsing of nested attributes to produce records that are ready for SQL processors. The key objective of the projection tree is to parse a large JSON file in parallel to produce records with a unified schema that can be processed with SQL. dsJSON is integrated into SparkSQL which enables users to run arbitrary SQL queries on complex JSON files. It also pushes projection and filter down into the parser for full integration between the parser and the processor. Experiments on up to two terabytes of real data show that dsJSON performs several times faster than existing systems. It can also efficiently parse extremely large files not supported by existing distributed parsers.

The fourth chapter explores representations of geospatial vector data for deep learning. There are multiple ways to format these data for deep learning models. In this chapter, we study how to use three popular DL architectures for dealing with geospatial data. The first one is an image-based architecture, where the geospatial data is first pre-processed into a fixed size geo-referenced histogram that substitutes the image. We use *ResNet* [85] and *UNet* [176] as reference models for the image-based architecture. The second one is a graph-based architecture, where the input of the considered model is a set of points having position coordinates, and each point is associated with additional non-spatial attributes. We adapt this approach by designing and testing variations of the

PointNet++ [169] architecture. The third is a vector-based model that encodes geometries into fixed-size vectors. To test this approach, we extend the Poly2Vec [188] encoder to work with a Transformer [206]. Furthermore, we model four different spatial problems using the three representations. This includes spatial data synopsis, spatial clustering, selectivity estimation, and walkability estimation. These problems involve different spatial operations and data characteristics, providing a lot of insights. Our goal is to study how the three proposed data representations and model architectures can capture these different problems while speeding up the query processing.

Fifth, SpatialQA is proposed as a benchmark for LLMs on geospatial question-answering. The use of large language models (LLMs) as question-answering (QA) systems is rising in popularity. A QA system is designed to provide relevant answers given user queries. To address the challenge of evaluating QA systems, standardized benchmarks have been introduced. This work presents a benchmark specifically for answering questions related to geospatial objects. It includes a large set of question-answer pairs using Open Street Maps (OSM) and Wikipedia data, covering various spatial objects and output types. It includes an evaluation methodology that highlights the complexities of geospatial data. LLM baselines are also evaluated to demonstrate the effectiveness of the benchmark.

In the next chapter, we look at the potential of diffusion models as geospatial data generators, which are typically used to generate realistic images from text prompts. Geospatial data generation has many use cases, such as spatial interpolation and change analysis. Existing data generators are based on pre-defined spatial distributions or simulating a specific behavior. We adopt existing diffusion models and provide an initial prototype

for generating geospatial vector data from text prompts. We also carried out experiments on the possibility of using one of the components in this prototype for similarity search using geospatial data embeddings, highlighting the potential of vector databases in this domain. Many challenges still remain for a true general-purpose geospatial data generator.

Finally, we draw some conclusions about the contributions made in this work and provide guidance on future research directions.

Chapter 2

Spatial Parquet: A Column File Format for Geospatial Data Lakes

2.1 Introduction

Recently, there has been a tremendous increase in the amount of publicly available data that are used for data science and data analysis projects. For example, Data.gov [57] contains more than 350,000 dataset that are provided by the US federal government alone. Other governmental and non-governmental open data repositories provide non-precedented volumes of data that grow faster than Moore's Law. These dataset open the door for many interesting data science projects but maintaining all this data is challenging. These dataset are often kept in data warehouses or *data lakes* where users can browse, download, and analyze all this data.

To store any dataset on disk, the two major formats are row-oriented and column-oriented formats. Traditional row-oriented formats, such as CSV and JSON, store the entire record in consecutive disk locations. These formats are usually easier to process and are suitable when the entire record is needed. However, for analytical jobs that need to access a few fields, i.e., columns, it adds unnecessary overhead. Thus, column-oriented formats have been proposed to overcome these limitations. In column formats, the entire column is stored in consecutive bytes on disk which provides two unique advantages over row formats. First, if only a few columns are needed for an analytical job, e.g., calculate average income, we can scan this entire column while not reading the rest of the file from disk. Second, it enables more efficient encoding techniques, such as delta encoding, to store each column in a more efficient way. In summary, column formats are preferred for large scale analytical queries.

One of the most popular column formats is Parquet [208] which is an open-source file format inspired by Google's Dremel [152] system. Parquet is more geared towards big variety data by allowing nested and repeated attributes such as in JSON files. Similar to other column formats, it supports a library of encoding and compression techniques for numeric values to increase its efficiency.

With the increasing amount of geospatial data, Parquet is a very attractive solution that has the potential of saving a significant amount of disk space while increasing the performance of data analysis jobs. However, Parquet is not readily suitable for geospatial data that is more complicated than simple numeric values. In particular, Parquet has three main limitations that limit its use with geospatial data. First, geospatial data is stored

as points, lines, and polygons, which have some internal structure that Parquet does not understand. Second, geospatial data consists mainly of (x, y) coordinates that are stored in floating-point format but Parquet does not provide an efficient encoder for floating-point values. Third, Parquet provides the feature of column statistics that can be used as an index but it does not work for geospatial data. The only solution that is currently available is GeoParquet [78] which partially addresses the first challenge and adds a significant overhead which defies the purpose of using a column store in the first place.

To resolve these issues, this paper presents SpatialParquet, an extension to the Parquet file format that overcomes the limitations of Parquet. First, it proposes a new data type that is compatible with the Parquet file format and can store all common geometry types, i.e., Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon. Second, it adds a novel FP-delta encoder that can significantly reduce the storage requirements for floating-point values that represent geospatial coordinates. Third, it combines the statistics feature of Parquet with the proposed structure to provide a light-weight spatial index that can skip disk pages that do not match a given spatial query range. We run extensive experimental evaluation and found that SpatialParquet outperforms all existing file formats in terms of storage size.

The rest of this paper is organized as follows. Section 2.2 explains how we structure all standard geospatial data in Spatial Parquet. Section 2.3 describes the FP-delta encoding method for floating-point geospatial coordinates. Section 2.4 introduces the light-weight spatial index. Experimental evaluation results are detailed in Section 5.5. The related work is explained in Section 2.6. Finally, Section 4.6 concludes the paper.

2.2 The Structure

This section describes how SpatialParquet stores all the geometry attributes into a unified structure when writing to disk and how it reconstructs them when reading back from disk. There are two main challenges that SpatialParquet has to overcome. First, since Parquet requires all records to have the same schema, we need to create one common schema that can support all the geometry types, i.e., Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon. GeometryCollection requires special handling that we mention at the end. The second challenge is to ensure that this structure keeps the semantic meaning of all the individual parts of the geometries to facilitate efficient storage and retrieval, e.g., the coordinates and sub-parts of some geometries.

To overcome the two challenges above, we propose the following schema to store geometries. We use Google Protocol Buffers Format (PBF) which is the one used by Parquet.

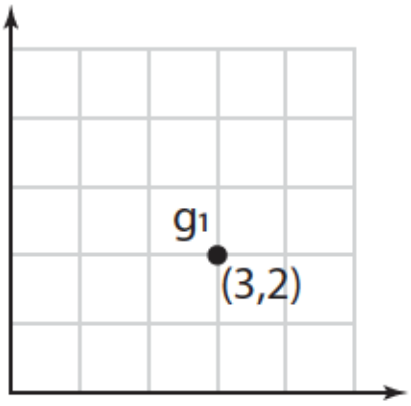
```
message Geometry {
  required int type;
  repeated group part {
    repeated group coordinate {
      required double x;
      required double y;
    }
  }
}
```

Now, let us explain the structure above. The `type` attribute stores a numerical value that represents the geometry type, i.e., 1=Point, 2=LineString, ... etc. We reserve type 0 to represent empty geometries. The outer group, `part`, represents a connected component in the geometry. For example, in a Polygon, the outer shell and each inner hold

is a part. Finally, the `coordinate` group represents a sequence of coordinates that comprise one part. For brevity, this paper assumes two-dimensional coordinates but the structure above can be directly extended to support three dimensions or more by adding their values in the inner-most group. Notice that PBF allows any level of nesting so if the geometry is a part of a feature along with other attributes, the entire definition above will be a single attribute in the feature as shown below where the '...' will be replaced by the definition above.

```
message Feature {
  required int id;
  optional string name;
  // Other non-spatial attributes
  optional group geometry { ... }
}
```

Now, looking at the structure above, we can see that it overcomes the two challenges described earlier. First, this unified structure can support all geometry types as detailed later in this section. Second, this structure contains three columns, `type`, `x`, and `y`, where each one holds a semantic meaning to the geometry and all of them are visible to Parquet to store them efficiently. Additionally, the overhead of maintaining the double nested group, i.e., `part` and `coordinate`, is minimal thanks to the Parquet structure. In this specific case, only four extra bits are needed for the `x` and `y` attributes, two bits for the definition level and tow bits for the repetition level. Interested readers can refer to the Dremel paper [152] for more details about the definition and repetition levels.

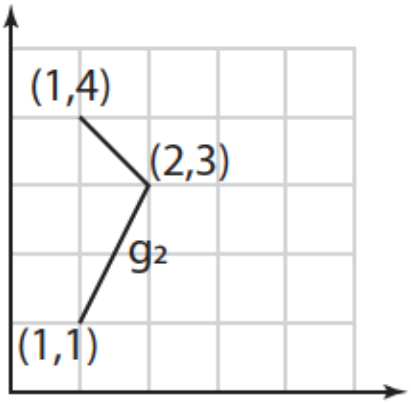


(a) Sample point

Type	Part	
	Coordinate	
	x	y
1	3	2

(b) Column Representation

Figure 2.1: Column representation for a Point



(a) Sample LineString

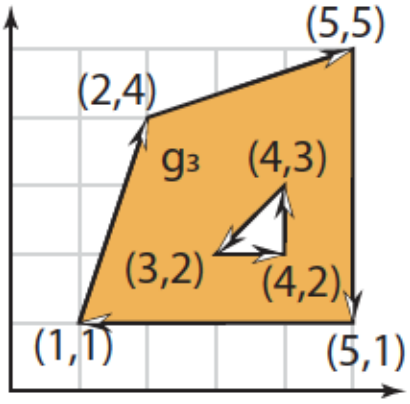
Type	Part	
	Coordinate	
	x	y
2	1	1
	2	3
	1	4

(b) Column Representation

Figure 2.2: Column representation for a LineString

2.2.1 Point (type=1)

A Point contains a single coordinate (x, y) which can be represented as shown in Figure 2.1.



(a) Sample Polygon

Type	Part Coordinate	
	x	y
3	1	1
	2	4
	5	5
	5	1
	1	1
	3	2
	4	2
	4	3

(b) Column Representation

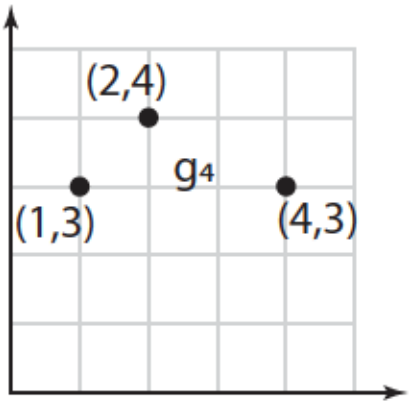
Figure 2.3: Column representation for a Polygon

2.2.2 LineString (type=2)

A LineString is a sequence of coordinates $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \rangle$. Figure 2.2 gives an example of a LineString, g_2, with three points. The points are all represented within one `part` in their respective order.

2.2.3 Polygon (type=3)

A Polygon contains a list of *rings*. Each ring is a LineString that has the same starting and ending points, i.e., $(x_1, y_1) = (x_n, y_n)$. The first ring represents the outer shell while the subsequent rings represent the inner holes. In SpatialParquet, each ring is represented as a `part` similar to how we store a LineString. For consistency, we follow a common convention for storing polygons where the outer shell is stored in clock-wise (CW) order while inner holes are stored in counter clock-wise (CCW) order. Notice that we do not need this information when parsing a polygon since there is only one outer shell, i.e.,



(a) Sample MultiPoint

Type	Part	
	Coordinate	
	x	y
4	1	3
	2	4
	4	3

(b) Column Representation

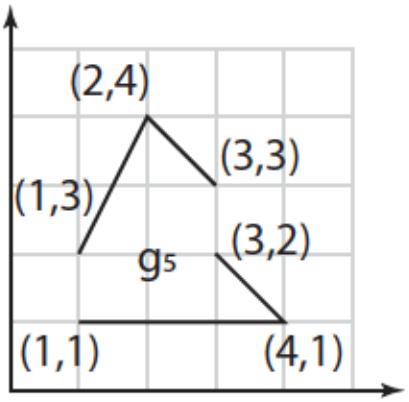
Figure 2.4: Column representation for a MultiPoint

the first ring, and all subsequent rings are holes. However, this information will become useful for MultiPolygons.

Figure 2.3 illustrates an example of a polygon, g_3, with one hole. The outer shell contains four segments that are stored in CW order. Notice that we repeat the last point which is similar to the first point as a common convention in most spatial file formats even though it could be redundant. The inner hole is stored as a second part and the points are ordered in CCW order. In the table representation, we use a horizontal line to represent the end of each part. In Parquet, this is represented by using a repetition level = 1 for the first value in the second ring.

2.2.4 MultiPoint (type=4)

A MultiPoint consists of a sequence of independent point locations. Each point is represented as a single coordinate (x, y). A MultiPoint is represented in SpatialParquet by creating a separate part for each point with a single coordinate inside it. Figure 2.4 shows



(a) Sample MultiLineString

Type	Part	
	Coordinate	
	x	y
5	1	1
	4	1
	3	2
	3	3
	2	4
	1	3

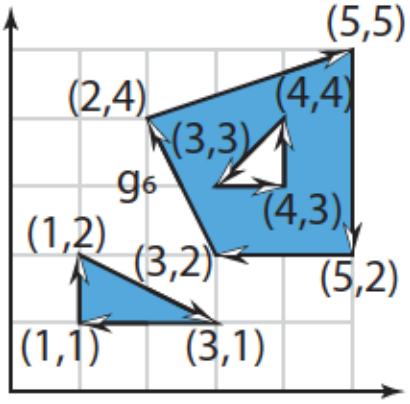
(b) Column Representation

Figure 2.5: Column representation for a MultiLineString

an example of a single MultiPoint, g_4, with three points inside it. Notice how each point is stored as a separate part. We could also store all the points in one part and it will use exactly the same storage size. However, we chose to use a single part for each point as it is semantically more accurate.

2.2.5 MultiLineString (type=5)

A MultiLineString consists of multiple line strings. Each line string is a sequence of coordinates. SpatialParquet stores MultiLineStrings by creating a separate part for each LineString. Each part contains the sequence of coordinates as done with LineString. Figure 2.5 illustrates an example with a MultiLineString, g_5, that contains two LineStrings. Each LineString is represented as a separate part in the column representation.



(a) Sample MultiPolygon

Type	Part Coordinate	
	x	y
6	2	4
	5	5
	5	2
	3	2
	2	4
	3	3
	4	3
	4	4
	3	3
	1	1
	1	2
	3	1
	1	1

(b) Column Representation

Figure 2.6: Column representation for a MultiPolygon

2.2.6 MultiPolygon (type=6)

A MultiPolygon consists of a sequence of polygons. Each polygon contains one outer shell and zero or more inner holes. In SpatialParquet, we represent a MultiPolygon by storing each ring as a separate part, i.e., exactly in the same way as a regular Polygon. To be able to know where each new polygon starts, we follow the convention of storing outer shells in CW order and inner holes in CCW order. Thus, when reading a MultiPolygon back, after reading each ring, we first test whether the coordinates are stored in CW or CCW order which is a linear-time operation. If it is an inner hole, we append it to a list of rings. If it is an outer shell, we first create a polygon from existing rings and then add the new ring as the first one in the next polygon.

Figure 2.6 shows an example of a MultiPolygon, g_6, with two sub-polygons. There is a total of three rings as shown in the column representation. When parsing this geometry back, SpatialParquet will read the first ring and keep it in a buffer. Then, it will read the

second ring and test the order of the points. It will find that they are stored in CCW order so it will keep that ring on the side as a hole. Then, it will read the third ring and find that it is stored in CW order which indicates that it is an outer hole. Thus, it will create the first sub-polygon with one hole. Finally, since no more parts are left in this MultiPolygon, it will create the second sub-polygon with one outer shell and no holes.

2.2.7 GeometryCollection

A GeometryCollection consists of a set of geometries that each can be any of the six geometry types described above as well as another GeometryCollection. Supporting this type is tricky since PBF and Parquet do not allow recursive definition, i.e., a GeometryCollection within another GeometryCollection. We can partially support GeometryCollection by making two changes. First, we change the original definition of Geometry to make the entire Geometry a repeated group. Second, before storing any GeometryCollection we first flatten it by replacing each sub-GeometryCollection with its contents. This way, we remove all recursive definitions in the GeometryCollection and store all sub-geometries at one level. Finally, each geometry in the GeometryCollection is stored as described above.

2.3 The Encoding

One of the main advantages of column-oriented stores, is that it groups together homogeneous values, i.e., from the same domain, in each column and makes use of the redundancy among these values to store them more efficiently. This is done through special encoding schemes that work at their best when encoding homogeneous values. For example,

a very popular encoding is *delta encoding* which stores the deltas (differences) between consecutive values to reduce the storage overhead if the differences are usually small.

This section describes how we encode the column-represented geometries in SpatialParquet to improve the storage efficiency. In SpatialParquet, we primarily have two column types, the geometry `type` column and the `coordinate` columns, which we describe below.

2.3.1 Geometry `type` encoding

Geometry type is an integer value that takes a value in the range $[0, 6]$. In almost all practical cases, all geometries in one dataset have the same type. For example, a point-of-interest dataset will consist of only points. Therefore, we use run-length-encoding (RLE) to encode the geometry type value. RLE replaces consecutive entries with the same value with two numbers, count and value. The former records how many times the latter value is repeated. For example, if all the dataset consists of a single geometry type, e.g., polygons, this column will be stored in SpatialParquet as a pair $(c, 3)$, where c is the total number of records in the file and `type=3` is the marker of the polygon data type. Thus, this method can reduce the storage overhead of the `type` column to virtually a constant that does not depend on data size.

2.3.2 Geometry `coordinate` encoding

The x and y coordinates are stored in floating-point representation[1] A very popular encoding for integer values is delta encoding which stores the first value in a column in full,

[1]We assume 64-bit IEEE double floating-point representation but the discussion seamlessly applies to 32-bit single floating-point representation.

and then for subsequent values it stores only the delta between each value and its previous one. For example, to store the sequence $\langle 15, 16, 15, 17, 20 \rangle$, they are replaced with the sequence $\langle 15, +1, -1, +2, +3 \rangle$. The key idea behind delta encoding is that when the deltas have a smaller magnitude value, they can be represented in fewer bits. Additionally, since all the deltas are stored consecutively in the column format, they are bit-packed to reduce the storage size.

Unfortunately, delta encoding can only be directly applied to *integer* values. In the IEEE floating point data representation, a smaller magnitude value does not necessarily need fewer bits. This is because any floating point value has to be represented in the (sign, exponent, fraction) format. Also the value has to be first normalized to move the *decimal point* right after the most significant one in the number. Check Appendix 2.8 for more details.

When looking at the geometry coordinates, we observe that subsequent values are usually close to each other. For example, a trajectory represented as a MultiPoint is expected to have geographically nearby values. Thus, for both x and y coordinates, each two consecutive values will have a very small difference. However, as mentioned earlier, if we just compute the floating-point difference, we cannot directly reduce the number of significant bits in the number. However, we make another observation that subsequent values are mostly within the same order of magnitude. In other words, they are expected to have either the same, or very close exponents in their floating point representation. Furthermore, if they have the same exponent, then their fractions are also expected to have a small difference.

FP-delta Encoding: Based on the observations above we have above, we proposed a floating-point-delta encoding, FP-delta, that requires only one single operation to calculate. FP-delta simply calculates the difference of the integer interpretation of the floating point values. In other words, we ignore the (sign, exponent, fraction) representation and just treat the entire 64-bit double floating-point value is a 64-bit two's complement long integer value. Of course, the difference in this case does not necessarily hold any physical meaning. However, since the exponents are in the most significant part of the value, and if the exponents are similar, then they will cancel each other. Furthermore, if they cancel each other, the resulting delta will represent the difference between the two fractions. Thus, if the two values have the same exponent and their values are close to each other, the FP-delta value is expected to have only a few significant bits which allows us to reduce the amount of storage. As in integer-based delta encoding, we follow our FP-delta encoding with zigzag encoding which maps the deltas of $\langle 0, 1, -1, 2, -2, \ldots \rangle$ to the positive-only value of $\langle 0, 1, 2, 3, 4, \ldots \rangle$. This encoding simply removes the leading ones that are present in negative values in the two's complement representation.

To summarize, the algorithm works as follows. Given a sequence of floating-point numbers, we scan all the values to calculate how many bits we need for the deltas (further explained later). Then, we start producing the output by writing the first value in full. After that, we scan the subsequent values and compute the delta followed by the zigzag encoding, again. At this point, if the delta can be stored in the determined number of bits, we store it directly. Otherwise, if it needs more bits, we store a special reset marker and store the value in full.

Algorithm 1 FP-delta encoding algorithm*

```
1: function FP-DELTA-ENCODE(double[] X, BitOutputStream out)
2:     n* = COMPUTEBESTDELTABITS(X)
3:     resetMarker = -1 ⋙ (64-n*)
4:     significantOnes = -1 ≪ (n*)
5:     out.write(n*, 8)
6:     out.write(X[0], 64)
7:     for i = 1 to |X| − 1 do
8:         delta = cast-long(X[i])-cast-long(X[i − 1])
9:         zigzag = (delta ≫ 63) ⊕ (delta ≪ 1)
10:        if (zigzag & significantOnes ≠ 0) or (zigzag = resetMarker) then
11:            out.write(resetMarker, n*)
12:            out.write(X[i], 64)
13:        else
14:            out.write(zigzag, n*)
```

* ≫ is the arithmetic shift right, ⋙ is the logical shift right, ≪ is shift left, & is the logical AND operator, and ⊕ is bit-wise XOR

Algorithm 1 provides the pseudo-code of the FP-delta encoding algorithm. The input is an array of floating-point values and the output goes to a bit output stream. The bit output stream bit-packs all the written values depending on how many bits are used for each value. Line 2 uses the function COMPUTEBESTDELTABITS, explained later, to compute the number of bits, n^*, to use for deltas that will minimize the overall output size. After that, it computes the reset marker which is simply the highest possible zigzag-delta, i.e., all bits set to one. It then computes a value with its most significant bits set to one, corresponding to the number of bits to be discarded. Line 6 writes the first value as a full 64-bit value. Then, the loop in Line 7 writes all the remaining values. First, it computes the FP-delta and encodes it using zigzag. Then, it checks if the zigzag-encoded delta has any of its significant bits set to one out of the bits to be removed or if it is equal to the reset marker. If so, we first write the reset marker followed by the full 64-bit value. Otherwise, it writes this value directly to the output stream in n^* bits.

To complete the encoding algorithm, we describe how to compute the number of bits that minimizes the storage size. The challenge in this part is to balance the trade-off between the number of bits needed for deltas and the number of overflow deltas that need to be encoded as 64-bit values. To overcome this challenge, we scan the entire dataset to compute the number of bits needed for each delta. As we do so, we build a histogram h of number of deltas for each number of bits. That is $h[n]$ indicates the number of deltas that need at least n bits to be stored. This means that $\sum h = |X| - 1$ since each value falls into one of the histogram bins. Then, we calculate the output size for each of the deltas and choose the minimum.

To calculate the output size S, recall that each value that can be encoded as a delta will take n bits, where n is the number of bits reserved for the deltas. Any value that cannot be encoded in n bits will be stored as $n+64$ bits, that is, the special marker followed by the full 64-bit value. Thus, for any value n, the output size can be calculated using the following equation.

$$S(n) = n \cdot \sum_{i=0}^{i=n} h[i] + (n + 64) \cdot \sum_{i=n+1}^{i=64} h[i] \tag{2.1}$$

Since $\sum h = |X| - 1$, we can rewrite this equation as follows.

$$S(n) = n \cdot (|X| - 1) + 64 \cdot \sum_{i=n+1}^{i=64} h[i] \tag{2.2}$$

The optimal number of bits for the deltas to minimize the storage size is as follows.

$$n^* = \underset{0 \leq n \leq 64}{\operatorname{argmin}} \{S(n)\} \tag{2.3}$$

There are two notes about this approach. First, since we can accurately compute the encoded data size, we can easily skip this algorithm altogether and store the data in its raw format in case the calculated saving is very little. Second, this algorithm has space and time complexities of $O(|X|)$ since it needs to store and scan all the values before writing them. However, keep in mind that this algorithm is applied for each *page* in the Parquet file which has a default size of 1MB. Thus, the overhead is not significant. Still, if we want to save the overhead, we can skip this function and use an empirical best value that can be calculated experimentally for the application depending on the dataset characteristics. The pseudo-code for this algorithm is available in Appendix 2.9.

FP-delta Decoding: The decoding algorithm is simpler since it does not need to calculate the output size or choose the best number of bits for the delta. The decoder begins by reading the first byte that contains the number of bits n that will be used for the deltas. Then, it reads a 64-bit value that represents the first-floating point value. After that, it keeps reading n bits from the input and checking the value. If it is not the reset marker, it is treated as delta and the next value is emitted accordingly. Otherwise, if it is the reset marker, it is ignored and the next 64-bit double value is read and emitted.

Algorithm 2 gives the pseudo-code of the FP-delta decoding algorithm. It takes an input bit stream and produces all the output values to a DoubleOutput stream. First, it reads an 8-bit value that indicates the number of bits n used for the deltas. Then, it reads

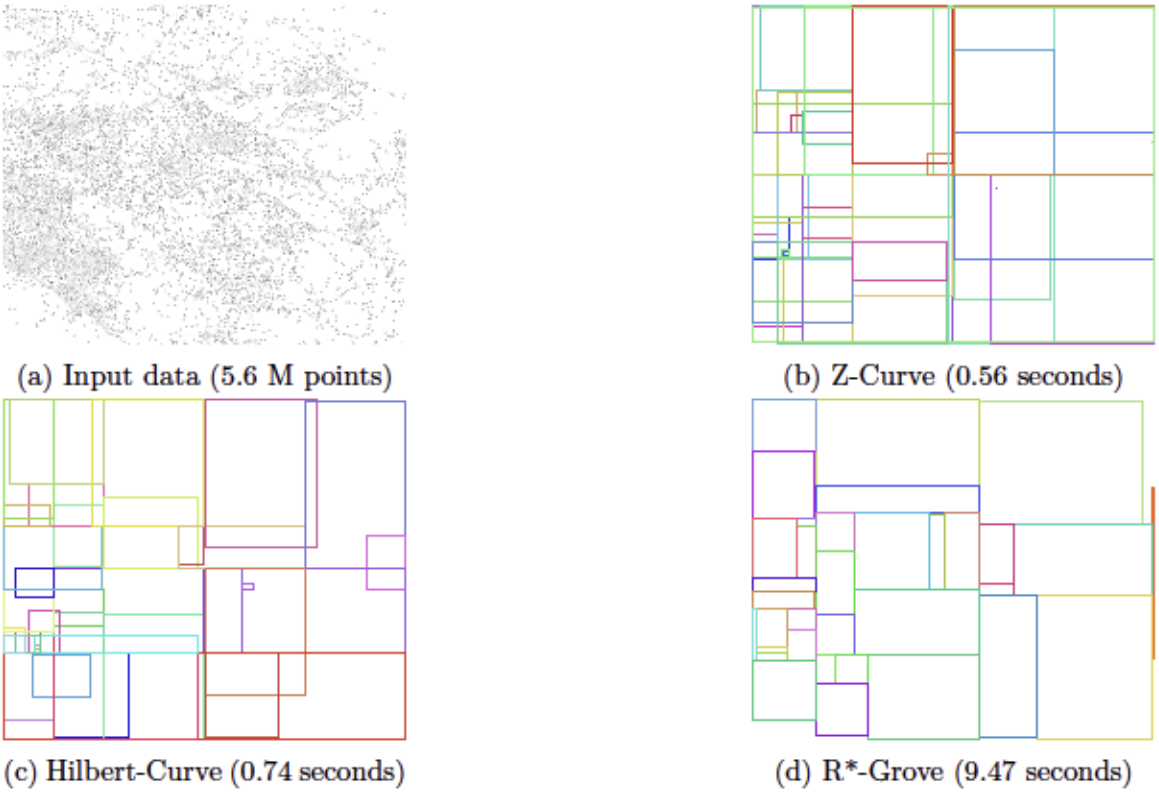
Algorithm 2 FP-delta decoding algorithm

```
1: function FP-DECODE(BitInputStream in, DoubleOutput out)
2:     n = in.read(8)
3:     reset-marker = -1 ≫ (64-n*)
4:     prev-x = cast-double(in.read(64))
5:     out.write(x)
6:     while more values in the input do
7:         zigzag = in.read(n)
8:         if zigzag ≠ reset-marker then
9:             delta = (zigzag ⋙ 1) ⊕ -(zigzag & 1)
10:            next-x = cast-double((cast-long)prev-x + delta)
11:        else
12:            next-x = cast-double(in.read(64))
13:        out.write(next-x)
14:        prev-x = next-x

⋙ is logical shift right, & is bit-wise AND
```

the first value as a 64-bit floating point value and emits it directly to the output. It also keeps it in the variable $prev - x$ to use for delta decoding. After that, it keeps reading from the input bit stream as long as there are value. Notice that Parquet determines if there are more values depending on the definition level so we do not need to keep this information. For each value, it first reads n bits that represent the zigzag-encoded delta. If this value is not equal to the reset marker, it reverses the zigzag encoding to get the delta value as shown. The next value is then calculated by adding this delta to the previous value. Notice that the addition is a 64-bit integer addition and we cast the values accordingly to get the actual double value. Otherwise, if the zigzag value is equal to the reset marker, we simply discard it and use the next 64-bit value as the next x value. Finally, we emit this value and keep it in the buffer so that we can read the next value.

Parquet Integration The only encoder for floating point values in Parquet is the raw-encoder that stores each value as-is, and may apply a run-length-encoding to save space on



(a) Input data (5.6 M points)

(b) Z-Curve (0.56 seconds)

(c) Hilbert-Curve (0.74 seconds)

(d) R*-Grove (9.47 seconds)

Figure 2.7: The effect of sorting on page boundaries with three methods with their respective running time

repeated values. To integrate our FP-delta method into Parquet, we extended the *Encoder* and *Decoder* interfaces in Parquet to support the encoder and decoder functions described above. We configured Parquet to automatically use the FP-delta method for floating-point values in the geometry data type. This allows Parquet to still recognize the floating-point values in the coordinate columns which is helpful for the next part, the indexing.

2.4 The Indexing

This section describes how to build a light-weight index on-top of SpatialParquet using the built-in structure of Parquet. The goal of the index is to be able to avoid reading

the entire file if the user wants to process data in a small geographical region. Most popular spatial indexes, e.g., Quad-tree, R-tree, and GridFile, group records into minimum bounding rectangles (MBR) so that an entire block can be skipped when it falls outside the query range.

Parquet provides a light-weight method of pruning non-relevant records by adding *column statistics*. Mainly, it adds the range, *[min, max]*, for each column and allows the user to skip reading the data if the desired range does not overlap with the column data range. To make this pruning more effective, Parquet splits each column into pages of roughly 1MB each so that it can skip over some of these pages. Furthermore, each page is compressed separately which makes skipping these pages more efficient.

The structure we propose in SpatialParquet gives us the opportunity of collecting statistics for the x and y columns. This is only possible because Parquet identifies each of these as a separate column. In contrast, if we store the entire geometry in WKB or WKT format, Parquet will not be able to collect these statistics. Together, the ranges of x and y make a spatial bounding box for each page. Thus index construction is as simple as instructing Parquet to collect the minimum and maximum for the x and y columns and store them in the output file.

At reading time, if the user provides a query rectangle in the form of $[(x_{min}, y_{min}), (x_{max}, y_{max})]$, we translate it into two separate ranges, $[x_{min}, x_{max}]$ and $[y_{min}, y_{max}]$ for the x and y columns, respectively. Then, we pass these ranges to Parquet to read only the pages that intersect the query range.

The effectiveness of this light-weight index highly depends on the distribution of the data. in one extreme, if data is highly scattered such that each page contains data from all over the data domain, then this technique will be very ineffective since all $[min, max]$ ranges will cover almost the entire input space leaving no room for pruning. On the other extreme, this technique will be very efficient if the data is highly-clustered, that is, spatially nearby records are very close to each other in the input order. This will allow the range of each page to be very tight and more pruning can happen.

To improve the effectiveness of the index, we add a sorting step that tries to cluster nearby records. Notice that it does not have to be perfect. All we need is to avoid the very bad situation where each page covers the entire world. Thus, we do not want to pay a high cost for a very accurate partitioning technique. We decided to use two light-weight space filling curve sort methods, namely, Z-curve and Hilbert curve. Both techniques can provide a linear sort order that takes into account both x and y coordinates, with Hilbert curve known to be more effective with a slightly higher computation cost. Furthermore, since this is not a traditional hierarchical index with a single root, we do not care about sorting the entire dataset which can be very costly. Rather, we process the records into groups with a fixed number of records, e.g., one million. Whenever we have that number of records, we sort them and write them to SpatialParquet. This ensures that the memory overhead is upper bounded and that the computation overhead of sorting grows only linearly with the data size.

Figure 2.7 provides an example of the effect of sorting the records before writing the SpatialParquet file for a 5.6 million point dataset. Without using any sorting, all partitions

cover the entire input space so we do not show it here. While not perfect, both of them provide an opportunity for pruning some disk pages depending on the range query. Notice that SpatialParquet does not care how the partitions are created so if the user is willing to spend more time, more sophisticated big-data partitioning techniques can be used [65]. In the figure, we show an example of using R*-Grove [211] which yields much better partitions with no overlap but, in this case, it takes slightly less than 10 seconds. Given the static nature of the data, users might also be interested in using learned spatial indexes [19] but we leave this as future research directions.

2.5 Experiments

This section shows the results of an extensive experimental evaluation that compares SpatialParquet to existing spatial data formats, as well as, studying the effect of various parameters.

All experiments are executed on a machine running on Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz and 64GB of RAM.

We use four datasets for all evaluations. All the dataset are publicly available on UCR-Star [79] and are summarized in Table 3.4. Each of these datasets contains predominantly one geometry type. This helps us evaluate the effect of different geometry types on compression and storage size. These datasets also have different sizes in terms of the total number of geometries and points contained in them. The versions of these datasets only contain geometry data, and all objects are stripped of any metadata. This so that we compare purely on the geometry values, since this is the main focus of our work. Second,

Table 2.1: Experiment Datasets

Dataset (Acronym)	Geometry Type	# of Geometries	Num points
Porto Taxi (PT)	MultiPoint	1.7 M	83 M
TIGER18/Roads (TR)	MultiLineString	18 M	350 M
MSBuildings (MB)	Polygon	125 M	753 M
eBird (eB)	Point	801 M	801 M

the objects on some of these datasets already follow some sorted order as they are provided from the source. We discuss the effect of sorting in detail.

We implement SpatialParquet in Java based on the original Parquet Java repository [11].

In the remainder of this section, first we compare our proposed work to existing spatial formats. We discuss both savings in terms of required storage size, as well as, performance implications. After that, we show the effect of space-filling-curve-based sorting on the distribution of values in each column chunk. Following, we show the effect of various parameters like encoding, compression, and sorting on the data size and on the performance. Finally, we show how column statistics improves the performance of reading by pruning column chunks and pages based on a provided filtering range.

2.5.1 Comparing to Existing Spatial Formats

We compare Spatial Parquet to three existing baselines: GeoParquet, ShapeFile, and GeoJSON. We use Java implementations for all of these baselines. For GeoParquet, its existing implementation is only available as a Python package, so we provide a new

Java implementation for a fair comparison. Its implementation differs from SpatialParquet in which it requires five values per geometry object. One value represents the Well-Known-Binary (WKB) of the geometry, and the other four values determine the minimum-bounding-rectangle of the geometry for easy filtering. For reading and writing ShapeFile and GeoJSON we use implementations provided in Beast [66].

For evaluations in this section, the source data for writing these files are sorted using the Hilbert-curve method. The details of how the sorting is applied and its effects are provided in the following sub-section.

In the first evaluation, we evaluate the formats based on the total data size without any compression. The left part of Table 2.2 shows the size of data stored in these formats without any compression. For all datasets, SpatialParquet with delta encoding significantly decreases the size of the data. In the case of the PT, MB and eBird datasets, its size is less than half that of the nearest data size for the other formats. We believe that this alone makes SpatialParquet a strong candidate for storing Geospatial data.

Compressing the data using a general purpose compression techniques can further reduce the sizes, but it has some performance implications for both reading and writing. We store the same files in the previous evaluation after compressing their contents using GZIP [58] compression technique. However, there are some differences in how the compression is applied. In SpatialParquet and GeoParquet, the compression is applied on column pages, usually one megabyte in size. This is important to minimize the overhead when reading records from an arbitrary position. Also, compressing small chunks of data adds minimal overhead at write time, but it loses a little bit compared to when the compression

Table 2.2: Output size in GB with/without compression

	Uncompressed				Compressed			
Format	**PT**	**TR**	**MB**	**eB**	**PT**	**TR**	**MB**	**eB**
SpatialParquet	**0.856**	**3.5**	**8.2**	**11**	**0.388**	**1.9**	4.0	1.9
GeoParquet	1.8	6	17	43	0.718	3.5	8.7	6
ShapeFile	1.4	6.4	19	28	0.654	3.5	7.8	5.7
GeoJSON	2.2	14	32	97	0.439	2.2	**3.8**	**1.8**

is applied to the entire dataset at once. For GeoJSON files, the entire dataset is written as one giant `.geojson.gz` file. Thus, the compression technique is applied to it as one large file. ShapeFiles are compressed a little differently. Due to the implementation used for writing Shapefiles, to avoid running out of memory, we divide the data up to one million geometry object and write them into a separate Shapefile partition. So when we apply the compression, we apply it to each of these files individually. The result of this evaluation is shown in the right part of Table 2.2. Clearly, compression adds significant benefits for all formats, with SpatialParquet still considerably lower than all other formats, except in two cases. The compressed GeoJSON file of the MSBuildings dataset is slightly smaller than that of GeoParquet. This can be attributed to the fact that compression is applied differently as described. This shows that compressing an entire dataset using GZIP can save a little bit in storage size compared to compressing small chunks separately. However, keep in mind that a compresses GeoJSON file has to be processed entirely and sequentially while SpatialParquet provides more efficient access methods such as choosing specific column or filtering by rows using the light-weight index (as detailed shortly). Other binary formats are up-to three times bigger than SpatialParquet.

Next, we compare the writing time of SpatialParquet against the baselines. Table 2.3 shows the writing time in seconds for the uncompressed files. SpatialParquet has

the best performance by far for the PT and eB datasets. However, it performs slower than GeoParquet on the TR and MB datasets. These two dataset contain geometries of type MultiLineString and Polygon, respectively. These two data-types are more complex than the Point and MultiPoint types. More complex types require more calls to the parquet interface, since we send each individual value by itself, and it has to track the size of each geometry part. Because Parquet has BYTE ARRAY as a native type the well-known-binary (WKB) is sent directly as one value through the Parquet interface. Keep in mind that our current implementation is a first-cut solution while WKB reading and writing has been optimized for years. Given the huge space saving of SpatialParquet, we will further optimize the writing operation to reduce any potential overhead.

Finally, we compare SpatialParquet to the baselines in terms of the reading time. Table 2.3 shows the reading time in seconds for the uncompressed files. GeoParquet and Shapefile have the best reading times. Similar to writing, reading data in WKB is much more efficient than requesting values repeatedly through the Parquet interface. We believe we can improve the reading performance in the future by providing a lower-level access to the coordinate arrays from Parquet rather than reading one value at a time using the current API. Notice that the information is already stored in the Parquet file in the format that we want, i.e., consecutive arrays of floating points values, so we should be able to further optimize the reading part in the future without changing the SpatialParquet format.

In summary, SpatialParquet provide an excellent alternative for the existing geospatial data formats. It can add significant savings in storage requirements. It does not currently always perform the best in terms of writing and reading speed for all datasets, but

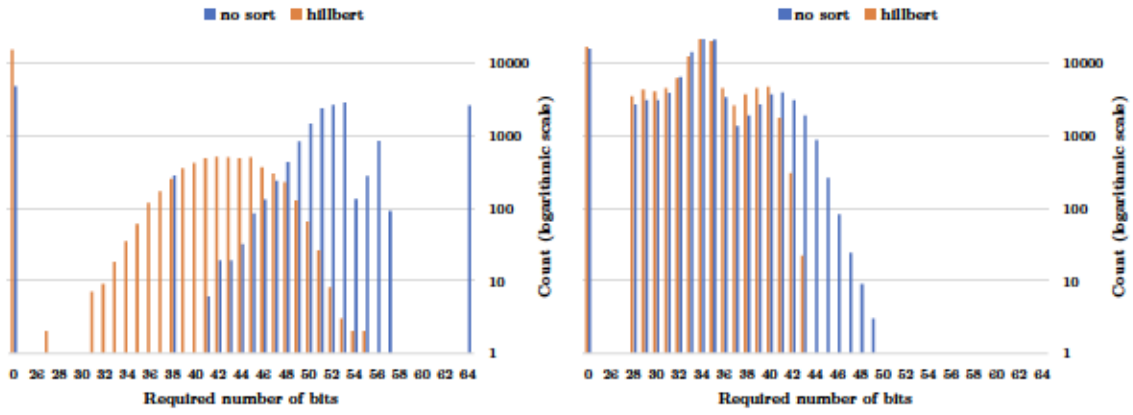
Table 2.3: Write/Read time in seconds for uncompressed formats

	Writing Time				Reading Time			
Format	**PT**	**TR**	**MB**	**eB**	**PT**	**TR**	**MB**	**eB**
SpatialParquet	**74**	215	544	**833**	49	143	455	546
GeoParquet	226	**99**	**425**	1490	**17**	64	204	**500**
ShapeFile	123	490	1445	4246	88	**43**	**161**	534
GeoJSON	105	485	956	2342	55	424	610	1280

future improvements can be added to bridge this gap without changing the proposed format. In addition to adding more improvements to the implementation, we plan to integrate it within Beast [66], which would make it more straightforward to convert from and to SpatialParquet with other geo-spatial formats, on top of making it possible to process big data in distributed systems.

2.5.2 Effect of Sorting on Sample Distribution

In this section, we highlight the effect of sorting on the distribution of the deltas in a column chunk. Sorting results in delta values with less number of significant bits. We show the histogram of the first column chunk of the eBird dataset and the MSBuildings dataset. Sorting has a considerable effect on the eBird dataset because it is not readily sorted from the source. The MSBuildings dataset is somewhat sorted because the data is divided by U.S. state, but applying a sort function still adds more benefit. The other two datsets seem to be well sorted from the source and do not benefit much from sorting. Figure 2.8a shows how a Hilbert-curve sort shifts the distribution of values towards the left, meaning more values requiring less number of significant bits. The count of delta values that require 64 significant bits is large for the unsorted eBird dataset because it includes consecutive values alternating between positive and negative numbers. However, this basic sorting completely



(a) eBird (b) MSBuildings

Figure 2.8: The effect of sorting on the number of records that require at least n bits for delta encoding

removes this spike. We notice the same effect on the MSBuildings dataset in Figure 2.8b, but to a lesser degree since all records are in the US. Applying a Z-curve sort, has more or less of a similar effect on all datasets.

Another thing we notice from these two figures is that a considerable number of values are concentrated at the zero significant bit bar. This bar represents consecutive values that are exactly the same. This is common for geotagged values that are created from the same address. This means that we can add an additional run-length-encoding after the deltas to store all of these consecutive equal delta values in a more compact way. We leave this as a future improvement.

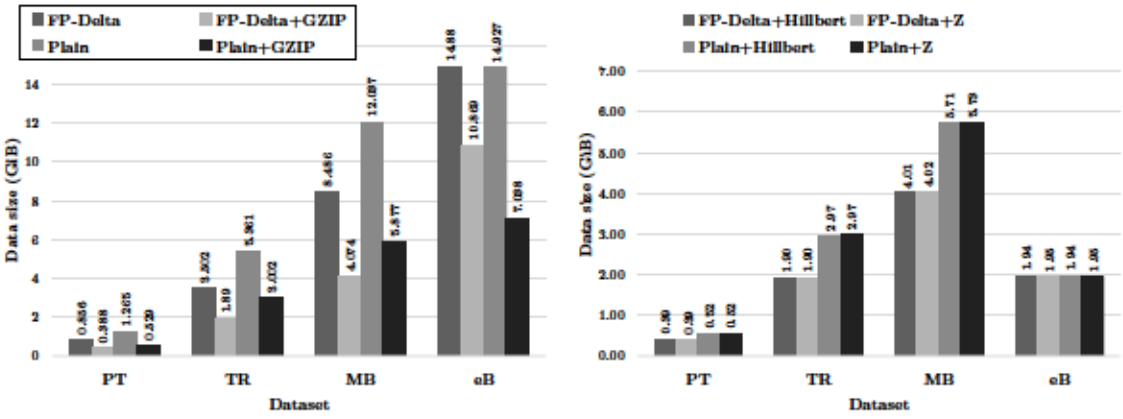
2.5.3 Evaluating Possible Configurations in SpatialParquet

In this part, we delve into SpatialParquet to evaluate the possible configurations for it. First, we look into the effect of using FP-delta with and without compression, before applying any sorting. In Figure 2.9a, in all cases FP-delta results in a smaller size with

and without GZIP compression, except for the eBird dataset. The eBird is not sorted by default and since all of its geometries are points, there is no gain from applying the delta to a single geometry object. Therefore, sorting is required to significantly reduce the size.

We show the sizes of compressed data after sorting in Figure 2.9b. The main difference can be noticed in the eBird dataset because it is the only one that is not originally sorted, although we could have shuffled the other datasets for the purposes of this experiment.

Both FP-delta and sorting add benefits in reducing the final data size, but they add some performance overhead. This evaluation is depicted in Figure 2.10. FP-delta requires an additional iteration over the data, given that it calculates the final bit size after completing all the differences. Also, its current implementation involves allocating an additional buffer to store the final values. In the worst case, it seems that it adds up to 80% of overhead compared to writing the plain double values. Sorting can add a significant overhead, because it is performed sequentially on a buffer of size at most one million objects. However, considering the major benefits it adds this should be negligible. Moreover, in practice we can sort very big data using distributed sorting/indexing techniques. Beast [66] has several of these methods implemented on top of Spark. We plan to integrate Spatial Parquet within Beast, which would make sorting/indexing, among other optimizations, a more seamless process.



(a) No sorting (b) Hilbert- and Z-Curve sorting

Figure 2.9: The effect of sorting on output size in SpatialParquet

2.5.4 Column Statistics and Filtering

Parquet by default collects column statistics for column groups, and chunks. In this experiment, we show the case when no filter is applied, and two additional cases with a small range filter, covering less than 0.01% of the total area covered by the dataset, and a somewhat larger range filter, covering something between 0.33% to 4% depending on the dataset. Figure 2.11 shows these results for reading based on these configurations. Note that this filtering is applied per column group first, and then per column chunk. The figure clearly highlights the benefit of this type of filtering. Note that GeoParquet has similar benefit in terms of pruning parts of columns, but it stores additional columns for the minimum-bounding-rectangle and applies the filters based on them. The current default implementation of Parquet doesn't support filters on repeated columns, however, we made a slight modification to its source code to make it work in our case.

This is only the most basic type of pruning which is provided out-of-the-box. However, there are several opportunities for more advanced indexing. Additional metadata

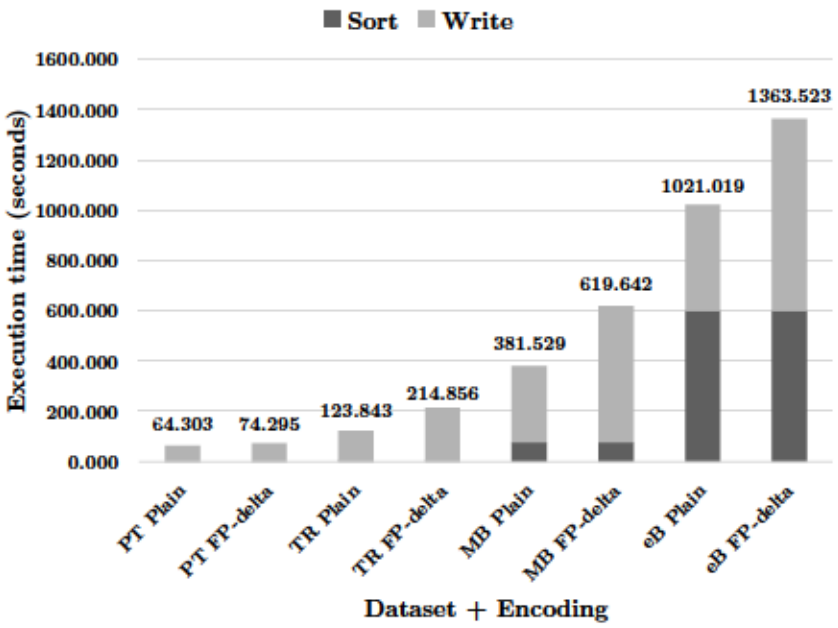


Figure 2.10: Encoding and sorting overhead

can be stored on each column and retrieved when reading prior to unpacking the values. By implementing custom filters specific to spatial indexes, it should be possible to apply more advanced pruning. Additionally, more accurate sorting can be applied by sorting the entire dataset efficiently. The partitioning techniques available in Beast [66] can be used to do so efficiently.

2.6 Related Work

This section covers the related work in two areas, column formats and encoding.

Column Formats Column stores [194] have been proposed for data warehousing and analytical queries due to their efficient storage and retrieval. To support semi-structured big-data with nesting and repetition, Dremel [152] was introduced by Google which then

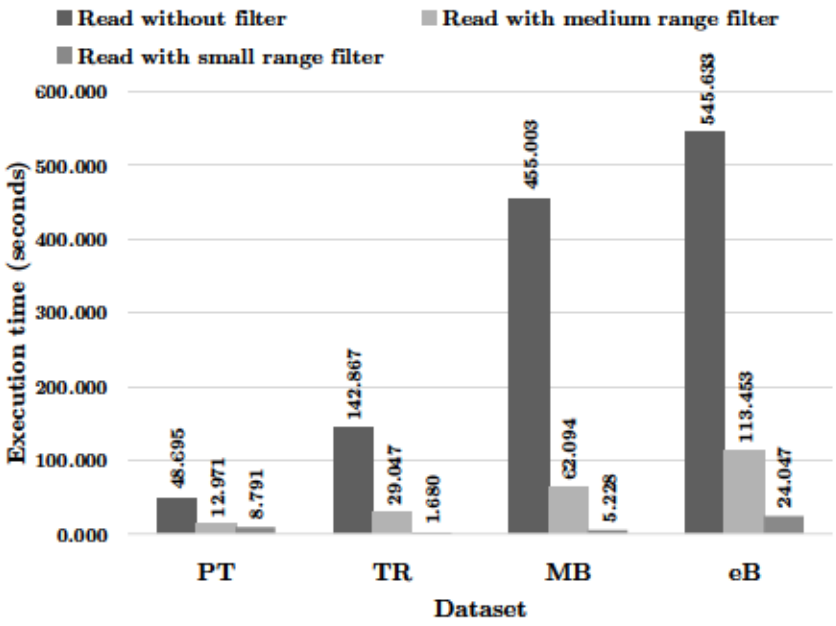


Figure 2.11: The performance of the light-weight spatial index

inspired the open-source Parquet file format [11]. It is widely used across many applications for data storage and data analysis. Several big-data systems adopted Parquet as one of its standard formats such as Apache Spark [229] which uses it in its Spark SQL [3] and MLlib [154]. An experimental evaluation [69] showed the efficiency of Parquet with text data. We plan to integrate SpatialParquet within existing big spatial data systems, e.g., Beast [66], which would make it possible to benefit from the existing functionality in Spark for the Parquet data format. The only existing attempt to provide a column-oriented format for geo-spatial data is GeoParquet [78], also referred to as geo-arrow, which encodes the geometry value in the Well-Known Binary (WKB) format. However, as shown in the experiments, this does not provide a good output size since it can only apply general purpose compression methods.

Encoding Parquet ships with encoding techniques for integer and string values, e.g., delta, run-length, and dictionary encoding[17, 160, 102, 116]. We use RLE for the type column but none of these techniques work with floating-point coordinates. Due to the complexity of encoding floating-point values, some recent work proposed methods that are tailored for specific applications, however, none of these focuses on geographic coordinates. Gorilla [166] targets time series data. It applies XOR between consecutive values and adds post-processing steps to remove leading and trailing zeros. Our preliminary investigation determined that this method does not work well for geographic data and we found that the proposed method is more efficient with geographic data. Similarly, the work in [39] focuses on time series data and improves over Gorilla [166]. Also, [110] focuses on time series data but provides a different approach by encoding similar patterns in time series by mapping them to a dictionary.

Furthermore, other literature focused on the lossless compression of scientific data. The work in [174] uses the integer delta encoding and XOR for encoding the values, as well as, a hash-table for predicting the previous value. Other examples of lossless compression of scientific data include [71, 54, 113]. These compression techniques can be applied after the delta encoding but we leave this work for future research. We do not consider lossy compression techniques [26, 59, 234, 134] in this paper since we propose a general-purpose storage format while lossy techniques can be applied only to some applications.

There are several general optimizations that can be added for column structured data. For example, the work in [186] explores how re-arranging the data can help improve the space saved with run-length-encoding (RLE), which is something that we have discussed

in the experiments. RLE is not specific to any data type. Moreover, [94] provides a SIMD based optimization for better filtering of column data. It claims that this optimization can be faster by 90% than the default filtering in Parquet.

In summary, and to the best of our knowledge, SptialParquet is the only system that proposes a specialized lossless encoding technique for floating-point geographic coordinates.

2.7 Conclusion

This paper introduced SpatialParquet, a column-oriented file format for geospatial data. SpatialParquet is designed to store large-scale spatial data in a column format that reduces disk size and improves the performance of analytical queries. To accomplish its goals, we explained how SpatialParquet introduces a Parquet data type that structures all common geospatial data types, e.g., points, lines, and polygons, in a format that is compatible with Parquet. To make the storage of floating-point coordinate values more efficient, SpatialParquet introduced the FP-delta encoder, which is an efficient encoder that captures the redundancy in geospatial attributes and utilizes it to reduce the storage size. Finally, SpatialParquet used column statistics from Parquet to build a light-weight spatial index that can reduce disk access by skipping file pages that do not overlap with a spatial query range. Experiments on large-scale real data showed that SpatialParquet outperforms all popular spatial file formats when it comes to data analytics.

2.8 Preliminaries

This appendix provides some preliminaries that are needed to understand this paper in case the reader needs a quick memory refresh.

2.8.1 Geometry Data Types

Vector geometry data is represented as points, lines, and polygons. The Open Geospatial Consortium (OGC) defines an industry standard for representing geospatial data. It defines primarily seven data types as detailed shortly. Notice that each of these data types has a variation that stores three-dimensional points (x, y, z) and four dimensional points (x, y, z, m). We focus on two-dimensional points for simplicity but all the proposed techniques in this paper can seamlessly apply for any number of dimensions.

1. **Point** is defined by a single coordinate (x, y).
2. **LineString** is defined as an ordered sequence of coordinates. A special case of LineString is a *Ring* which has the same starting and ending points.
3. **Polygon** is defined as a sequence of Rings. The first Ring defines the outer shell and subsequent rings define holes in the polygon.
4. **MultiPoint** is a set of Points.
5. **MultiLineString** is a set of LineStrings.
6. **MultiPolygon** is a set of Polygons.
7. **GeometryCollection** is a set of geometries that can include nested GeometryCollections.

2.8.2 Parquet

Parquet is a column-oriented file format that is geared for big data storage. It can store any data type that can be defined using Google Protocol Buffers Format (PBF). Basically, PBF can store primitive values, e.g., numbers and strings, arrays, and nested objects. Parquet only stores the primitive values in columns. Nesting and repetition are supported by attaching definition and repetition levels to each column as further detailed in [152]. The only limitation in Parquet is that all values in the file have to follow the same exact PBF schema. Therefore, two records cannot contain mismatching values for the same column, e.g., string and number.

Parquet groups the records into *row groups* which are typically 1GB in size. Within each group, each column is stored separately as a *column chunk*. Each column chunk is further split into *pages*. Finally, each page is encoded and compressed to improve storage efficiency. When reading a file back, the minimum reading unit is a page. By default, each page is about 1MB of size.

Parquet provides the delta encoding and run-length-encoding (RLE) to reduce the storage size for each page. It also provides lossless compression techniques such as GZip and Snappy to reduce the storage size. Finally, Parquet can collect statistics for each page, e.g., minimum and maximum, and can use this information to skip reading pages that do not fall within a user-provided query range. For example, if a user wants to read all entries with the anomalous temperature of $125°F$ or more, Parquet can skip all pages that have degrees below 125.

2.8.3 IEEE Floating Point Format

In computers, floating-point values are represented in the IEEE 754 floating point standard as shown in Figure 2.12. The value is stored in three parts, sign, exponent, and fraction. The value stored in this format can be calculated as:

$$(-1)^{sign}(1.fraction)_2 \times 2^{exponent-1023} \tag{2.4}$$

Sign 1-bit	Exponent 11 bits	Fraction 52 bits

Figure 2.12: The IEEE 754 standard for floating point numbers

The key idea behind this representation is that any binary floating-point number can be represented in the scientific notation as $1.frac \times 2^{exp}$. Therefore, any value is first normalized to make the *decimal point*[2] right after the most-significant one bit. After that, only the fraction and exponent need to be stored.

2.9 Algorithm Pseudo-Codes

Algorithm 3 provides the pseudo-code for the algorithm that computes the number of bits that minimizes the output size. First it initializes the histogram that captures the number of deltas for each number of bits. Notice that the histogram has 65 bins since the number of bits can go from zero to 65. The for loop in Lines 3-, scans all the values and computes the delta as done in the original algorithm. For each value, it computes the

[2]We use the term *decimal point* for convenience even when describing binary numbers.

Algorithm 3 Find the number of bits to minimize the output size

```
1:  function computeBestDeltaBits(double[] X)
2:      h = Array[0..64]                          ▷ The histogram has 65 bins
3:      for i = 1 to |X| − 1 do
4:          delta = cast-long(X[i])-cast-long(X[i − 1])
5:          zigzag = (delta ≫ 63) ⊕ (delta ≪ 1)
6:          n = num-significant-bits(zigzag)
7:          h[n] + +
8:      for n = 63 downto 0 do                    ▷ Compute suffix sum
9:          h[n]+ = h[n + 1]
10:     n* = 0
11:     S_min = 64(|X| − 1)
12:     for n = 1 to 63 do
13:         S = n · (|X| − 1) + 64 · h[n]
14:         if S < S_min then
15:             (n*, S_min) = (n, S)
16:     return n*
```

minimum number of bits required for this delta and increments the corresponding bin in the histogram. After that, the loop in Line 8 calculates the *suffix sum* of the histogram. This is to allow the summation in Equation 2.2 to be calculated in constant time. After that, to find the best value, we simply try all the 65 possible values and choose the one that yields the minimum output size by applying Equation 2.2. Finally, we return the best value n^* that corresponds to the minimum.

Chapter 3

dsJSON: A Distributed SQL JSON Processor

3.1 Introduction

The JavaScript Object Notation (JSON) is an open data-interchange format. It gained its popularity with the rise of web applications, since it is the native representation of data in JavaScript. As a result, it has been adopted in various applications resulting in the need for storing and processing very big JSON data.

Consider the JSON file snippet in Figure 3.1. A user is interested in extracting the objects (enclosed by { and }) in the `products` array, that fall within a specific category (`category == 11`). To avoid expensive serial parsing, which can take hours for terabytes of data, the file should be partitioned and parsed in parallel by different machines, as shown in the figure. However, this is extremely challenging because, for the second and subsequent

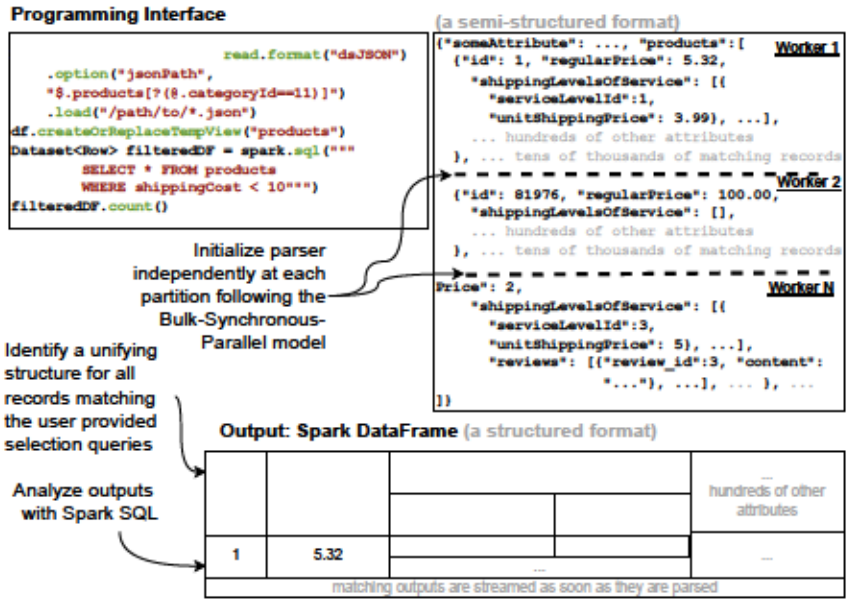


Figure 3.1: An example highlighting dsJSON processing steps

partitions, the parsing state needs to be determined in the middle of the file. Moreover, as the attributes (key-value pairs) of an object might be split among two (or more) partitions, it is non-trivial for the parser to examine the filtering conditions specified by the user. Even worse, since big data frameworks, like Spark [4] and AsterixDB [22], use the bulk-synchronous-parallel (BSP) model, the parallel parsers are not allowed to communicate asynchronously while parsing. To be able to perform different types of analysis readily available like SparkSQL queries, the JSON processor should build a unified schema that matches the user provided queries. In this case, it builds a schema for all products that fall within the specified category. Then, various SQL queries can be easily applied on the extracted records. The example on the top-left highlights a simple query that counts the extracted records with `shippingCost < 10`.

JSON data is produced and consumed in a large variety of use cases. As a popular data interchange format, it is widely used to store data logs, backups, in raw text format, usually encoded in UTF-8 or other suitable text encodings. It is also usually used to store big data dumps that are collected from web APIs, or scraping websites. Furthermore, JSON is used to represent datasets generated by professionals from various fields, like GeoJSON which is a popular format for Geospatial datasets, thanks to its flexibility in metadata storage.

Due to the vast range of use cases, JSON data often contains complex nested structures and can be very large in size, posing several challenges. Unfortunately, existing tools have limitations that can make it difficult to process JSON data in certain situations, leaving users with the option of creating their own ad-hoc programs to extract the necessary data. However, this process can be time-consuming, and even the final program may still have issues due to the complexity of the problem. The following section outlines the three primary limitations of current state-of-the-art JSON processors.

L1. Limited flexibility for parsing complex nested JSON records. Existing tools do not provide full flexibility to the user to determine the most relevant parts that should be parsed and included in the output. Without this flexibility, processing big JSON data can be impractical, due to memory and time constraints. The simplest JSON parsers simply parse the entire input to one large object in memory. Others provide a little more flexibility by taking a simple query from the user and extract values that match it. However, it is still not possible to select multiple attributes from different nesting levels. Hence, the user

is still limited in defining the values to be included in the output. Distributed systems like Apache Spark don't support all use cases due to lower flexibility at the parsing stage.

L2. Limited scalability for data in the general JSON format. Existing distributed systems like [4, 22, 66] cannot parse the general JSON format, instead they are limited to a specific type like the JSONLines [100] format, in which records are separated by the new line character, where each line corresponds to one record in the output. However, many real applications require processing general JSON files like the one shown in Figure 3.1, which can grow in size, making it a huge bottleneck when processed in a single machine environment. Existing parallel parsers of the general JSON format are designed for parallel shared-memory systems or asynchronous message passing interface (MPI) and cannot be used in big-data shared-nothing systems such as Apache Spark and AsterixDB which rely on the BSP model where asynchronous communication is not allowed. This makes it impossible to process very large files where the resources of a single machine are not sufficient.

L3. Limited support for reading JSON data to a structured format to support executing analytical queries. Performing analysis, like executing SQL queries or training a machine-learning model, requires identifying a unified structure of the data. This can be challenging on its own. Systems like Spark perform an inference step to identify this structure by simply computing the union of the object type of each JSONLine in the input data. However, this would fail when there are hundreds of thousands of different attributes in all records which is too big to handle for Spark as further detailed in Section 3.6. Furthermore, existing single-machine parsers that support the general JSON format are not integrated

with a full-featured data analytics system, making it very difficult to perform complex analysis without applying intermediate steps like converting the data into a different format.

Simply converting data to the JSONLines format is not always practical. First, there is no unique way to do this conversion. In the example in Figure 3.1, a user interested in product reviews will only extract the reviews which include the review score and text, while a user interested in price analysis will extract product data including category, price, and description. Second, there is no scalable way to convert general JSON to JSONLine as mentioned in L2. Ensuring data is generated as JSONLines is not always possible when data is collected from various sources. Third, even if data is strictly collected in this format, existing JSONLine processors could fail with complex records since they provide very limited options for schema inference, like identifying a schema for a specific product category, or simplifying complex schema.

Moreover, it is worth noting that the data we are concerned with is stored in raw text format, encoded using UTF-8 or similar, making it even more challenging to process, since the start and end of each record and the number of attributes it contains is not known prior to parsing. This is in contrast to data stored in managed databases like [2, 165] that can efficiently process JSON records by storing them in a binary format and possibly building indexes around them, after a required data insertion step.

To address the above limitations and challenges, this work introduces the first Distributed SQL JSON (dsJSON) processor, fully integrated into an analytics system (Apache Spark). It parses any valid JSON file into a set of records (Dataframe) that can then be

processed using SQL queries or machine learning models. In mitigating L1, dsJSON proposes a data structure called the projection tree, that is used to track the parsing state, and selectively extract the desired attributes, which ensures minimal overhead. The projection tree is built from user provided queries, providing a lot of flexibility for the user to define the parsing stage. To overcome L2, dsJSON provides two partitioning techniques that make it possible to initialize and process the parser at each partition fully independently, following the BSP model. The minimal memory requirement, in addition to the robust partitioning methods, make dsJSON easily scale to very big JSON data. To address L3, dsJSON provides a schema inference method using the proposed projection tree. Building a schema, based on the selectively parsed records, makes it possible to handle very complex records. Schema inference is required for integration with systems that expect (semi-)structured data, like SparkSQL [3] and MLlib [154]. Furthermore, dsJSON pushes SQL projection and filter operations into the parser to improve the performance by skipping unnecessary parts of the input file.

The rest of this paper is organized as follows. Section 3.2 provides some preliminaries about JSON grammar and parsing. Section 3.3 introduces the projection tree and provides an overview of the system. Section 3.4 describes how we build the initial projection tree for the user provided JSONPath queries. Partitioning and schema inference are discussed in Sections 3.5 and 3.6, respectively. Section 3.7 discusses SQL projection and filter push down into the projection tree. Section 3.8 gives the details of record parsing. Section 3.9 provides an extensive experimental evaluation of the system. Section 3.10 describes the related work. Finally, Section 3.11 concludes the paper.

3.2 Preliminaries

The standard JSON grammar has two main components: objects and arrays. An object can have multiple key-value pairs. The order of the pairs has no significance, and this can add to the complexity of parsing the records into a structured, uniform format. The other major component is arrays. The elements of arrays can be any valid JSON value. A value in JSON can be any nested JSON record, or a primitive value. Primitives can be strings, numbers, or one of the reserved words: `true`, `false`, and `null`. The latest version of the standard JSON grammar can be accessed at [99].

There are many variants of the JSON syntax. One very popular variant is the JSONLine format, in which each line in the file contains a separate JSON object. While this format imposes more restrictions over the general JSON format, it is adopted in some systems, e.g., Apache Spark [4] and AsterixDB [22], due to the simplicity of parsing it in parallel. Simply, the file is partitioned into lines and each line is parsed as one unit. Another well-known variant of the JSON format is GeoJSON. This format follows the conventions of the standard JSON format, and the records can exist in multiple lines. It follows a pre-detrmined structure, but may contain metadata that is different from one object to another.

A JSON document can be tokenized into several parts, defined next, and these tokens can be used to track the position within the document while parsing.

Definition 1 (JSON token) *is either one of the control characters, '{' , '}' , '[' , and ']' , a key, which is a string that comes before a colon, or a primitive value, e.g. a number or a string.*

While parsing a JSON document similar to how it is done in [96], a stack is used to track the position. Encountering one of the open control characters { or [or a key results in a push operation. Encountering one of the closing characters } or] or reaching the end of a value after a key results in a pop operation, and it must correspond to the top of the stack.

A simple way to query a JSON structure is by using a JSONPath query [80], which is similar to the XPath [55] queries for XML data. Similar to XML, a valid JSON file is represented as one tree with a single root. This makes it unsuitable for SQL processing which requires a set of records, unless we are processing small JSON files, where each file represents one record.

Some JSON data can get very complex, with records containing hundreds of key-value pairs, and deeply nested objects. Furthermore, some queries can further complicate the parsing with the addition of filtering and descendant elements. To mitigate this, JPStream [96] introduces the concept of streaming-automaton that is built from a user provided JSONPath, and uses it to track the parsing state.

3.3 Projection Tree

This section gives an overview of the projection tree, which is a data structure that dsJSON introduces to selectively parse JSON data and convert the extracted semi-structured JSON records to a structured data format. This is a vital step to run any SQL query on the JSON data and what distinguishes dsJSON from regular JSON parsers that just produce a JSON object. In relational algebra, projection is the operator that can add

or remove columns in relations. The name of the projection tree comes from the fact that it selects a subset of values from different nesting levels from complex JSON documents, and converts them from a semi-structured textual format to a structured one that can be processed using relational algebra operators. Before formally defining the projection tree, first we provide definitions for its basic building blocks.

Definition 2 (JSON path) *A set of JSON tokens that determine the position of a value within a JSON structure. These tokens contain open object '{', open array '[', or a key, i.e., attribute name.*

For example, in Figure 3.1, both attributes `id` and `regularPrice` exist under the JSON path: ⟨ `'{', 'products', '[', '{'` ⟩. Knowing the path leading to a value allows us to determine whether its value is needed, leading us to the *essential path.*

Definition 3 (Essential path) *A path in a JSON document that leads to an attribute required for parsing a desired record.*

In dsJSON, essential paths are determined from user provided queries, i.e., JSONPath and SQL queries. When parsing, a JSON path that does not correspond to an essential path triggers a skipping mechanism to discard values descending from it, until reaching the prefix of an essential path. To track all of these transitions and more, we unify them under the concept of the projection tree.

Definition 4 (Projection tree) *A tree comprised of all essential paths that are used to selectively parse a JSON document and apply complex nested record projections. Each node in the tree corresponds to a token within an input JSON document, where leaf nodes correspond to primitive attribute values, e.g., numbers or strings. Transitions in the tree occur*

as JSON tokens are read from the input, and can trigger several operations like filtering, skipping, or projecting the selectively parsed values to a structured format.

The design of the projection tree makes it more efficient to overcome the limitations we are addressing. dsJSON goes through a few stages prior to producing the final structured output, and all stages relate to the projection tree. Figure 3.2 highlights the five stages of projection tree processing in dsJSON, namely, initial tree construction, input partitioning, schema inference, SQL push down, and row parsing, further detailed below.

The first stage in dsJSON is building an initial projection tree by processing the user-provided JSONPath queries into tree structures, and merging them. Figure 3.3 shows an example of a projection tree. The first part of the tree (nodes #0-#10) is built after processing the user provided JSONPath queries. This initial tree can only support sequential JSON parsing but cannot produce structured records. This process is detailed in Section 3.4.

To be able to process the input data in parallel, the next stage in dsJSON is partitioning. This stage uses the projection tree to determine how to shift the starting position at each partition, such that the starting JSON path at an arbitrary starting position is known, and to avoid splitting records among different processors. We provide two partitioning methods, which are specifically required for data in the general JSON format. In this stage, one node in the tree is identified as the *split node*, e.g., node #3 in Figure 3.3. Section 3.5 provides the details about this stage.

Next, schema inference is performed on the initial projection tree. This stage produces the expanded projection tree, adding nodes #11-#24 in the tree in Figure 3.3. This makes it possible to identify a structured format for the final output. Performing

schema inference through the projection tree resolves some of the limitations in existing systems that cannot produce structured records. The details about schema inference are provided in Section 3.6.

The fourth stage is SQL push down that further optimizes the projection tree by pushing down SQL projection and filters. Based on the types of queries the user executes on the structured data, SparkSQL determines the required attributes and pushes down some of the filters, especially those non-aggregate in nature. In Figure 3.3, SQL push-down results in pruning nodes #10, #16, #18-#24, and adding the last predicate to the filter associated with node #4. In a lot of cases, SQL push-down may result in pruning or skipping hundreds of attributes which can provide significant performance gains. This step produces an optimized projection tree, which is what is used in the final stage of actually parsing the input records. This part is detailed in Section 3.7.

Finally, after obtaining the optimized version of the projection tree, the row parser starts producing the final output records in the expected schema. The transitions that occur in the projection tree while parsing JSON documents are detailed in Section 3.8.

3.4 JSONPath Query Processor

The first problem we face when parsing JSON files is how to define the output records. Keep in mind that a well-formed JSON file is just one big object but it also contains many nested objects at various levels. There is really no right or wrong definition of which records to produce since it all depends on the user application. Notice that in the JSONLine format, each line in the input corresponds to one record in the output, while for

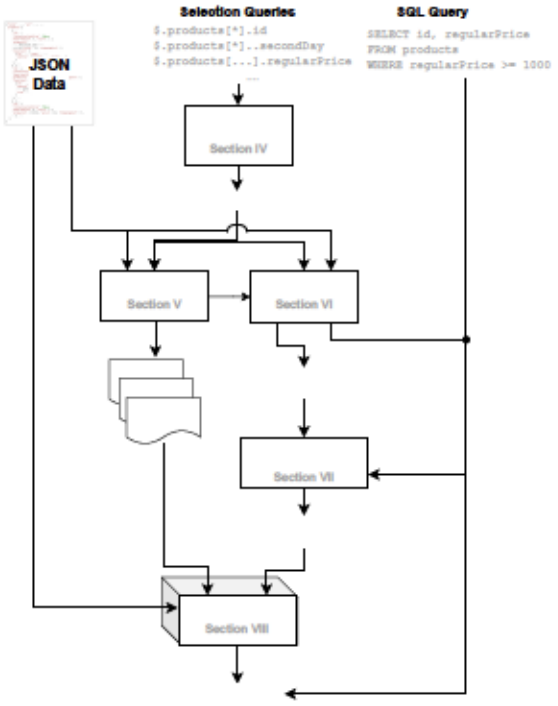


Figure 3.2: System Architecture of dsJSON

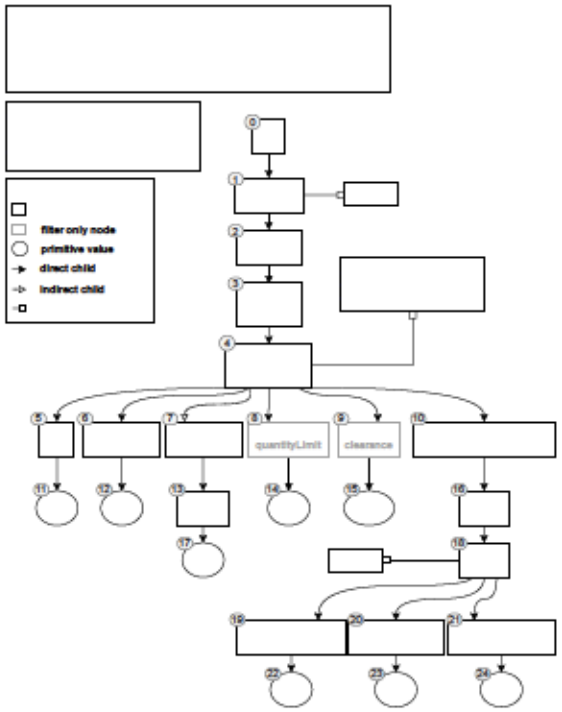


Figure 3.3: Complete Projection Tree Example

the GeoJSON format each record is defined by a geometric feature. However, even for these formats it can be more practical to parse complex records more selectively. To resolve the issue of defining the selected attributes while giving the user the full flexibility, we adopt the syntax of the JSONPath query as a method of defining which records to return.

In the remainder of this section, we first describe how a single JSONPath query is processed to construct a projection tree. Then, we show how multiple projection trees produced from different JSONPath queries are merged into a single projection tree. After that, we describe how JSONPath filters for each object are combined.

3.4.1 Processing a single JSONPath query

This part explains how we construct an initial projection tree from a single JSONPath query. A JSONPath query indicates to the parser that all values descending from this path must be included in the output. The first step in processing a JSONPath query is tokenizing it into its components, and identifying the type of each one. The first token in a query based on the JSONPath syntax always starts with the root symbol `$`. Following tokens can then be one of three types: *key-token*, *descendent-key-token*, and *array-token*.

The *key-token* type corresponds to a key in the corresponding depth in the JSON file, and is preceded by a dot, e.g., `'.products'`. The second token type is *descendent-key-token* written as `..` and followed by a key, which indicates that the depth of the JSON path for matching the next token should not be dependent on the corresponding level on the query, e.g., `'..secondDay'`. The third type is *array-token*, written as a pair of square brackets, e.g., `'[*]'`.

An array-token can can either take the wild card symbol `*` for accepting all elements in the array, or can take a filter, e.g., `'[?(@.categoryId==11)]'`. The original JSONPath query syntax can take the indexes of arrays as filters. However, we omit indexing since it is impractical to know the exact index of each object when the file is partitioned and parsed in parallel. This is because communication is not allowed among Spark executors, following the BSP model. For most use cases, filters and projections are sufficient for selecting the relevant records and the order of records in an array is not important. Furthermore, if a fixed number of records is desired, the parsing can stop after producing that desired number. Additionally, in most cases, the order is still preserved knowing the

partition id, and we also provide the option to store an additional field that can be used to access the rows based on their order.

The original JSONPath filters are only supported for arrays, but for additional flexibility, we allow filters to be provided after any key in the query as long as the value is a nested JSON object, and it can be written between a pair of round parentheses instead of square brackets to differentiate between the different token types. For example, this makes it possible to add a filter on the `address` object, like '`$.products[*]..address(?(@.city == Seatle))`'.

After tokenizing a selection query and extracting any filters, those tokens are then used to build an initial projection tree. We show four examples in Figure 3.4. Notice that each token in a provided query has one corresponding node in the produced tree. Additionally, prior to any node corresponding to a `key-token` we add a preceding object node. Keys in a JSON document always exist within a JSON object. Also, object nodes in a projection tree are used to apply filters and transitioning down to and up from an object node corresponds to reading an { and } , respectively. The projection tree transitions are described in more detail in Section 3.8.

When building a projection tree, we identify a special node that determines when a record is fully processed and sent to the final output. We refer to this node as the projector node.

Definition 5 (Projector node) *A node that indicates the level at which the record is complete and ready to return. It is always the first node from the root where the tree diverges into multiple paths.*

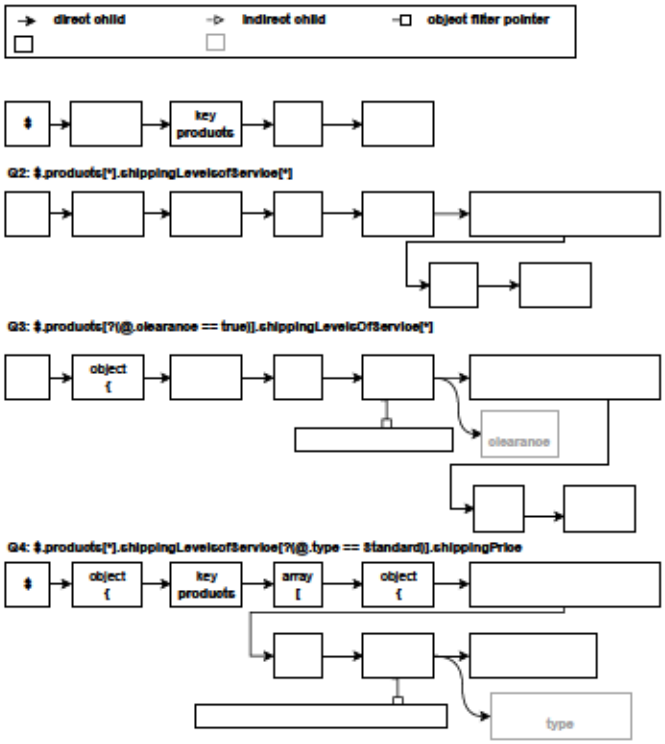


Figure 3.4: Example projection trees from single JSONPaths

In a tree built from a single JSONPath, the projector node is always the last node corresponding to the last token in the query.

3.4.2 Merging multiple JSONPath queries

While one JSONPath query could be enough to properly select the desired set of records, there are many cases where the user might need to specify two or more JSONPath queries. For example, in Figure 3.1, the nested object contains hundreds of attributes but the user might want to select only '`serviceLevelId`' and '`unitShippingPrice`'. If we use a single JSONPath query, then all the sub-attributes will be selected which incur a huge overhead on all the stages of JSON parsing. Therefore, the design of dsJSON allows users to provide many JSONPath queries, which unlocks the potential for complex use cases.

The main part in merging multiple JSONPath queries is to find the most common path among them, ignoring any filters. The queries in Figure 3.3 all share the JSON path ⟨ `'{', 'products', '[', '{'` ⟩. This corresponds to the first five nodes in their individual initial projection trees. Merging the queries results in producing an initial projection tree containing nodes #0-#10 shown in Figure 3.3. The last node that is common among all the queries (node #5) is defined as the projector node, since it is the first node when the initial trees start to diverge into multiple paths.

3.4.3 Defining object filters

The filter expressions, from the JSONPath queries, can be used to improve the performance of JSON parsing by early skipping records that do not match without fully parsing them. To integrate these filters into the projection tree, each filter is converted to an internal tree structure, e.g., Figure 3.5, and attached to corresponding nodes in the tree. The leaf nodes in the tree correspond to JSON attributes (variables) or constants. Each internal node represents a predicate, e.g., $\geq$, or a Boolean operation, e.g., AND (&&). Additionally, we add a hashmap that maps each variable to its leaf node in the tree to quickly locate it. As soon as a variable is parsed from the input, we plug its value in the tree and propagate the evaluation up to the root. A predicate is evaluated when all its operands are available. For example, in Figure 3.5, since the root node is an AND operator, as soon as one of its children evaluates to `false`, the remainder of the record is skipped. In the case of the `AND` operator, it propagates if one of the operands evaluates to `false` or all the operands evaluate to `true`, while for the `OR` operator it propagates if one is `true` or all are `false`. By default, when a filter does not fully evaluate because a value is missing

is considered `true`, the user can change this behavior by including a predicate that checks that these required values exist (`?(... && @.someAttribute != null)`). When merging multiple JSONPaths, the filters at corresponding nodes are also merged. While we build the initial projection tree using multiple JSONPath queries, we opted for the option to support only one, possibly compound, filter per object. By default, if an object has filters coming from multiple paths, they are merged using an `AND` operator. While it can be trivial to add support for multiple filters, and only discard values descending from the path of a filter that evaluates to `false`, it makes the behavior harder to follow from the perspective of the user. The user can control logical operators by simply writing the filter corresponding to a specific node in one JSONPath query, instead of splitting among multiple queries.

In addition to the simple filters on primitive attributes, dsJSON is extensible and can supports user-defined complex filters for different data types. In the experiments, we use filters for the Geometry data type when parsing GeoJSON files.

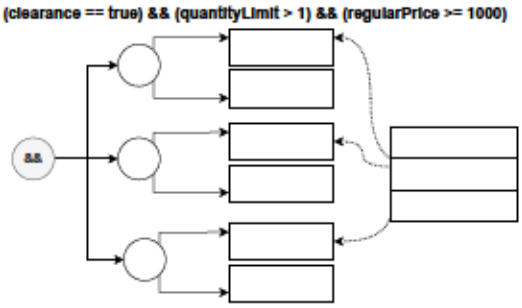


Figure 3.5: Example filter expression tree and variables map

3.5 Partitioning

Similar to most big-data systems, Spark uses the bulk synchronous parallel (BSP) model in which asynchronous communication is not allowed between executors. This means that each partition in a JSON file should be parsed independently. However, the projection tree used to track the parsing will not be able to parse the second and subsequent partitions without being initialized at the correct node. This is very challenging since the initial path at an arbitrary position generally depends on the entire file before. The JSONLines format does not require this part, since the start and end of each partition is shifted to the next newline character, hence, the entire JSON object is always contained in only one partition.

To make sure that partitioning does not split output records and projection filters among two processors, the projection tree has a special node referred to as the *split node*.

Definition 6 (Split node) *It is the deepest node in the tree prior to the projector node and any object with a filter. It determines the initial JSON path allowed at the start of a partition. The start of a partition must be shifted such that its start is not descending from the path corresponding to this node to avoid splitting output records and filters between two partitions.*

The *split node* in Figure 3.3 is node #3. When initializing the projection tree at an arbitrary token the partitioner must shift to a token at a path that is not descending from the split node. To illustrate this concept, Figure 3.6 shows how the positions of characters suitable for splitting change with different queries. The projection tree of each of these queries are those in Figure 3.4. Q1 extracts all objects in the products array. Each of these objects will represent one row in the final Dataframe (table) output. The split node, being

at the array-node just prior to the projector node, makes any control character { } [ ] , or white space characters that are not within the matched objects suitable for splitting. Q2 selects inner objects, and this allows for more split points, as long as the matched inner objects are not split. Q3 is similar to Q2 except that it adds a filter to objects in the products array. This filter causes the split node to be the same as that of Q1, to ensure that filters are not split among partitions, and similarly for Q4.

This design choice has considerable practical benefits, but it imposes two minor limitations. First, it cannot efficiently partition a document when filters are added at the root node or with attributes that exist only at the start or end of a very large document. These types of filters might be useful to prune JSON documents out of a very large set. This will cause no issues for small files since they do not need to be partitioned, but for large documents, it hinders the level of parallelism that can be achieved. In practice, this is not a big issue since existing JSONPath processors only support applying filters to JSON arrays of type object, because typically users are interested in filtering large arrays. Furthermore, for this type of analysis, each matched row is not expected to be big. Second, it limits partitioning opportunities when matching documents that are extremely large. However, most database stores impose a limitation on the size of a row, usually in the order of megabytes, and this is still too small to cause any serious partitioning issues.

To overcome this challenge, dsJSON provides two methods for partition initialization, *speculative* and *full pass*. The speculative method collects information from the start of the file and then speculates based on this information at each partition. The second method uses a full pass on the file that tracks the path at the end of each partition. Then, a merge

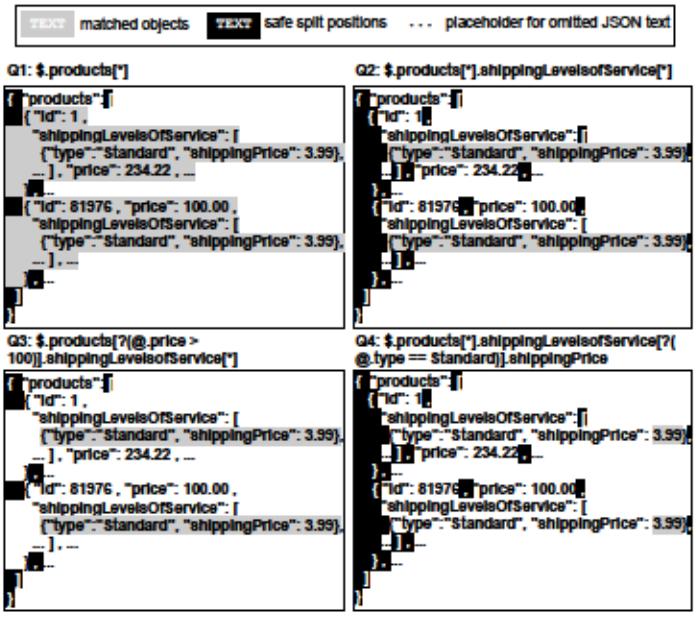


Figure 3.6: Changes in split positions and matched records

step determines exactly the start state of each partition. The full pass method is helpful in the uncommon use cases where speculation might fail. At the end of the partitioning method, the initial path for each partition is determined and is used to initialize the state in the row parsing stage.

```
{"total": ..., "url":..., ..., "products":[
  {..., "id": 1, "productId":
  ---- Partition line ----
  1435108,"locations": [
      {"id": ..., "name": ...,
       "address": {"street": ..., "city": ...} }
      ], ...},
  {..., "id": 2, "productId": 2536543,"locations":
  [ {"id": ..., "name": ...., "address": {"street": ...,
  ---- Partition line ---
  "city": ...} } ], ... }, ... ]}
```

Figure 3.7: Partitioned JSON Data Example

3.5.1 Efficient with speculation

The key idea of the speculative partitioning method is to collect some information from the beginning of the file and use this information to accurately speculate the projection tree at each partition. This method assumes that the entire file will mostly follow the structure of the first part and will use that to resolve any confusion. For example, if it knows from the first part of the file that the key '`address`' appears only while the parser is at a specific projection tree node, the parser will use this information to adjust the projection tree node whenever that key is encountered.

This method collects the information needed for speculative partitioning only from a small portion at the start of the input file. We empirically found that in the majority of use cases, the first megabyte of the file is sufficient for accurate state identification. It starts from the beginning of the input file and collects all the encountered keys along with the corresponding path encountered prior to reaching the key. The key-collector also has a counter for how many times a key is encountered at a given path. The counter helps select the best keys for speculation when partitioning. To illustrate this, the key collector will generate the map in Table 3.1 from the JSON data in Figure 3.7. Some rows are omitted from space.

Table 3.1: Collected keys map

Keys	Path	Count
total, products	{	1
id, locations	{ products [{	1000
id, name, address	{ products [{ locations [{	5234
city	{ products [{ locations [{ address {	5234

This table is used to resolve the ambiguity of the starting node of the projection tree without having to start parsing from the beginning of the file. In most cases, the format of the JSON file is consistent throughout, which makes the parsing state similar when one of the keys in the table is encountered. Furthermore, most keys in the table have only one entry which makes it possible to identify the initial node without any ambiguity. It is also possible to speculate on a key that has multiple entries as long as they result in the same projection tree node and exist at the same depth in the JSON structure and only differ in the value of the keys. However, there are cases where the same key could appear at multiple paths. For example, the key `id` in the table appears at two different paths. Moreover, the counter keeps track of the frequency of each occurrence so that the state initializer will rely more on the more frequent occurrences that has one entry in the table. The most frequent keys with only one entry, in this example, are `name`, `address`, and `city`, followed by `locations`. All of these make good choices for speculation. In the real dataset that this is based on, there are hundreds of such keys. If one of these keys is encountered, it is used to determine the state. In the rare case, where the matched key exists at a different path than the one on the table, then the wrong node will be identified.

Based on this concept, the speculative partitioning method continues after building the table in two steps. First, it defines an initial data-oblivious partitioning that simply partitions the file into fixed-size partitions, e.g., 128 MB each. Second, it applies a data-aware adjustment step that shifts the partition boundaries to the next identified key from the table. Notice that the adjustment step happens in parallel since each partition boundary is shifted independently, which suits the BSP model. After that, the position is shifted to

match a node at or prior to the split node to ensure that a matching record is not split. Speculation can also be improved by searching for multiple keys that are correlated, even if some of them exist in multiple paths. However, since we found that practically the simpler version works well we leave this as a future improvement.

In the example in Table 3.1, if the partition initially starts at the `name` field, which is at a path that is descending from the path corresponding to the split node, then it must search for the following characters },], and }, respectively, to arrive at the split node. This ensures that matching records are not split.

With invalid speculation, the parser either will encounter unmatched closing characters when traversing the projection tree, which would raise an exception, or the parser will continue unaware of the error. However, if the parser continues, it is unlikely that it will match any records, or it will match records that look very different from the inferred schema. To ensure that no errors happen, we add a post-processing check that verifies that the initial path of each partition matches the termination path of the previous partition. If they are equal for all consecutive partitions, then speculation is guaranteed to be correct. The program raises an exception if speculation is not verified with information about at which partition it occurred and the token that was used. Users will then fallback to the full-pass method which is described next.

3.5.2 Exact with a full file pass

While speculation works most of the time, there are cases where it could fail. It is when the schema of the first N records is not enough to generalize to all records in the file. This can happen when a dataset contains recursive objects. Consider the `products` array

example shown earlier, if records within it also contain an attribute called `relatedProducts` that also contains all the attributes within the parent product, it would not be possible to speculate on this dataset since for any attribute within the `products` array there are at least two possible parsing states, assuming the `relatedProducts` is also repeated and the nesting can go multiple levels deep. In this case, the only option is to do the partitioning using the full-pass method. The user can also try to normalize the structure of the data through appropriate JSONPath queries by only selecting the `productId` attribute of all related products, for example.

One reasonable use case that we were able to identify is partitioning GeoJSON files that contain nested GeometryCollections. That is, it contains objects of type GeometryCollection that also contain GeometryCollection objects within them, and the nesting can go multiple levels deep. This way, all the keys in the file will exist in multiple depths. Using speculation on this file, will raise an exception, since filtering the encountered-keys table will result only in the keys that exist once at the top of the file. Hence, a full pass partitioner is required for these use cases that rarely occur in practice. Previous work considers multiple paths that are possible and start parsing assuming one of them, and restarting when a path is identified as false. However, the way the system is designed based on the Spark Data Source API, records are returned as soon as they are parsed. While we can delay returning those values until we eliminate all the alternative possibilities, which might result in multiple full passes over a partition, we opted to have one full pass that guarantees the correct state is initialized, and ensure no significant memory overhead is required to store intermediate values. The data is not fully parsed during this iteration, and it requires

minimal memory and communication overhead since only a small array is gathered from all the partitions at the end of this stage.

The key idea of the full-pass partitioner is to scan all partitions in parallel and record the path that leads to each key *within this partition*. Except for the first partition, the second and subsequent partitions will record invalid paths since they are unaware of the *initial path* at the beginning of this partition. For example, in Figure 3.7, the second partition starts at the path ⟨ '{', 'products', '[', '{', 'productId' ⟩. To resolve this issue, the full-pass partitioner keeps track of two additional pieces of information for each partition. 1) The *inner path*, which is the path that leads to the end of the partition that is not closed yet as shown in Table 3.2. 2) Any close object '}' or close array ']' tokens that are not matched with a corresponding open token. This information is collected at a central node that corrects the paths within each partition. The idea is to propagate the inner paths, representing open token with no closing token, with non-matching close token in the same order of partitions to determine the initial path for each partition. Once the initial path is determined, the correct path leading to each token can be easily calculated by prepending the initial path to the one computed within the partition. Finally, the start and end position of each partition are shifted according to the stored positions of the latest open symbol. Using this method also makes it possible to skip an entire partition in later processing if it starts at a redundant path that does not end by the end of the partition. The start positions are also shifted to match the split node similar to the previous method.

The main edge case when applying the full-pass partitioning function is when the start of a partition falls within a string. This is easily handled knowing the JSON syntax.

Simply, we start parsing as if the partition does not start in a string. If parsing fails due to incorrect JSON format, we restart assuming that the partition starts within a string. Interested readers can refer to [95] for more details.

Table 3.2: Full-pass partitioning complete output

Part	Inner Path	Initial Path
0	{ products [{ productId	EMPTY
1	} { locations [{ address {	{ products [{ productId
2	} }] }] }	{ products [{ locations [{ address {

3.6 Schema Inference

This step takes as input the initial projection tree and the input data. It produces a unified schema for all records that will be added to the final output. The schema is defined as a list of pairs (name, type), each indicating an attribute with the name that appears in the JSON file and its data type, e.g., string, numeric or a nested structure. This schema is important for SparkSQL which needs an initial schema to be able to parse and analyze the SQL queries that the users provide. For example, it needs to verify that there is a numeric column named `regularPrice` for the SQL query in Figure 3.3. This section describes two methods for schema inference, optimistic and pessimistic. Then, it follows with a discussion on how to mitigate a JSON file with an extremely complex schema.

Table 3.3: Inferred schema after expanding projection tree

id	regularPrice	secondDay	shippingLevelsOfService		
			array[object]		
long	double	array[double]	unitShippingPrice	secondLevelId	serviceLevelName
			double	long	string

The optimistic method produces the schema from a fixed size at the start of the input data, which we limit to 1000 records by default, but it is left as an option to the user. This method is much more efficient since it uses a very small portion of the data under the realistic assumption that all other records will follow the same schema. The likelihood that the matched records follow the exact same schema increases based on how selective the JSONPath filters provided by the user. For example, given a filter that selects products from a specific category, then, the variability of the schema would be minimal, hence, only a small number of records will be needed to build a unified schema. The pessimistic method runs a full pass over the file to ensure the result is correct without making assumptions.

Optimistic schema inference works by first getting the initial projection tree in order to find matching records. It starts parsing from the beginning of the file, whenever a record is matched with the JSONPath query, it stores all attribute names and their inferred data types. Figure 3.3 illustrates how schema inference works. In a sense, dsJSON expands the initial projection tree (nodes #0-#10) when parsing a record, producing all other nodes in the tree. After it completes a record, the row projector node returns the row schema of the matched record, like the one shown in Table 3.3. Then, as more records are matched, the schema of all matched records are merged into one while updating data types to be more broad. Most type conflicts are when all previously obtained records have a key of undefined type, while a new record might have a different type. Some attribute types might also be resolved from long type to a double type, as new records are matched. We limit the inference to be based on at most the first 1000 records by default. We found that this default number produces accurate schema while still being reasonably efficient, but it can be

easily changed by the user. This method is executed simultaneously with the partitioning stage if the user chose the speculation method.

For the *pessimistic schema inference*, first it is required that the data is partitioned and the projection tree at each partition is initialized accurately, so that this stage is processed in parallel. Pessimistic schema inference is required for data where some keys may only exist in some records, and when the entire correct schema is required. It works similar to the optimistic schema inference described above, but it uses the entire data. After each partition is processed independently similar to the optimistic approach to produce a schema. Then, all schemata are merged to produce the final complete schema.

Extremely complex schema: In some use cases, merging the schema of matched records may result in a very large nested schema that causes failures due to requiring a very large memory footprint. This is because some fields may not have uniform nested structures across the data. In existing systems, like in the Apache Spark JSON reader, failures can only be mitigated when the user manually provides the schema and setting those complex nested objects as string columns, or discarding them completely. However, the data in these columns would still be in their raw semi-structured JSON format, and will not benefit readily from the existing tools like SparkSQL queries without additional processing iterations that might require implementing some custom functionality. dsJSON resolves this schema expansion issue in three ways, filters, multiple JSONPath queries, and lazy parsing.

First, because dsJSON supports applying filters when inferring the schema, providing appropriate filters results in selecting objects that are more uniform in structure, minimizing the issue of schema expansion. Consider the OpenStreetMap dataset [232] that

we use in the experiments, without using filters, the merged schema grows very large resulting in an estimated size of 1MB per record, which is much larger than the size of a record individually, eventually resulting in execution failure due to running out of memory. This is because it has a large variety of objects which have different attributes in their metadata. Adding appropriate filters, like limiting the selected types and geometric boundaries, reduces the estimated size per record to 400 bytes, which is the expected size for 20 attributes of type string, thus, avoiding the failure. This is not to mention that this dataset cannot be processed using the existing Spark JSON reader, because it is not in the JSONLines format, and the entire dataset is one large JSON object that exceeds two terabytes in size. While Beast [66] supports reading GeoJSON files, its current version fails while partitioning this dataset, moreover, it doesn't support applying analytical queries like those provided by Spark SQL, because it doesn't have schema inference. Therefore, this only leaves dsJSON as the only solution that supports this type of complex large scale data processing.

Secondly, because dsJSON supports extracting values using multiple JSONPath queries, this allows designing path queries that avoid sections in the JSON data that causes schema expansion. The Wikipedia dataset [9] that is also used in the experiments causes schema expansion failure when the schema of the entire records are merged. The way this dataset is designed includes id names as keys within objects, and the size of one record can be hundreds of kilobytes. When merging the schema of different objects, the schema size keeps increasing until the program fails. dsJSON avoids this issue by using multiple JSONPath queries including descending paths, which results in building a compact schema that covers only the relevant part of the data for the desired output.

In this case, the user can adjust their JSONPath queries in a way such that that the final matched records are more unified, especially with the use of descendent attributes. There is also the option to apply the JSONPath filters while building the schema. An appropriate filter may lead to selecting records that mostly follow the same schema, thus avoiding that issue. Without these options, it would not be possible to find a schema of a reasonable size that matches all the records. For example, because none of these options is available in the JSONLines reader in Apache Spark, it fails prematurely while inferring the schema of some files, as shown later in the experiments.

For the lazy parsing method, dsJSON provides the option to automatically detect fields that may result in schema expansion by setting a threshold for the number of allowed subfields, and treat those detected fields as strings, where they can still be processed separately in later stages and they will need to be parsed then. This comes as the last option for avoiding schema expansion without any loss of data. The user also has the option to provide their own schema which would save the schema inference time, assuming the provided schema is valid for the matched records.

3.7 Projection Tree Optimizer

This section shows how the projection tree is finalized by pushingSQL filters and projections down into the tree. While we can simply let SparkSQL operators apply the projection and filter operations, this additional optimization step can greatly speed up the parsing by skipping parts of the input that are not needed. For example, if the filter does

not pass, the rest of the record can be skipped. Also, if only a few attributes are needed, the parsing can stop once these attributes are found.

Note that SQL queries can only be applied to the fields in the inferred schema of the records matching the JSONPath queries, since Spark SQL [3] requires analyzing the schema before pushing down filters and projections, and the schema is based only on the output of the matched records.

SQL projection push-down optimizes the projection tree by pruning paths that lead to non-required attributes. SparkSQL pushes-down projections as a list of fields in the schema that are required for the applied analysis. Based on this list, different changes on the projection tree are applied. If a node does not exist on the list of required attributes and it is not used in a filter, then it is pruned out of the tree along with its descendent nodes. If a node does not exist on the list but it is used in a filter, it is changed to a filter-only node. Nodes that exist on the list are kept as-is. In the example provided in Figure 3.3, this operation results in pruning nodes #6, #7 and #10 and all of their descending nodes. In practice, as further confirmed in the experiments, projection push-down can result in pruning hundreds of nodes, which provides significant performance gains.

SQL filter push-down is integrated by converting the provided filter to an internal expression and appending it to the filter associated with the *projector node*, e.g., node #4 in Figure 3.3. It also results in updating the expression tree of the associated filter, like the one in Figure 3.5. Note that the existing JSON reader in Spark also implements projection and filter push-down which confirms the effectiveness of this step. However, dsJSON makes two fundamental differences. First, Spark uses the Jackson parser [6] as a black-box and adds

the projection and filter support as a post-processing step while dsJSON integrates them into the row parsing. Second, dsJSON seamlessly combines JSONPath and SQL projections and filters into one data structure, i.e., the projection tree, which makes it more powerful than Spark that only works with SQL filters.

3.8 Row Parser

In this final stage, each worker in a distributed cluster takes one partition produced by the partitioner, i.e., start and end offsets in the file, as well as, the initial JSON path. It initializes the optimized projection tree accordingly. The first partition always starts at an empty path and is initialized to the root node. Other partitions are initialized as discussed in Section 3.5 so that the active node matches the initial path. Then, it iterates over the input, token-by-token, while processing them using the projection tree. When the projection tree transitions into the projector node, it starts parsing a new row. This section first describes the type of operations that occur when processing a projection tree, and when they are triggered. Then, we describe the error handling mechanism.

3.8.1 Projection Tree Operations

Several operations can occur as the parser iterates over the input.

Transition downward. It occurs when the parser reads a token that matches the token of one of the children of the current active node. Object nodes are matched with an open curly-brace ({), array nodes are matched with an open square-bracket ([), key

nodes are matched with an equivalent key of the node, while primitive values are matched with a primitive value of the same type.

Transition upward. This occurs when the parser completes reading a value corresponding to a node. For objects, it occurs after reading the closing curly-brace (}). For arrays, it is after reading the closing square-bracket (]). For key nodes, it occurs when the value associated with the key is fully processed, and results in adding it to its parent object. An upward transition from any value results in adding it to its parent node or appending it to its parent array. Additionally, an upward transition from the projector node triggers returning the parsed record to the output, and resets filters.

Propagate to filter. A value is propagated to a filter after an upward transition from a key node, if the key is one of the variables in the filter expression tree for its parent object. If the filter evaluates, to `true` or is still not fully evaluated, processing continues normally. Conversely, if the filter evaluates to `false` the remainder of the object is skipped, and any already processed values descending from this object are discarded.

Skip redundant paths. This operation is triggered when there is no matching essential path for a downward transition, resulting in skipping a value without serializing it. The active node remains the same if its value is not yet fully processed. Skipping is also triggered at object nodes when all of the required values are obtained, or a filter is evaluated to `false`. It searches for the matching closing character keeping track of all control character in between.

These summarize the main operations that occur when processing a JSON document using a projection tree.

3.8.2 Error Handling

The flexibility of JSON means that unexpected input tokens are bound to be encountered, especially when dealing with very large scale data coming from different sources. Most existing JSON parsers fail immediately when encountering unexpected tokens or malformed inputs. However, we designed dsJSON to work in a more permissive mode, making it a lot more robust to failure. Next, we discuss how some unexpected tokens are handled.

Handling mismatched types or invalid primitives. This occurs when the schema expects a specific type while the input contains a different one. In this case, it is easy to avoid any errors. First, dsJSON attempts to convert the type to the expected type. For example, if the expected type is a string and the encountered type is numeric, this number value is stored as a string. If the value cannot be converted to the expected type, it is simply stored as `null`. This possibility of data loss is avoided when using pessimistic schema inference, since it would ensure selecting the broadest datatype, and in case of mismatched types, it is by default set to a string.

Handling malformed JSON structures. Failures from some syntax errors can be easily avoided. For example, missing comma characters between array values, or two key-value pairs within an object, can be easily ignored. More problematic syntax errors are those related to control characters, the curly-braces, the square-brackets, and the double quote character. If an input file has a missing chunk at the end, then that can easily be ignored, and the parts processed prior to it are considered valid. However, we consider encountering non-matching closing tokens a fatal error, which happens when the last token in the current path corresponding to the active node in the projection node does not match

it. This is because ignoring such a syntax error would put the execution in speculative mode, similar to what happens with invalid speculation. If it is important that malformed structures are detected, the user can use full-pass partitioning which also can be used to verify the structural integrity of JSON documents.

3.9 Experiments

In this section, we execute an extensive set of experiments to provide an experimental evidence of the scalability and efficiency of dsJSON over existing JSON parsers. We also evaluate the effectiveness of the components that comprise dsJSON. Notice that since dsJSON supports some use cases that are not supported by any of the existing parsers, e.g., parsing general JSON files, there is sometimes no suitable baseline to compare to. We also provide a more detailed example to showcase one example enabled by dsJSON.

3.9.1 Experimental Setup

Table 3.4: Experiments datasets and queries

Dataset	Short name	Format	Size	JSONPath
Bestbuy [5]	BB	Std. JSON	1GB	`$.products[*]`
IMDB [37]	IMDB	Std. JSON	7.2GB	`[*]`
Wikipedia [9]	Wiki	JSONLine	1.3TB	`[*].claims..mainsnak`
MSBuildings [156]	MSB	GeoJSON	28.5GB	`$.features[*]`
OpenStreetMap [232]	OSM	GeoJSON	2TB	`$.features[*]`

The experiments are executed on a cluster with one head node and twelve worker nodes. The head node is running on two Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz, with 8 cores per chip and 2 sockets per core for a total of 32 processing units. Each worker node is running on Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz with two sockets, for a

total of 12 processing units per node, resulting in a total of 144 processing units among all worker nodes. The head node is running on 128GB of RAM, while each data node is running on 64GB. The data is stored using Hadoop Distributed File System (HDFS) running on the same worker nodes. Also, Table 3.4 shows the datasets used along with the JSONPath queries. We use the same version of the BB dataset from [96] but we scale it to 16GB for all experiments, unless otherwise indicated.

We use the end-to-end running time as a performance metric which includes any required reformatting, parsing, and processing time. During experiments, we vary the input size and the SQL query that we run on each dataset. We compare to four sequential parsers and three distributed parsers. The sequential parsers are JPStream [96], SIMDJSON [115], Jayway [7], and Python JSONDecoder [1]. The distributed parsers are the built-in Spark JSONLine reader, a hand-crafted RDD-based Spark+Jayway reader, and Beast [66] which supports GeoJSON files. We added the hand-crafted parser to enable JSONPath processing for JSONLine files, a feature that is not supported by the built-in Spark JSON reader.

Table 3.5: Comparing to other distributed systems

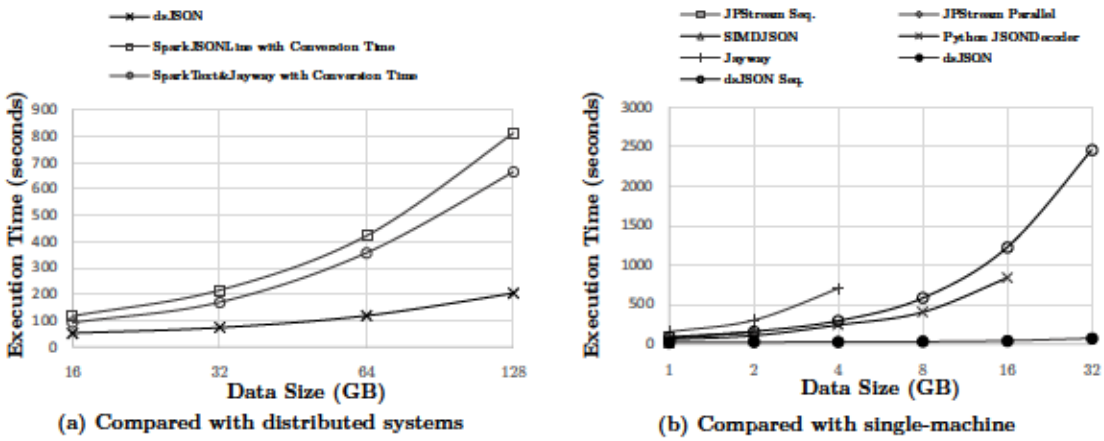
System	**Qualitative Comparison**					**Total execution time (seconds)**				
	Std. JSON	**JSON Line**	**Geo JSON**	**JSON Path**	**SQL**	**BB 16GB**	**IMDB 7.2GB**	**MSB 29GB**	**Wiki 1.3TB**	**OSM 2TB**
dsJSON	✓	✓	✓	✓	✓	52	32	86	1275	2712
Spark JSON Reader		✓			✓	117 (70+47)	91 (60+31)	N/A	Fails	N/A
Spark Text Reader + Jayway		✓		✓		92 (70+22)	89 (60+29)	N/A	1222	N/A
BEAST			✓			N/A	N/A	42	N/A	Fails

3.9.2 Scalability of Distributed Parsing

Due to the variety of baselines and the different features supported by each, we show a qualitative comparison between these on the left half of Table 3.5. These missing features in other baselines limit the quantitative experiments that we can run. In this part of the experiments, when processing a standard JSON file, we run an efficient reformatting step that converts it to a JSONLine format that can be processed by Spark JSON and Spark+Jayway readers. We report the conversion time separately for convenience. The conversion scripts we used are custom to each dataset and only perform string operations without the need for full serialization which makes them highly efficient.

The right half of Table 3.5 reports the total execution time, where N/A indicates a non-supported case. When conversion is needed, the breakdown is provided between parentheses (conversion + processing). Firstly, only dsJSON can successfully process all the datasets and is consistently scalable. Spark JSON reader lags behind for both BB and IMDB datasets due to the bottleneck in file conversion. Interestingly, even after conversion, dsJSON is very close to the parsing step, which indicates the low overhead of schema inference and partitioning. This slight difference is mostly due to implementation details like the way the input reader is used and intermediate data type conversion. However, given that this is a first cut implementation of dsJSON, the results are promising for moving forward to potentially replace the existing Spark JSON reader, after some development iterations.

For GeoJSON files (MSB and OSM), dsJSON can still parse them efficiently. For MSB, Beast was almost twice as fast for three reasons. First, it runs on the low-level RDD



(a) Compared with distributed systems

(b) Compared with single-machine

Figure 3.8: Processing time with increasing input size

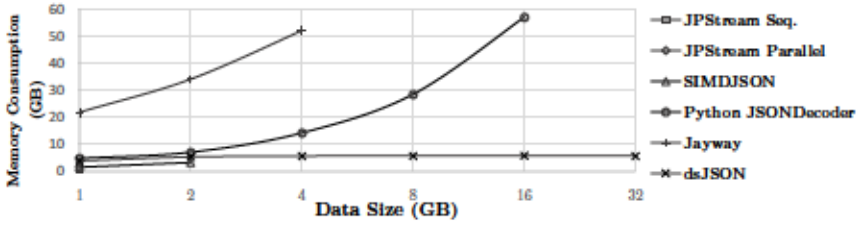


Figure 3.9: Memory usage compared to single machine parsers

API, hence, does not require a schema inference step. Second, the code is hand-crafted to support only GeoJSON files while dsJSON is more general. Third, Beast can directly parse geometry objects into a compact in-memory representation while dsJSON keeps it as a nested JSON object with a complex schema. Even with that, Beast failed to parse the 2TB OSM file due to parsing errors. In the future, we can add a custom parser to GeoJSON files that can represent geometry objects in a compact way to reduce the schema size and increase efficiency.

One of the main motivating factors for introducing dsJSON is to skip the conversion step from the standard JSON format to the JSONLine format for very large datasets. Figure 3.8 shows the scalability of dsJSON when parsing standard JSON files with a `SELECT`

* query. In this experiment, we scale the BB dataset from 16GB up to 128GB, and compare the time for converting and processing using existing systems against only using dsJSON.

Figure 3.8(a) compares dsJSON to the distributed parsers, Spark JSON reader and Spark+Jayway RDD parser,including the required time to convert to JSONLine. Since the conversion step requires a full pass over the file, dsJSON is up-to 400% faster. Note that the conversion step is IO-bound so a more efficient converter will not help much while dsJSON completely eliminates this step.

In Figure 3.8(b), we compare to four single-machine parsers which can parse and process a JSONPath query in a single scan over the data. JPStream [96] (both sequential and parallel) and SIMDJSON are implemented in C++, Jayway [7] is in Java, and the Python JSONDecoder is readily available in Python. All of them run on the head node. We use `$.products[*].categoryPath[*]` as the JSONPath query which selects records with only two attributes. For SIMDJSON, we use its interface to extract the same values. This query reduces the complexity and size of the final output, since some of these single-machine parsers are not optimized for complex output. All of these parsers produce an error as we increase the data size, mainly due to memory requirement. dsJSON scales well to large data sizes even in the single machine serial execution case.

We also show the memory consumption in Figure 3.9. The execution time and memory consumption are measured using the `time` tool in Linux. When measuring the memory consumption, we executed dsJSON in a single machine to make it easier to capture its total consumption. Also, note that memory allocation in Java, and Spark specifically, works differently. It allocates a large memory chunk for all Spark components as configured

in the cluster. Therefore, this experiment focuses on how the memory grows as the input size increases. For the 1 GB file, JPStream executes the fastest and with the least memory consumption. However, its current implementation fails for larger sizes, due to implementation issues. We attempted to modify its implementation to make it scale to larger sizes, but it caused memory leaks and we conducted this finding to its authors. Regardless, its design requires storing the entire extracted data for filtering and finalizing the output. SIMDJSON only works on data less than 4GB, although it can be easily scaled on JSONLines, especially for queries that don't require aggregation of results. It consumes memory at a factor of 1.6 of the input size. The other two parsers are considerably slower and use considerably more memory. Except on the 1GB file, dsJSON executes the fastest, and its memory consumption scales well as data increases. The most important part is that dsJSON can process arbitrarily large files without requiring extra memory. To confirm that, we estimated the memory used by all the data structures in dsJSON using the `SizeEstimator` Spark library, and they only consume a few hundred kilobytes per process. Finally, when executing dsJSON in a single thread, we observed that SIMDJSON is about 60% faster, highlighting a major potential benefit of integrating them.

Finally, Table 3.6 shows the breakdown of all the stages of dsJSON. These results are for executing a `SELECT *` query with no filtering and with *speculative partitioning* and *optimistic schema inference*. The main takeaway is the low overhead of schema inference and partitioning steps even for the very large datasets, e.g., 31 seconds in the optimistic case for the 2TB one. Verification does not add considerable overhead and it passes for all datasets and queries.

Table 3.6: dsJSON breakdown of processing stages in seconds

Dataset	Schema	Partitioning	Parsing	Verification	Other	Total
BB	2	4	27	2.5	12.5	48
IMDB	1	4	14	0.5	12.5	32
MSB	2	4	60	1	12	79
Wiki	1	16	1538	5	20	1580
OSM	1	41	2938	10	15	3005

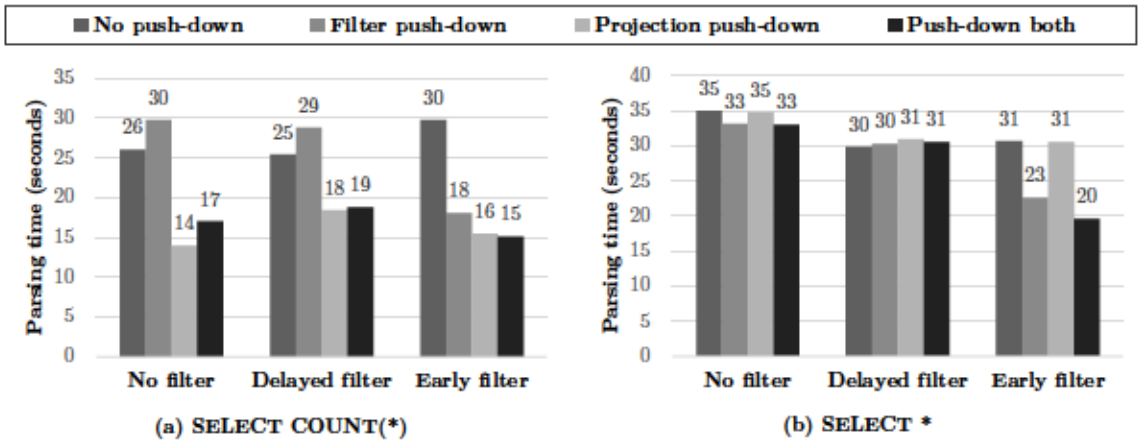


Figure 3.10: Effect of integrating SQL on dsJSON

3.9.3 SQL Integration

Following, we study the effect of pushing down filtering and projection on the parsing time. To study the effect of projection, we study two extreme SELECT clauses, `SELECT COUNT(*)` and `SELECT *` which project an empty set and a full set of attributes, respectively. Additionally, to study the effect of filtering, we study three `WHERE` clauses, no-filter, delayed filter, and early filter. *No-filter* does not apply any filters at all. *Delayed filter* applies a condition that select about 1% of the data and works on attributes that appear towards the end of each record in the file. *Early filter* applies a condition that also select 1% of the data but works on attributes that appear towards the beginning of each record. With early filter, we expect the parser to be more efficient since it can skip parsing the rest of the record once the condition fails.

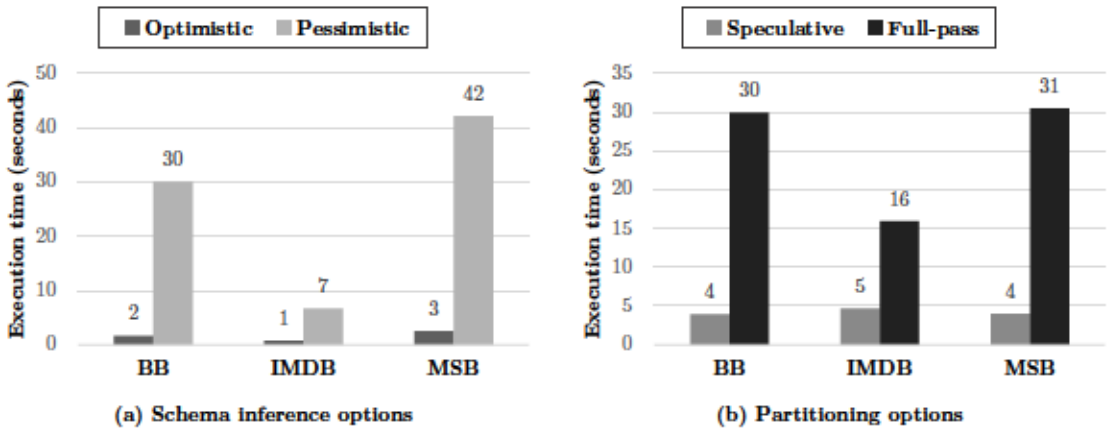


Figure 3.11: Comparing partitioning and schema inference

The results are shown in Figure 3.10. The times shown in this figure are the average of five executions, to reduce the effect of variability, especially due to scheduling, and fetching files from HDFS. These results support our claims in the paper. With a `COUNT(*)` query, projection push-down can save up-to 50% of the execution time. Similarly, with the early filter condition, the speedup of filter push-down is significant even with the `SELECT *` query. In the worst case, the filter and projection push-down add minimal or no overhead so it would be wise to enable these features by default.

3.9.4 Partitioning and Schema Inference

Next, we study the effect of the partitioning and schema inference techniques given different datasets. The results of this evaluation are shown in Figure 3.11. The optimistic schema inference depends on the average size of the matched records. Since for larger records, it would have to consume more bytes to build a schema on the first 1000 matches. It also depends on the complexity of the records and the level of nesting. The time for

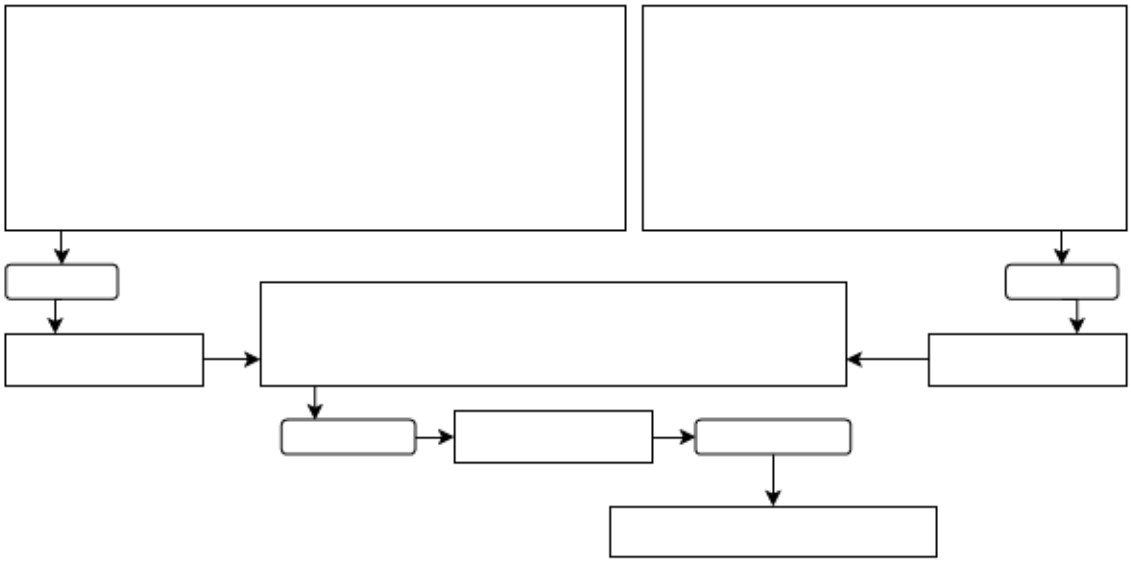


Figure 3.12: Complete use case example

the optimistic schema inference also incorporates the time for the key-collector, since they are performed simultaneously. Partitioning with speculation is always small relative to the total execution time, and this time also involves issuing the tasks and collecting the results, since it is performed as a separate stage. Some of this time can be saved by making the speculation as the first step in the parsing stage, should that be desired. The pessimistic schema inference can be faster or slower than full-pass partitioning, depending on the nature of the schema. Some may contain nested objects and arrays, which takes more operations to merge than a flat schema. The combination of the optimistic schema inference and partitioning with speculation provide basis for the best-case, while the combination of the pessimistic schema inference and the full-pass methods provide basis for the worst-case.

3.9.5 Detailed Use Case

To demonstrate the power of dsJSON in running arbitrary SQL queries on JSON files, we perform a slightly complex SQL query, that involves join, on data extracted from the OSM and the Wiki datasets. The OSM version in this experiment is smaller in size, but contains records with more metadata. From OSM, we select records within the US of type `boundary` (usually a state, or county boundaries, etc.) that has a reference to a Wikipedia record. From Wiki, we select the id, the English label and description, and all the references to other Wiki items. Then, both dataframes are joined based on the Wiki id. The final dataframe contains the geometry objects of the selected OSM records as well as all references to Wikipedia items related to it. We then store this dataframe in Parquet, which is a column oriented data format, just to demonstrate the full integration with SparkSQL. Being able to process two very big JSON datasets by extracting the desired information and perform an expensive join operation on them in an hour is extremely valuable. Note that both of these datasets are poorly supported by existing parsers, as discussed in Section 3.6, so a user that needs to perform a similar operation in a distributed fashion might need to spend a lot of effort in implementing a custom processor specific to their use case. While this is a more memory intensive query, Spark successfully completes it with the same fixed memory allocated for each executor.

3.10 Related work

This section goes over the most related work to the contributions proposed in this paper. First, we compare dsJSON to existing parallel implementations and show how it

differs based on the following attributes: A1) Can process data in the general JSON format in parallel, A2) Scales well to very large data sizes, A3) Can directly apply SQL queries or other types of analysis, and A4) Provides options to handle complex JSON schema, as summarized in Table 3.7.

Table 3.7: Comparing dsJSON to parallel implementations

Processor	dsJSON	Spark [4]	AsterixDB [22]	BEAST [66]	SparkJayway	JPStream [96]	Pison [95]
A1	✓					✓	✓
A2	✓	✓	✓	✓			
A3	✓	✓	✓	✓	✓		
A4	✓						

Existing parallel JSON parsers [96, 95] don't scale well to large data, mainly due to their memory requirements and the fact they are not designed following the BSP model, making it difficult to adopt their proposed design in distributed systems, as discussed.

Existing distributed systems only support a specific subset of JSON like JSON-Lines [4, 22, 165] and GeoJSON [66]. To process very big JSON data in these systems, it can either be done sequentially or the user must convert it first to the JSONLines format sequentially or using a customized implementation. dsJSON supports all of these formats as well as the general format, eliminating the need for sequential conversion. dsJSON also solves the issue of complex attribute selection by inferring a schema using multiple JSON-Path queries, making it possible to unify the structure of the matched records, in cases where it is not straightforward to unify them.

On top of not scaling to big data, most single machine implementation are not integrated with a data analytics systems, making it a challenge to perform even a simple type of analysis. dsJSON, however, supports this functionality out of the box, since it is fully integrated in Spark. Some prior work focused on analytics, including [64, 91, 35].

However, their focus is more on integrating JSON into existing RDBMS, but they don't tackle the issue of distributed processing of very large JSON files or schema inference.

Single machine environment parsers like [8, 132, 115, 98, 164, 42] focus on adding low level optimizations, like utilizing SIMD instructions. dsJSON on the other hand focuses on solving issues related to scalability and big data analysis. The optimizations they propose can be integrated into dsJSON in the future. SIMD optimizations can further improve dsJSON. The optimizations introduced in [98] can improve the skip functionality in dsJSON. The indexing techniques in [95] can be used to improve the full-pass partitioning, by utilizing SIMD instructions to track the positions of control characters. Furthermore, the optimizations in [115] can be used to improve the main parsing process and serializing the objects. There are several approaches for this integration: including using the SIMD supoort in the Vector API in Java, utilizing the Java-Native-Interface to communicate with C++ code, or implementing dsJSON in a low level language in a system that is compatible with the Dataframe API and the BSP model. There are many sequential JSON parsers, implemented in different languages with different features. Interested readers can refer to a JSON parsing benchmark [226] for details.

There are some similarities between parsing JSON and parsing XML, [96] goes through some of these differences and how working with JSON is more challenging. The work in [142] provides a parallel XML parser that divides the data based on the learned schema. It basically identifies the position in the structure at each partition based on the open tag of array elements, however, these open tags do not exist in JSON and this limits the splitting to array objects. In [97], they use the learned grammar of the XML data to

reduce the possible execution paths at each partition. We opted for avoiding techniques that consider multiple execution paths to avoid having to fully parse records more than once, which may not scale well and also does not fit the computational model where results are immediately sent to the next processing stages. Moreover, [31] provides a grammar based parallel parser that also supports JSON, but can only provide outputs once the entire data is parsed.

dsJSON takes distributed JSON parsing to a whole new level by providing a general-purpose parser that can support any JSON file. The work in [76] provides a parallel parser for the CSV format. Similar to our work, they provide a speculative approach and a fallback full-pass alternative, however, JSON requires more complex partitioning and schema inference. While we provide schema inference techniques, our focus was not on schema analysis and discussing the intricacies associated with it, refer to [28, 56, 60, 29] for details. [112] is similar in that it builds a data structure that stores the inferred types, but it focuses on finding discrepancies or updates in the schema. dsJSON instead infers a unifying schema based on attributes selected by the user as discussed earlier.

3.11 Conclusion

This paper introduced dsJSON, a full-featured scalable JSON processor for distributed shared-nothing systems. dsJSON is integrated into Spark to provide SQL query processing and complex data analytics. It includes several novel components starting with the projection tree for selective parsing, and the robust partitioning techniques, as well as, the techniques for schema inference. It overcomes limitations in existing systems, and fills

a gap that enables more complex analysis of large scale JSON data. Experiments on real data of up to two terabytes confirmed the scalability and efficiency of dsJSON.

Chapter 4

Towards Learned Geospatial Data Analysis & Exploration

4.1 Introduction

We are witnessing continuous growth in data-driven scientific methods that rely on the abundance of real data to drive scientific findings. Most of these applications are based on *data lakes* [161, 24] that store very large datasets and provide scalable query interfaces such as SQL. Studies show that 60-80% of data contains a geospatial component which is very important for data analysis [87]. Example datasets include temperature measurements [155], traffic prediction [200], and online text posts [144], with applications in urban planning [61], epidemiology [185, 181], and weather forecasting [143].

Recent advances in machine learning (ML) and deep learning (DL) are shifting data analytics from manual processes to model-driven approaches. Vector databases [215]

now enable robust semantic search, powering applications like Retrieval Augmented Generation (RAG)[123]. However, search quality depends on embedding quality, produced by models such as CLIP[171]. Meanwhile, geospatial data remains only supported in traditional data lakes. Our goal is to move closer towards better support for geospatial data by DL architectures.

Consider a realtor with no geospatial expertise searching for regions matching complex, poorly articulated criteria (e.g., amenities, density, proximity to roads and coastlines). Traditional GIS queries and custom programs for such tasks are difficult and require domain expertise, and processing large datasets. Recent advances in DL can help provide an alternative solution, where models can be trained to learn different types of properties about geospatial data. Then, they can be used to estimate those properties, or the embeddings they generate can be used to build advanced geospatial semantic and similarity searches. However, many challenges still exist in building such models. One challenge is how to best represent the data to feed into a DL model, as well as understand how well the models capture the different geospatial operations.

This paper studies how to enrich DL models to allow them to express a wide range of complex geospatial analytic functions. This is a step towards enabling DL-based geospatial query analytics where users can search thousands of geospatial datasets efficiently with the help of deep learning. To accomplish this goal, this paper breaks down the problem into two major components, *spatial data representation* and *spatial problem modeling*.

First, we study how to use three popular DL architectures for dealing with geospatial data. The first one is an image-based architecture, where the geospatial data is first

pre-processed into a fixed size geo-referenced histogram that substitutes the image. We use *ResNet* [85] and *UNet* [176] as reference models for the image-based architecture. The second one is a graph-based architecture, where the input of the considered model is a set of points having position coordinates, and each point is associated with additional non-spatial attributes. We adapt this approach by designing and testing variations of the *PointNet++* [169] architecture. The third is a vector-based model that encodes geometries into fixed-size vectors. To test this approach, we extend the Poly2Vec [188] encoder to work with a Transformer [206]. Our initial study showed that all techniques can work well for a variety of geospatial problems, but each has advantages and drawbacks.

Second, we model four different spatial problems using the three representations. This includes spatial data synopsis, spatial clustering, selectivity estimation, and walkability estimation. These problems involve different spatial operations and data characteristics, providing a lot of insights. Our goal is to study how the three proposed data representations and model architectures can capture these different problems while speeding up the query processing.

To evaluate the three models, we conducted an extensive experimental evaluation using synthetic and real datasets. Our results show that the three architectures perform well, with some trade-offs. The graph-based approach works better when data are more dense and distances between geometries are very small. The other two approaches scale much better since the input is always fixed in size. The image-based approach works better for inputs that join multiple datasets. All approaches make inferences much faster than computing the exact values, while maintaining high accuracy. For the data synopsis problem, the

speed-ups reach more than 600x in the best case and 8x in the worst case. These results are very encouraging and open several new directions for future research.

The remainder of this paper is structured as follows. Section 4.2 discusses the three representation methods that we considered. Section 4.3 goes over the four geospatial problems and discusses their definitions, data preparation, and model architectures. Section 4.4 details the extensive experimental evaluation and discusses the results. Following that, Section 4.5 presents the related work. Finally, Section 4.6 concludes the paper and discusses future directions.

4.2 Geospatial Data Representation

Geospatial vector data involves a set of records with geometric shapes and non-geometric features. This data cannot be used directly with existing deep learning architectures. A conversion step is required to transform the data into a more suitable representation. This step is not always straightforward since, based on different characteristics of the data, the conversion step may result in a loss of information affecting the quality of the modeled problem. The remainder of this section discusses some preliminaries, and then presents three approaches, **GeoImg**, **GeoGraph**, and **GeoVec** to properly represent geospatial data to exploit existing models for a set of well-known problems.

4.2.1 Preliminaries

In the problems that we are studying, a single input is a geospatial dataset $D = \{g \mid g = \langle \text{type}, \text{coords}, \text{attr} \rangle\}$, which is a set of vector geometries. Each geometry $g \in D$ is a tuple

including the type of the geometry, which could be: *Point*, *LineString*, or *Polygon*, an ordered list of coordinates, each one as a pair (longitude, latitude), representing together the position and shape of each object. Finally, each geometry could also be associated with a set of attributes. For example, a dataset could be a set of points each associated with a temperature value. Our objective is to model problems of the form $O = f(D)$, where f is an analytical function, and O is the target tensor it produces. Moreover, some problems require analyzing two or more input datasets which have the signature $O = f(D, D', \dots)$. For these problems, the target value has a fixed shape, where $O = m \in R^{d_1 \times \cdots \times d_k}$, where the number of dimensions k, and the size of each dimension depend on f. Furthermore, some problems have a target dataset D_T, which contains target geometries for the analytical function to compute values for. These problems will have the signature $O = f(D, D_T)$ or $f(D, D', \dots, D_T)$. In this case, $O = M \in R^{|D_T| \times d_1 \times \cdots \times d_k}$. Note that the first dimension of O is equal to the cardinality of D_T, while the others are fixed and defined based on f. For example, the target m could be the position of the neighborhood of the point with the maximum average temperature, while M will contain a label for each geometry of D_T that is produced for clustering D. Note that for some problems the geometries in D_T can be the same as D, and some definitions of f may also take additional parameters such as distance thresholds. In summary, O can be:

$$O = \begin{cases} m = f(D) \textbf{ or } f(D, D', \dots) & \textbf{where } m \in R^{d_1 \times \cdots \times d_k} \\ M = f(D, D_T) \textbf{ or } f(D, D', \dots, D_T) & \textbf{where } M \in R^{|D_T| \times d_1 \times \cdots \times d_k} \end{cases} \tag{4.1}$$

Our goal is to train a set of DL models that resemble analytical functions f involving complex geospatial operations. However, with existing DL techniques, we cannot

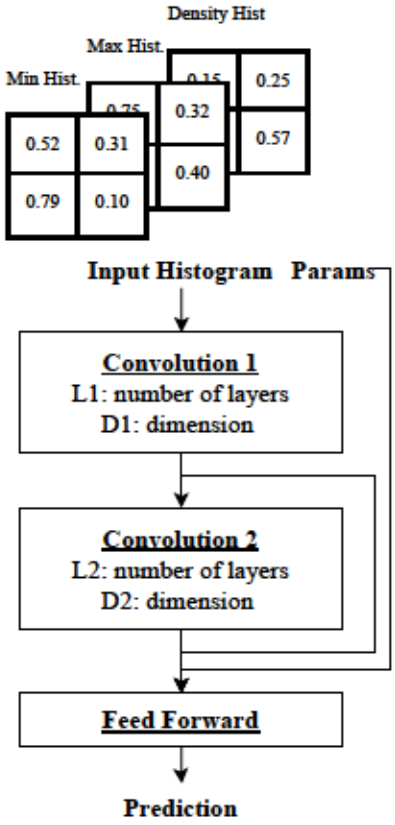


Figure 4.1: GeoImg architecture

directly feed the datasets D or produce target values for the geometries of D. We discuss three representation approaches next.

4.2.2 GeoImg: Image-based Representation

This section describes our first approach for image-based representation. This part is inspired by Convolutional Neural Networks (CNNs) which are commonly used for working with images. We chose this as our first approach because CNNs are versatile and can be used to model a variety of problems. Representing our data as images enables the use of

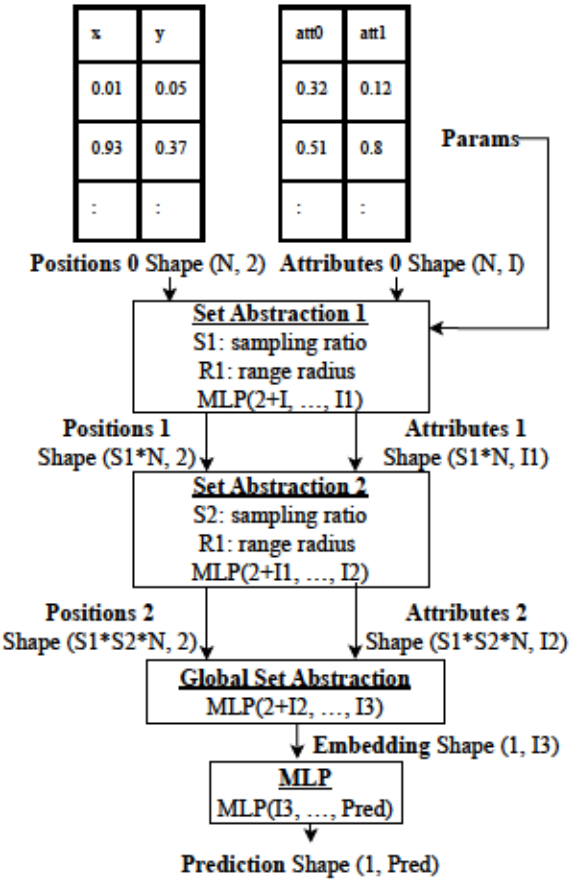


Figure 4.2: GeoGraph architecture

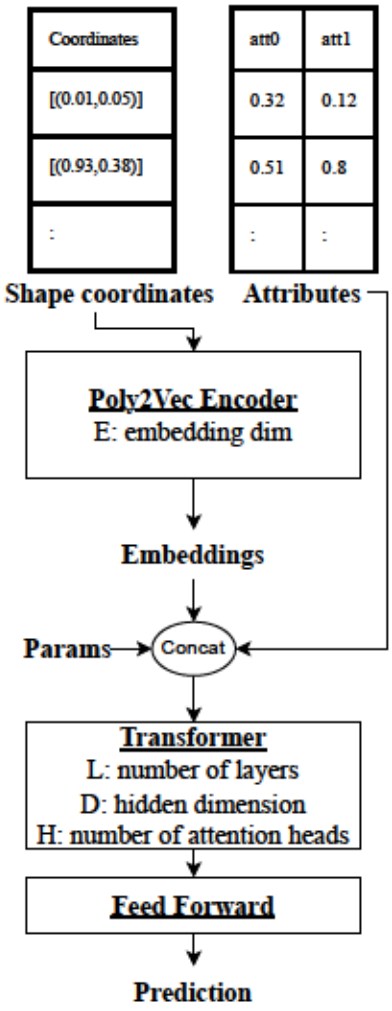


Figure 4.3: GeoVec architecture

existing CNN architectures. The simplest approach for converting our data to an image is by rasterizing them. In graphics, rasterization means plotting the shapes as an image of a given resolution, similar to rendering them on a map. However, one of the drawbacks of this is that it loses subtle features that get lost in small resolutions, making this approach more suitable for smaller regions and sparse data. Alternatively, histograms work by dividing the input reference space, into a grid of fixed-size cells and storing a summary of the geometries that fall within each cell. Previous work used a similar representation, but we show that this can be generalized to a variety of problems.

To address these limitations, this part introduces **GeoImg**, an approach for representing geospatial data as histograms for working with image-based DL models. We chose histograms because they are more flexible for representing a large set of problems, as long as we define a suitable set of aggregate functions. After defining the histograms, it becomes easy to utilize existing CNN architectures such as ResNet [85] and UNet [176]. We discuss this representation in four parts, including representing geometry positions, non-geometry features, handling multiple-input datasets, and representing target values, as shown next.

Representing geometry positions

To represent geometry positions inside a histogram, we first transform the geometries of a dataset D into a metric reference system, which is required if coordinates are latitude and longitude, so that the distance between points becomes meaningful and corresponds to the real distance. We also compute the Minimum Bounding Rectangle (MBR) of D to determine its reference space. Then, we select a histogram *size*, to compute the desired subdivision of the reference space into cells. Finally, we define an aggregate function

$f_{Fcount}(D, c)$ which returns the number of geometries intersecting a grid cell c divided by the input cardinality $|D|$. In this case, the value of each cell in the histogram will have the percentage of all geometries that intersect the range of coordinates that the cell represents, and the sum will always be 1.0 for points, but could be larger for other types of geometries when some shapes intersect multiple cells. The histogram $H_p = Histogram(D, f_{Fcount}(), size)$ is then created.

Representing non-geometric attributes

To represent additional attributes, we use the same $size$ of H_p (the positions histogram), but we define an aggregate function for each additional attribute. As a result, when i attributes are present in D, and we use one aggregation per attribute, we will end up with i histograms. These histograms are stacked on top of each other, starting with H_p, making a final input histogram $H \in R^{(i+1)\times size\times size}$.

Representing multiple datasets

For multiple datasets D, D', D'', ..., we first have to create the histogram based on the global reference space, i.e. the MBR of the union $D \cup D' \cup D'' \ldots$. Then, to combine the histograms, we stack them on top of each other since they are all of the same resolution. For example, given two input datasets with i_1 and i_2 attributes, respectively, and $dim = 64$, we will have a histogram stack $H_1 \in R^{(i_1+1)\times 64\times 64}$ and $H_2 \in R^{(i_2+1)\times 64\times 64}$. We combine them into one histogram $H \in R^{(i_1+i_2+2)\times 64\times 64}$.

Representing target values

For global values representing features of the whole input dataset D, that is, one target for the entire dataset. The representation is a single tensor m that contains such values. For local values, i.e., output for each target geometry, we define a new histogram for each target value, where each cell of the histogram represents all geometries it contains. In this case, the final result is obtained by assigning to each geometry of the cell the same target value. Note that the dimension of the output histogram does not need to be the same as the input, even though it can be the same for many applications.

While this method can capture more information than simply rasterizing the geometries, it still has some drawbacks. One drawback is that it cannot capture small patterns that are smaller than the cell size. Also, all geometries on the same cell will end up with the same label. For these reasons, we consider an alternative approach that avoids this type of data loss.

Model architecture

As mentioned, we used CNN-based architectures for this representation. For problems with a fixed target size m, we use a ResNet [85] architecture. An example of this is shown in Figure 4.1. In this example, the network takes our histogram representation that summarizes the input dataset D, and produces a fixed-size output m. For problems with variable output size M, we use a UNet [176] architecture, and the target is represented, as discussed earlier. The first part of a UNet is similar to a ResNet, but it has additional layers that perform up-convolution to produce an image output. UNet architectures are typically

used for problems like image segmentation [176]; here we use it to make estimations based on geographic positions.

4.2.3 GeoGraph: Graph-based Representation

This section introduces an alternative approach that is tailored for graph based DL architectures. Specifically, we focus on point-set based approaches that build a nearest-neighbor graph at run-time. These types of architectures operate directly on a set of points and are inspired by point-cloud-based models, e.g., PointNet++ [169]. They enable the model to have a richer representation and capture the local relationships among the points more accurately than an image-based approach. The same properties that enabled this approach to be successful in the point-cloud domain can also be beneficial in the geospatial domain. Additionally, one advantage of this approach over GeoImg is that it works with variable-sized datasets D, whereas an image-based approach represents any input dataset D in a fixed size, potentially leading to information loss. Furthermore, this approach does not require a heavy pre-processing step. However, geospatial problems have different characteristics and scales, and these types of models have not been studied in this domain yet. The remainder of this section introduces **GeoGraph** for representing geospatial data following a point-set approach. We iterate over its four parts, next.

Representing geometry positions

In this approach, for $|D| = n$, when the type of geometries is $Point$, the positions are represented as a tensor $G \in R^{n \times 2}$, where each point is given by two coordinates: the first column represents the longitude, and the second one is the latitude. For other

types, we chose to use the centroid point to represent them, but we added additional geometry descriptor attributes. For example, attributes about the geometry could represent the coordinates of its MBR, which makes $G \in R^{n \times 6}$, where the first and second columns represent the longitude and latitude of the centroid of the shape. The other columns represent the boundaries of the MBR. Other attributes can also be used, such as the area of geometries or the length or width of geometries. Furthermore, we transform the coordinates to a metric reference system as done for the GeoImg approach. In addition, we also scale the coordinates in the range [0..1], using the minimum and maximum coordinate values in each column for the given dataset.

Representing non-geometric attributes

Additional attributes associated with geometries can be represented as a tensor $A \in R^{n \times j}$, where j is the number of attributes, so each row contains the attributes associated with the corresponding geometry in G. Pre-processing can include normalizing the features for each set independently or using global statistics about the entire training set, which is composed of a collection of datasets D.

Representing multiple datasets

This simply requires generating a geometry position tensor G and an attribute tensor A for each one of the input datasets. However, in this case, the positions must all be normalized using the MBR of the union of all input datasets before generating the position and attribute tensors.

Representing target values

Target values represent the output that we need the model to learn. In the case of a single aggregate prediction for an entire dataset D, the target becomes a tensor m which will have a fixed shape. Depending on the range of values, we might need to normalize them to allow the model to work more effectively. In case we have a target set of geometries, D_T, we need to represent its geometries similar to the positions tensor G for the input dataset, and the target tensor M will have its first dimension equal to the cardinality of D_T, while the other dimensions will be fixed similar to m depending on the shape of the target values. The geometries represented in D_T can be exactly the same as the ones of D, but this is not always necessary, since the specific problem can generate values for geometries different from those in the input.

Model architecture

An example of the architecture of this model for problems with a fixed target size m is shown in Figure 4.2. It is characterized by four main components. As input, it takes a set of points, represented by their position coordinates, and a set of attributes for the points.

The first layer of the model is called a Set Abstraction layer, referred to as SA1. This layer has three main steps. **(i)** A sampling step with a percentage parameter that selects a subset of the points to propagate to the next layer. **(ii)** A range query step that computes the edges needed for message passing for the sampled points and takes a range and a maximum number of neighbors as parameters. **(iii)** Finally, a multilayer perceptron

(MLP) is trained based on the coordinates of the points and their features by applying a message-passing step per the design of graph neural networks. The MLP part is the only trainable component in a Set Abstraction layer. The output of this layer is the coordinates of the sampled points and their associated features from the MLP. Similarly, additional SA layers can be used, that are then followed by a global abstraction layer that summarizes the entire input dataset. This is then followed by a final component that produces the final target m. For problems with a target dataset D_T, additional layers are used. These are called feature propagation layers. This works by reversing the process and generating new features for the larger point set in the previous layer, from the features of the points in the lower layer, until producing one prediction for each point in the target dataset. The features are interpolated by finding the k-nearest neighbors in the points from the previous step, and using an MLP to generate a new one for the target point. Refer to [169] for more details on how this architecture works. For problems where there are multiple input datasets, we add an SA layer that joins datasets. In this method, we use one dataset as the reference points and use the other as the sampled points. From this layer, we get a set of features for each point in the second dataset from its nearest points in the first dataset.

4.2.4 GeoVec: Vector-based Representation

This section introduces our third approach based on a Transformer [206] model architecture. Transformers are mainly used for natural language processing (NLP) applications, but have been applied successfully for a variety of applications including geospatial data [131, 131]. However, there are still challenges related to how it generalizes to different types of geospatial problems, and it is not clear how it compares to the other approaches.

This approach requires less preprocessing and is more flexible than GeoImg. However, it is less flexible than GeoGraph, since there is a fixed upper bound for the number of geometries. The remainder of this section introduces **GeoVec**, a Transformer-based architecture, where geospatial objects are represented as vectors. We discuss its components next.

Representing geometry positions

In NLP, words are typically represented using word vectors and positional encoding, in a way that ensures that the relationships among the words are maintained. Similarly, for the geospatial objects, we must use embeddings that ensure that the spatial relationships among the objects are still maintained. To achieve this, we use Poly2Vec [188], a recently proposed geospatial encoder that has been shown to produce vector embeddings that maintain the spatial relationships among the objects. This encoder takes as input a sequence of coordinates for each input shape. For example, if all the inputs are points, the data is represented as $G \in R^{n \times 1 \times 2}$, for rectangles it is $G \in R^{n \times 5 \times 2}$, since a polygon must start and end with the same point. For mixed shapes, the representation becomes $G = \{G_i \in R^{c_i \times 2}\}_{i=1}^{n}$, where c_i is the number of coordinates for each shape. This encoder produces an embedding of size e. Since the Transformer architecture has a fixed upper bound b for the input sequence, we perform a sampling step if $n > b$, and pad the sequence with zeros if $n < b$. The sampling is performed prior to encoding the geometries. Therefore, after this step, the output is always $V \in R^{b \times e}$.

Representing non-geometric attributes

The attributes are represented similarly to GeoGraph as discussed in Section 4.2.3. However, this step only includes the attributes of the sampled objects. After this step, the attributes tensor is always $A \in R^{b \times j}$, where j is the number of attributes. This tensor is concatenated with the geometry embeddings to make the input to the Transformer model to have the shape $R^{b \times (e+j)}$.

Representing multiple datasets

For this, first, we merge all the datasets as one, but we add a one-hot encoded label determining which dataset each record belongs to. Then, we perform stratified sampling based on the dataset label, ensuring the proportion of the datasets is maintained. For two datasets, each with $j1$ and $j2$ number of attributes, we have $A \in R^{b \times (j1+j2+1)}$. The final input to the model after encoding becomes of the form $R^{b \times (e+j1+j2+1)}$. Any additional parameters to the input are also appended as columns. This matrix is also padded with additional zero columns to ensure that number of columns is a multiple of the number of attention heads, which is a requirement for this architecture. Also, the data is normalized similar to GeoGraph.

Representing target values

When having a fixed target size m, this is represented similarly to GeoGraph. However, when having a target set of geometries, D_T, this is not as straightforward. When the geometries in D_T are the same as the input geometries in D, we set the target for the model to be the labels of the input geometries, which is of fixed size $M \in R^{b \times d_1 \times \cdots \times d_k}$.

Then, we assign the labels for all the points in D_T based on the nearest points in the sampled input. If the geometries in D are different from D_T, first we must assign labels to the input geometries based on the labels of the nearest target geometries.

Model architecture

In this architecture, first, the geometries are passed to the Poly2Vec encoder. Then, the embeddings are concatenated with the geometry attributes. This is then passed as input to a Transformer model. An example of this architecture is shown in Figure 4.3.

4.3 Geospatial Problems

This section discusses the four target geospatial problems that we consider. For each of them, we provide the definition of the problem, the input and output format for each specific model. The goal is to show that the proposed models can be applied to a variety of spatial data analysis problems and can capture a wide range of spatial data characteristics, e.g., proximity and data density, producing an approximate solution faster than the exact solutions. Table 4.1 summarizes the four problems and the corresponding proposed models as we further detail in the rest of this section.

4.3.1 Spatial Data Synopsis

Spatial data can have many characteristics that are used in applications, e.g., skewness, symmetry, coverage of geometries, or density analysis regarding specific thematic

Table 4.1: Problems definition summary

Problem	Approach	Model Name	Input Shape	Output Shape
Synopsis $m = f(D, k, r)$	GeoImg	GI-S	$H \in R^{3\times64\times64}$	R^9
	GeoGraph	GG-S	$G \in R^{\lvert D\rvert\times2}$ $A \in R^{\lvert D\rvert\times1}$	
	GeoVec	GV-S	$V \in R^{1024\times(32+1)}$	
Clustering $M = f(D, D_T, \epsilon_1, \epsilon_2, MP)$	GeoImg	GI-C GI-CP	$H \in R^{3\times64\times64}$	$R^{3\times64\times64}$
	GeoGraph	GG-C GG-CP	$G \in R^{\lvert D\rvert\times2}$ $A \in R^{\lvert D\rvert\times1}$	$R^{\lvert D_T\rvert\times C}$
	GeoVec	GV-C GV-CP	$V \in R^{1024\times(32+1)}$	$R^{1024\times C}$
Selectivity $m = f(D, q)$	GeoImg	GI-E	$H \in R^{4\times128\times128}$	R^1
	GeoGraph	GG-E	$G \in R^{\lvert D\rvert\times6}$	
	GeoVec	GV-E	$V \in R^{1024\times(32+4)}$	
Walkability $m = f(D, D')$	GeoImg	GI-W	$H \in R^{11\times64\times64}$	R^1
	GeoGraph	GG-W	$G_D \in R^{\lvert D\rvert\times2}$ $G_{D'} \in R^{\lvert D'\rvert\times2}$ $A_{D'} \in R^{\lvert D'\rvert\times10}$	
	GeoVec	GV-W	$V \in R^{1024\times(32+11)}$	

attributes. However, as data sizes become very large, which is typical for geospatial datasets, the computation of these characteristics becomes very expensive.

Problem definition

Spatial analytics often require the computation of aggregate values, called **synopsis**, that summarize different spatial features of a given dataset. We consider three categories of synopsis: (i) finding extreme values and the regions where they are located (this can require a preliminary phase to calculate the k nearest neighbors for all geometries in the dataset and to compute the average of the thematic attribute among its neighbors); (ii) compute the K_{value} of a dataset, which is a measure of the connectivity of a set of geometries, given a maximum distance r; this value is also very expensive to compute: when K_{value} is large, most of the geom in the set are very close to each other. When it gets lower, it means that the geoms are farther apart, and we expect them to be sparsely distributed

throughout the reference space; (iii) compute the box-count values, $E_0(D)$ and $E_2(D)$, as defined in [33]. These are good descriptors of the skewness of the data.

These operations can be formally described as follows. Given a dataset $D = \{p_i = \langle Point, g_i, v_i \rangle, ...\}$, and the parameter k, the average value of the thematic attribute on the k-nearest neighbors of each point $p_i \in D$ is computed as:

$$avgAtt(D, k) = \{(p_i, avg_i), ...\} \text{ where } avg_i = \frac{1}{k} \sum_{x \in kNN(D, p_i)} x.v_i$$

From $avgAtt(D, k)$, we then extract four results: the maximum and minimum of avg_i and their location p_{max}, and p_{min}.

The computation of K_{value} of D with radius r is as follows:

$$K_{value}(D, r) = \frac{1}{n \times (n-1)} \sum_{x \in D} |range_r(D, x.p_i)|$$

where the function $range_r(D, x.p_i)$ computes the range query for a point p_i given a window r. Finally, the computation of $E_0(D)$ and $E_2(D)$ works by dividing the space into a grid with cells of side length r, and counts the number of cells in the grid that intersect at least one point in the set (for $E_0(D)$) and the number of points intersecting each cell (for $E_2(D)$). It keeps increasing the number of boxes in the grid until the counts do not change. This was shown to approximate the skewness in the spatial distribution of geometries.

These problems help us to evaluate if the models are able to predict not only a numeric aggregate value, but also properties that depend on the spatial distributions of the

geometries and their relative position and distance. Moreover, they are all expensive to compute and also capture complex spatial behaviors.

Target values

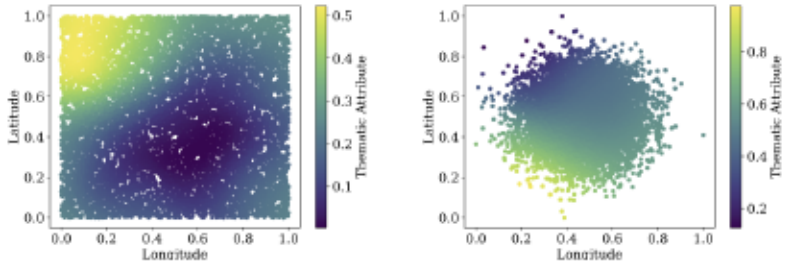
Considering Equation 4.1 proposed in Section 4.2, for spatial synopsis the target value O is computed as an array m:

$$[max(avg_i), min(avg_i), p_{max}, p_{min}, K_{value}(D, r), E_0(D), E_2(D)]$$

These are a total of nine values since the positions are represented by two coordinates. Therefore, the function signature is of the form $m = f(D, k, r) \in R^9$, as shown in Table 4.1.

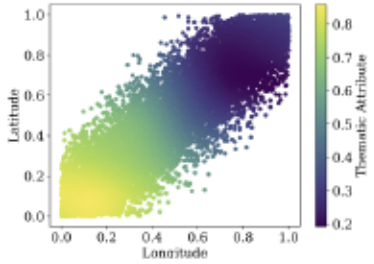
The input datasets containing 2D points have been synthetically generated using the Spider Generator [106], producing datasets with various distributions and including values of a thematic attribute.

Figure 4.4 shows some examples of the generated data, where the color associated with each point denotes a different value of the thematic attribute. Given the generated datasets with associated attributes, we compute the target values above described by applying the implementation of the algorithms representing the exact solution, building in this way the ground truth for training the models. We then pre-process the data following the approaches illustrated in Section 4.2, and build three models GI-S, GG-S, and GV-S. These are summarized in Table 4.1 and discussed next.

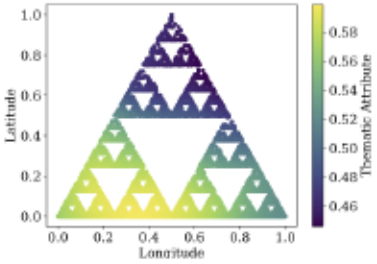


(a) Uniform

(b) Gaussian



(c) Diagonal



(d) Sierpinski

Figure 4.4: Example of synthetic datasets with both different spatial distributions and distributions of thematic attributes.

GeoImg model (GI-S)

This model is based on a generic ResNet [85] architecture that takes the histogram as input and is trained to produce as output the tensor of target values.

We set the histogram *size* to 64, by default, and define two aggregation functions for the thematic attribute, f_{Fmax} and f_{Fmin}, that we use to assign a summarized representation of the thematic attribute to each cell. This makes the input histogram $H \in R^{3\times64\times64}$. Given a cell c the functions f_{Fmax} and f_{Fmin} compute the minimum and maximum values of the thematic attribute, respectively, considering the points in c. The model takes as input H, and two parameters k and r.

GeoGraph model (GG-S)

This model is based on the original PointNet++ [169] model for point-cloud classification. The input is organized as follows: (i) the coordinates of the points are represented as a tensor G with shape $(N, 2)$, N representing the cardinality of the dataset D, and the thematic attribute values as a tensor A with shape $(N, 1)$, following the approach described in Section 4.2.3.

GeoVec model (GV-S)

This model is based on a Poly2Vec approach [188]. First, a sampling (or padding) phase is applied to fit the upper bound $b = 1024$ of the Transformer model. Then, the geometries are passed to the Poly2Vec encoder, with the embedding size $e = 32$. Then, the embeddings are concatenated with the thematic attribute and passed as a single input to the Transformer, resulting in an input tensor V with shape $(1024, 32+1)$. Furthermore, any parameters are appended to the input; the columns are padded to a size that is a multiple of the attention head dimension.

4.3.2 Spatial Clustering

Clustering is one of the most commonly used techniques for spatial analysis. It is used to find regions where specific events are concentrated. One motivation for considering this problem is that it requires making a prediction for all points in the input, and not one prediction for the entire input. Similar problems might be related to outlier detection and spatial interpolation.

Problem definition

DBSCAN is one of the most popular clustering algorithms. It has two main parameters, a distance ϵ and a minimum number of points MP. It starts by selecting a random point P and determining if it is a *core point*, i.e. if it has at least MP neighbors within ϵ distance from it. If P is a core point, it is used to initialize a cluster, and the process continues by adding points to it that are within ϵ distance of any point in the cluster. Then, the process repeats until no more *core points* are found. Points that do not end up associated with any cluster are labeled as *noise points*, or outliers. Many variants of DBSCAN have been defined in the literature, in particular, ST-DBSCAN [36] is an extension tailored for dealing with spatio-temporal data. In this case, two distance thresholds, ϵ_1 and ϵ_2, are defined, one for the spatial and one for the time dimension. However, the second distance is not strictly required to be related with the time dimension but could be associated with any other attribute. In the original paper [36], the authors also do experiments where the second distance refers to the temperature measured in each point, and the algorithm detects clusters based on the spatial distance, as well as an additional attribute, like temperature or time. We define this problem on the basis of this generic definition. The objective is to build models that resemble the clustering labels produced by this algorithm given some parameters, since our focus is on supervised learning.

Data preparation

To prepare the collection of datasets, a recent copy from the Global Historical Climatology Network (GHCN)-Daily [155] dataset has been downloaded. For this problem,

only the longitude and latitude as the point coordinates and the maximum temperature associated with each point are used. The data is then partitioned by year and month, and each group is saved in a separate file. This results in having one file for each month that contains the average maximum temperature for each weather station across the world. Then, the ground truth labels are generated using ST-DBSCAN for clustering; we use the implementation provided by [45]. The algorithm is applied to every month separately, and the target value for each point is composed of the identifier of the cluster it belongs to.

Next, we save the labels for each point in every month based on the cluster label produced by ST-DBSCAN, for all parameter sets. We show an example of cluster assignment in Figure 4.5. Since the order of the labels produced by the algorithm has no meaning, we reorder the labels based on the average temperature of each label. The outliers are given label zero, then the cluster with the minimum average is given label one, and so on. Finally, we randomly select 80% of all the years to be used for training and 20% for testing. Furthermore, the temperature is scaled to [0..1] using the global minimum and maximum temperatures.

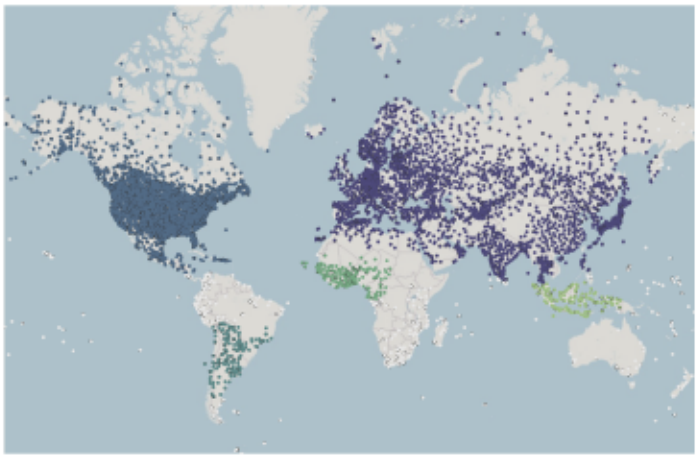


Figure 4.5: Weather Station Clustering Example

Target values

For this problem, the input is a dataset D and a target dataset D_T, as described in Section 4.2, where all the geometries are of type *Point*, and each point is associated with an attribute, representing for example the measured temperature. The geometries in D_T are the same as D, since we are producing a label for defining the cluster each point belongs to. Considering Eq. 4.1 proposed in Sec. 4.2, for this problem the target value is computed as the tensor $M = f(D, D_T, \epsilon_1, \epsilon_2, MP) \in R^{|D_T| \times C}$, where C is the maximum number of clusters, since the labels are one-hot-encoded, and ϵ_1, ϵ_2 and MP are the clustering parameters. This function generates one prediction for each point in D_T, i.e. the probability to belong to each cluster in C.

We build six models for this problem. Two models for each representation, with one being parameterized, and one trained on fixed parameters. The models are labeled GI-C, GG-C, and GV-C, for the ones with fixed parameters. A summary is provided in Table 4.1.

4.3.3 Clustering

GeoImg model (GI-C)

We use a UNet [176] like architecture.The input histogram H is similar to the first problem, as defined in Section 4.3.1. Thus, $H \in R^{3\times64\times64}$. For the target, we represent the output as a histogram with shape $(3, 64, 64)$, and assign each label a unique color. Each cell will have the color corresponding to the most common label associated with the points it contains. We implement a function that translates a color back to a cluster label. GI-CP is similar but takes parameters with the input.

GeoGraph model (GG-C)

This model is based on the original PointNet++ [169] model for segmentation. Given a set of points, this model makes a prediction for each point in the set, instead of making a single prediction for the entire set. The first part of this model works the same as the original model, reported in Fig. 4.2, until generating an embedding for the set from the Global Abstraction layer. Then, a few more layers are added to propagate the features back to all points in the input using interpolation.

The input is organized as follows: (i) the coordinates of the points are represented as a tensor G with shape $(N, 2)$, N representing the cardinality of the dataset D, and the thematic attribute values as a tensor A with shape $(N, 1)$. The output in this case is a tensor of dimension (N, C), since the label is one-hot-encoded, and the model predicts probabilities for each point.

GeoVec model (GV-C)

The adopted architecture is similar for all problems. Inputs are the same as for GeoGraph model, but limited by the upper bound b of the Transformer layer. The target output represents the cluster labels for each point in the input. The other points are assigned labels based on the closest point in the points used in the input set.

4.3.4 Selectivity Estimation

This problem has applications in database systems and query optimization. We include it as an example of a problem that involves working with shapes other than points and in the presence of another parameter beside the dataset D. This can be a simple query window q for the selectivity of range query, or another dataset D' for the selectivity of spatial join.

Problem definition

Given an input dataset D and a geometry q defining a range query, the objective is to estimate the percentage of geometries $g \in D$ that intersect with q. This problem is based on [34], which studies selectivity for both range and join queries of spatial data, using a histogram approach. Due to space constraints, we consider in the following only the selectivity estimation of the range query operation, since the selectivity of join can be considered an extension of this one, where each geometry of the second dataset represents a query window for the geometries of the first one.

Target values

We start from a subset of the datasets used in [34] which includes several datasets of rectangle geometries following different distributions. For each of these spatial datasets D, many range query windows q have been identified and the corresponding selectivity value has been computed. Therefore, this problem is in the form $m = f(D, q) \in R^1$. Selectivity is a value between 0 and 1. We build three models for this problem: GI-E, GG-E, and GV-E, summarized in Table 4.1, and described next.

GeoImg model (GI-E)

We use a model that is very similar to $GI - S$ model, and using as input a similar histogram according to the reference work [34]. The histogram representation still matches our representation approach described in Section 4.2.2. The aggregation functions include the averages of the area, the horizontal length, and the vertical length of all geometries intersecting each cell.

GeoGraph model (GG-E)

The difference between this model and the one used for the first problem is that in this case the inputs are the MBRs of a geometry.

An MBR is compactly represented by four numerical values: x_{min}, y_{min}, x_{max} and y_{max}, representing the minimum and maximum x and y coordinates of one geometry, respectively. We also need to extend the notion of distance to work with rectangles instead of points. There are several ways to compute the distance between two rectangles. The simplest one is to pre-compute the centroid of each rectangle and then compute the Euclidean

distances based on that. The centroid is defined as $(\frac{x_{min}+x_{max}}{2}, \frac{y_{min}+y_{max}}{2})$. The input geometries are represented as a tensor $G \in R^{|D|\times 6}$, where the first two columns represent the centroid used when computing the distances in the range query, and the last four columns represent the MBR used in the message-passing step.

GeoVec model (GV-E)

The adopted architecture is similar for all problems. We used the centroid of the rectangles to represent the geometries. First, we experimented with representing the rectangles using five coordinates as polygons. However, this was not efficient since each input includes thousands of shapes, and the encoder step was taking longer time. Instead, we represented the rectangles similar to GG-E, using the centroid and the MBR as additional features. Furthermore, sampling and padding are applied appropriately like the previous problems.

4.3.5 Walkability Estimation

The last problem that we consider is for walkability estimation. The motivation to consider this problem is due to its complexity, given that it requires a spatial join operation involving two separate datasets, followed by an aggregation step. Building a reasonably accurate model for this problem will give us more confidence about the suitability of the two approaches for general spatial analysis.

Problem definition

We define the walkability of a given geographic region to be about accessibility of various amenities and services from any location within that region. Each geographic region is given a score from 0 to 1, with 0 representing non-walkable regions. The calculation involves two different sets of points. The first set represents nodes in the road network, but only for the type of roads that are pedestrian-friendly. The second set includes the points of interest (POI) within the same region, from a fixed set of ten categories, like shopping, restaurants, etc. The main idea is that a point on the road network that is within a very close distance from at least one POI of all these categories are considered to have a perfect walkability score. The score for a region is simply the average of the scores for all points within its road network.

The first step in computing the score is finding the nearest point from each POI category for every node in the road network and storing the distances. The score for a specific node in the network is computed on the basis of these distances. A POI that is within a half-kilometer distance is given a score of 1, while those that are farther than two kilometers are given zero. Then, the scores for each category are averaged to produce the score for each node. Finally, the scores for each node in the road network are averaged, and the value is the walkability score for the entire region. These steps are shown in Fig. 4.6.

Data preparation

First, we extracted road networks using [41] and the points of interest for several geographic regions from OpenStreetMap (OSM). Then, we subdivide those large regions

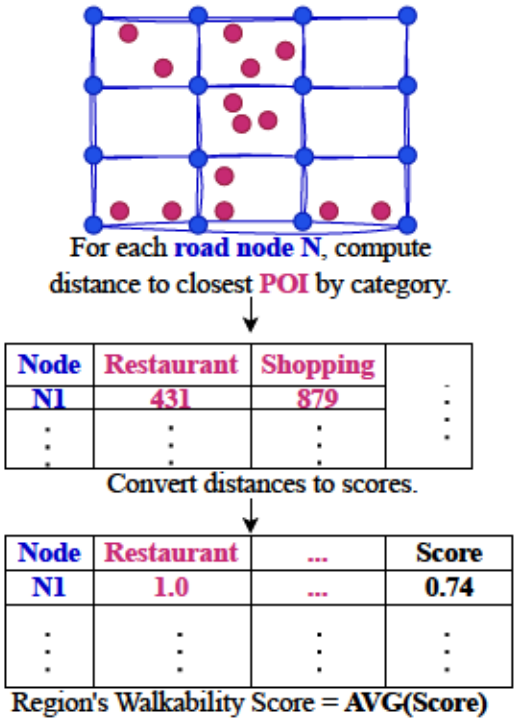


Figure 4.6: Example of walkability computation.

to smaller areas, to make the computations more feasible and to increase the number of examples for training and evaluation. After that, we computed the exact walkability scores as described earlier.

The resulting data could be very unbalanced. It could be quite common to have a very small number of partitions that have a perfect walkability score, while low walkability scores have a very large number of examples. Therefore, to increase the effectiveness of the training, an augmentation phase is necessary. In particular, two augmentation strategies are used: we rotate the points around the region center with a random angle, and we shuffle the labels of the POI categories randomly. Both changes do not affect the walkability score because it is not dependent on the direction and all the POI categories have the same weight.

Finally, the data is pre-processed following our discussion in Section 4.2, noting that this problem involves two datasets.

Target values

This problem can be formalized as a function $m = f(D, D')$ where D is the nodes dataset, and D' is for POIs. The value m represents the walkability score. We train three models for each representation, GI-W, GG-W, and GV-W, shown in Table 4.1 and described next.

GeoImg model (GI-W)

We use an architecture similar to the first problem. The input is represented as a histogram $H \in R^{11\times64\times64}$, where we have one histogram channel for each type of points: one for the road network, and the rest for the POI categories, one for each. All are based on the aggregation function f_{Fcount}, as described in Section 4.2.2.

GeoGraph model (GG-W)

We experimented with different variations for how to perform the joining of the two datasets. We used the version that produced the best results. It first includes a set abstraction layer between the POI points only. Then, the following set abstraction uses the POI points and their features produced from the first layer as the reference points, and computes features for the road nodes. Then, it is followed by a similar architecture as the one used in the data synopsis problem.

GeoVec model (GV-W)

The adopted architecture is similar for all problems. The main difference here is using a stratified sampling step to ensure that the data is sampled in proportion to its size. We use all the labels in the stratified sampling including the labels of the points of interest, and the label that indicates the dataset. This ensures that points from all types of points of interest in the region are included.

4.4 Experiments

In this section, we run an extensive set of experiments to evaluate how both types of representations perform for all four problems.

4.4.1 Setup

Hardware

For CPU based processing, such as when computing ground truth values, the tasks were ran on two *AMD EPYC 7713 64-Core Processor* with a total of 256 threads and memory of *128GB*. GPU-based tasks, e.g., training and inference, were run on a server with *AMD EPYC 7543 32-Core Processor* with a total of 64 threads, 256GB of memory and *NVIDIA A100-SXM4-80GB* GPU.

Datasets

The datasets used in the experiments are grouped into different collections, which have been detailed in Table 4.2 based on their type, i.e., synthetic or real, and the problem

for which they have been considered. In particular, we have: (i) collections that contain synthetic datasets, i.e., type = syn, generated through the Spider Generator tool [106]. These collections are grouped based on the considered spatial distributions: uniform, diagonal, Gaussian, Sierpinski, bit, or parcel. The suffix "_large" has been added to distinguish synthetic datasets with a larger number of geometries. Additionally, we obtain the range_queries dataset from previous work [34] which includes a collection of synthetic datasets with different spatial distributions and associated with the results of range queries and their selectivity. (ii) Collections that contain real datasets, i.e., type = real, namely *weather* and *walkability*, which are obtained from [155] and Open Street Map (OSM), respectively.

Table 4.2: Dataset Collections Summary: "syn" and "real" states for synthetic and real datasets, respectively, while "SDS" means spatial data synopsis, "CL" is spatial clustering, "SEL" is selectivity and "WK" is walkability.

Collection	Type	Problem	# Sets	Min Set Size	Max Set Size
diagonal	syn	SDS	675	10000	20000
gaussian	syn	SDS	300	10000	20000
sierpinski	syn	SDS	300	10000	20000
uniform	syn	SDS	300	10000	20000
diagonal_large	syn	SDS	108	25000	100000
gaussian_large	syn	SDS	75	25000	75000
sierpinski_large	syn	SDS	100	25000	100000
uniform_large	syn	SDS	100	25000	100000
bit	syn	SDS	102	10000	20000
parcel	syn	SDS	108	10000	20000
range_queries	syn	SEL	8000	10240	29974
weather	real	SDS, CL	1568	1025	11613
walkability	real	WK	20000	2	27999

All of these datasets are preprocessed following our three representations described in Section 4.2. Some collections are only used in evaluation, while for the collections that are used in training, we keep a 20% subset for testing.

The following subsections describe in detail the obtained results for each considered problem.

4.4.2 Data Synopsis

The models for this problem are trained to predict nine different values, given a dataset D, as discussed earlier in Sec. 4.3.1, i.e., the maximum and minimum average value (hotspots), the location of hotspots, the K_{value}, and the box count values E_0 and E_2. We evaluate these models from different perspectives. First, we look into how they perform for different collections of datasets both those used in training and additional ones. The ones used in training include a separate 20% that is used for this evaluation. We summarize the results for this part for models GI-S, GG-S, and GV-S in Table 4.3. The last column is added to compare how the models perform to a random number generator, since no baseline is available in this case. The metric used in this table is the average of the weighted Mean Absolute Percentage Error of all the model outputs, where wMAPE = $(\sum_{i=1}^{n} |y_i - \hat{y}_i|)/(\sum_{i=1}^{n} |y_i|)$ and y_i is the actual value, while $\hat{y}_i$ is the predicted value. Note, that wMAPE is computed for each output value separately.

The first four datasets collections were seen in training. The GeoGraph model (GG-S) performs better for the seen datasets and their scaled versions, like *uniform* and *uniform_large*. There is only one case where it performs slightly worse than the other approaches. For the *parcel* distribution, GG-S performs much better for the hotspot-related values, but worse for some of the K_{values} and the box counts. However, we note that the actual K_{values} where GG-S performs worse are very close to zero, and small deviations from the actual value result in larger errors. All models, however, have much smaller errors

than the random predictions. In general, GG-S actually produces much better results for the K_{value} with the smallest radius. These results help give us confidence that the models actually learn the functions they were trained to predict, especially since some distributions were not seen in training, and most of the sets in the evaluation have hotspot positions that were not seen in training. The results on the larger datasets show that the quality is still consistent even if we make inferences on data sizes that are a few times larger than those seen in training.

Table 4.3: Data Synopsis Summary by Collection (wMAPE)

Collection	GI-S	GG-S	GV-S	Random
Collections seen in training				
diagonal	0.1059	**0.0660**	0.0801	1.5825
gaussian	0.7096	**0.1377**	0.1660	2.3565
sierpinski	0.1271	**0.1133**	0.1235	2.7929
uniform	0.6664	**0.1932**	0.2632	10.2451
Collections not seen in training				
bit	0.3271	**0.1831**	0.3026	1.4528
parcel	0.2381	0.4653	**0.2261**	2.7672
weather	0.4323	**0.3326**	0.4937	0.8745
diagonal_large	0.1091	**0.0638**	0.0788	1.6031
sierpinski_large	0.1177	**0.1075**	0.1134	2.9641
gaussian_large	0.5805	**0.1243**	0.1633	2.2329
uniform_large	0.6529	**0.1905**	0.2979	9.7232

Next, we evaluate the results by output type and summarize the results in Table 4.4. For this evaluation, we also use wMAPE, but this time it is computed for each output value for all datasets across all collections. The table shows that GG-S performs better for hotspot prediction, similar performance for K_{values}, and worse for box counts. It is not surprising that the image-based model works better for box counts since the first step for computing the box counts is creating a histogram, where a similar histogram is provided as input to the model. However, the GeoImg model performs noticeably worse when it predicts K_{value} with the smallest radius. This is most likely due to the fact that

Table 4.4: Data Synopsis Summary by Output (wMAPE)

Attribute value	Params	GI-S	GG-S	GV-S	Random
hotspots min-value	$k = 16$	0.20	**0.12**	0.28	0.84
hotspots x_{min}	$k = 16$	0.25	**0.12**	0.13	0.60
hotspots y_{min}	$k = 16$	0.33	**0.16**	0.21	0.68
hotspots max-value	$k = 16$	0.05	**0.04**	0.11	0.31
hotspots x_{max}	$k = 16$	0.25	**0.18**	0.25	0.74
hotspots y_{max}	$k = 16$	0.19	**0.14**	0.18	0.72
hotspots min-value	$k = 32$	0.21	**0.13**	0.21	0.80
hotspots x_{min}	$k = 32$	0.25	**0.13**	**0.13**	0.59
hotspots y_{min}	$k = 32$	0.32	0.18	**0.17**	0.67
hotspots max-value	$k = 32$	0.05	**0.03**	0.12	0.31
hotspots x_{max}	$k = 32$	0.25	**0.18**	0.26	0.78
hotspots y_{max}	$k = 32$	0.17	**0.14**	0.16	0.71
hotspots min-value	$k = 64$	0.21	**0.13**	0.18	0.78
hotspots x_{min}	$k = 64$	0.25	**0.12**	0.15	0.60
hotspots y_{min}	$k = 64$	0.32	**0.18**	**0.18**	0.69
hotspots max-value	$k = 64$	0.04	**0.04**	0.12	0.32
hotspots x_{max}	$k = 64$	0.25	**0.17**	0.30	0.75
hotspots y_{max}	$k = 64$	0.16	**0.15**	0.17	0.69
K_{value}	$r = 0.025$	1.88	0.91	**0.76**	16.30
K_{value}	$r = 0.05$	**0.56**	0.60	0.68	5.30
K_{value}	$r = 0.1$	**0.36**	0.50	62	1.69
K_{value}	$r = 0.25$	0.34	**0.15**	0.42	0.60
box counts E_0	–	**0.03**	0.05	0.07	0.22
box counts E_2	–	**0.11**	0.20	0.33	0.28

each pixel in the input histogram covers space that is multiple times larger than the radius. This result is consistent with our expectations.

Moreover, we perform experiments on real-world datasets, specifically, the *weather* collection. For this collection, we evaluated six models, three models trained on synthetic data, GI-S, GG-S, and GV-S, and three models trained on weather data, GI-S-W, GG-S-W, and GV-S-W. The results are shown in Table 4.5. The results are consistent with our previous observations.

Furthermore, we evaluate the running time for estimating these values, using our models, relative to computing them by using traditional algorithms. This is summarized in Table 4.6. The prediction time by the GeoImg model remains small even for large sizes since

Table 4.5: Data Synopsis for Weather Data Outputs (wMAPE)

Attribute value	Params	GI-S	GG-S	GV-S	GI-S-W	GG-S-W	GV-S-W	Random
hotspots min-value	$k = 16$	0.40	**0.22**	0.52	**0.07**	**0.07**	0.12	0.55
hotspots x_{min}	$k = 16$	0.46	0.22	**0.15**	0.10	**0.08**	0.14	0.45
hotspots y_{min}	$k = 16$	0.72	**0.33**	0.41	0.16	**0.12**	0.26	0.72
hotspots max-value	$k = 16$	0.07	**0.05**	0.25	**0.02**	**0.02**	0.03	0.28
hotspots x_{max}	$k = 16$	0.61	**0.46**	0.60	0.27	**0.24**	0.34	0.70
hotspots y_{max}	$k = 16$	0.27	0.27	**0.24**	0.17	**0.15**	0.20	0.52
hotspots min-value	$k = 32$	0.40	**0.24**	0.34	**0.04**	**0.04**	0.07	0.54
hotspots x_{min}	$k = 32$	0.47	0.23	**0.14**	**0.10**	**0.09**	0.14	0.48
hotspots y_{min}	$k = 32$	0.70	0.39	**0.34**	0.14	**0.09**	0.23	0.67
hotspots max-value	$k = 32$	0.07	**0.04**	0.32	**0.01**	**0.01**	0.02	0.26
hotspots x_{max}	$k = 32$	0.66	**0.48**	0.72	0.27	**0.25**	0.31	0.76
hotspots y_{max}	$k = 32$	**0.21**	0.25	0.26	0.12	**0.12**	0.16	0.52
hotspots min-value	$k = 64$	0.41	**0.24**	0.28	**0.07**	0.08	**0.07**	0.51
hotspots x_{min}	$k = 64$	0.48	0.23	**0.17**	0.11	**0.09**	0.14	0.48
hotspots y_{min}	$k = 64$	0.67	0.39	**0.35**	0.14	**0.11**	0.20	0.66
hotspots max-value	$k = 63$	0.08	**0.07**	0.33	**0.02**	**0.02**	**0.02**	0.26
hotspots x_{max}	$k = 64$	0.67	**0.49**	0.95	0.28	**0.24**	0.29	0.79
hotspots x_{max}	$k = 64$	**0.18**	0.30	0.32	**0.11**	**0.11**	0.15	0.45
K_{value}	$r = 0.025$	1.24	**0.60**	0.85	2.47	**0.46**	1.16	8.05
K_{value}	$r = 0.05$	**0.30**	0.76	0.96	0.43	**0.18**	0.30	2.14
K_{value}	$r = 0.01$	**0.49**	0.74	0.95	0.23	**0.17**	0.28	0.57
K_{value}	$r = 0.25$	0.44	**0.24**	0.91	0.17	**0.05**	0.09	0.47
box counts E_0	–	**0.05**	0.12	0.19	**0.02**	**0.02**	0.03	0.20
box counts E_2	–	**0.30**	0.64	1.32	**0.01**	**0.01**	0.02	0.49

the input size is fixed; however, the time to prepare the histogram increases with increasing data sizes, even if it still scales well. Similarly, GV-S scales well, since its input is bounded in size.

The running time for the GeoGraph model can increase significantly with increasing the dataset size, but it is still much lower than the time it takes to compute the target values with traditional algorithms. We also evaluate how GG-S scales with increasing the dataset size. Some results are shown in Figure 4.7. We start with 10 thousand points and keep doubling the size. We get a memory access error once we reach the size $2^8 * 10^4$ points. The memory consumption increases linearly, while the inference time increases sub-quadratically. The main effect is caused by the number of edges required for each

message-passing step. Another observation we make is that the GeoGraph model converges in a smaller number of epochs compared to the others, as shown in Figure 4.8.

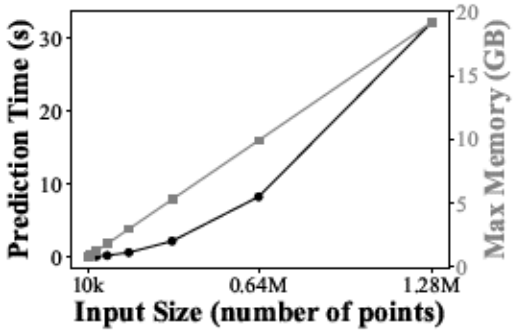


Figure 4.7: GG-S time and memory vs. input size

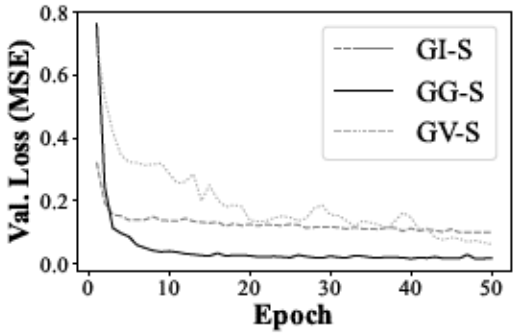


Figure 4.8: Data Synthesis Validation Loss by Epoch

Table 4.6: Execution time by distribution in seconds

Collection	GI-S		GG-S	GV-S	Exact value
	Inference	+Histogram			
uniform	0.02	1.61	2.37	0.23	199.54
diagonal	0.02	3.53	5.02	0.49	521.88
gaussian	0.01	1.55	2.38	0.22	338.21
sierpinski	0.01	1.61	3.02	0.22	178.17
weather	0.04	4.49	3.39	1.16	2051.31
bit	0.02	2.75	4.25	0.38	542.20
parcel	0.01	2.83	4.35	0.39	398.92
uniform_large	0.01	11.25	204.86	0.37	2796.00
gaussian_large	0.01	10.93	202.96	0.37	6749.55
diagonal_large	0.01	11.83	219.14	0.39	1756.75
sierpinski_large	0.01	10.63	205.47	0.36	1659.95

4.4.3 Clustering

We compare the clusters estimated by models with the clusters discovered by ST-DBSCAN, which is what the model was trained to predict. We used three scores for this evaluation: (i) A perfect homogeneity score indicates that all points within a predicted cluster are within the same cluster in the ground truth. (ii) Completeness has a perfect score

of 1.0 if all points in a single cluster in the ground truth are predicted to be within the same cluster. (iii) The V-measure is an average of the other two scores. For more information on these scores, see [177]. The results of this evaluation are shown in Table 4.7. The best clustering was produced by GG-C, the non-parametrized GeoGraph model. Generally, the models perform well in all cases except two. The two cases are the ones with *MinPoints* equal to 5. It also happens that these parameters result in a larger number of clusters. We also evaluate the running time, and computing the estimated results in both cases is much lower compared to computing the ground truth with ST-DBSCAN.

Table 4.7: Evaluation of estimated clusters

Model	**Param.**	**Homogeneity**	**Completeness**	**V-Measure**
GI-C	(0.05, 200, 50)	0.86	0.94	0.90
GI-CP	(0.02, 200.0, 5.0)	0.85	0.67	0.74
GI-CP	(0.03, 50.0, 5.0)	0.81	0.69	0.74
GI-CP	(0.03, 200.0, 20.0)	0.83	0.79	0.81
GI-CP	(0.05, 100.0, 50.0)	0.83	0.87	0.85
GI-CP	(0.05, 200, 50)	0.85	0.89	0.87
GI-CP	Average	0.83	0.78	0.80
GG-C	(0.05, 200, 50)	0.95	0.95	0.95
GG-CP	(0.02, 200.0, 5.0)	0.79	0.71	0.74
GG-CP	(0.03, 50.0, 5.0)	0.73	0.68	0.70
GG-CP	(0.03, 200.0, 20.0)	0.80	0.81	0.80
GG-CP	(0.05, 100.0, 50.0)	0.85	0.87	0.86
GG-CP	(0.05, 200, 50)	0.85	0.88	0.86
GG-CP	Average	0.80	0.79	0.79
GV-C	(0.05, 200, 50)	0.90	0.92	0.90
GV-CP	(0.02, 200.0, 5.0)	0.55	0.4	0.45
GV-CP	(0.03, 50.0, 5.0)	0.51	0.35	0.4
GV-CP	(0.03, 200.0, 20.0)	0.75	0.67	0.7
GV-CP	(0.05, 100.0, 50.0)	0.82	0.81	0.81
GV-CP	(0.05, 200, 50)	0.83	0.81	0.82
GV-CP	Average	0.69	0.61	0.64

4.4.4 Selectivity Estimation

We summarize the results in Table 4.8. We notice that the GeoGraph approach performs better in most cases, from very low selectivity to higher selectivity. However,

notice that we only used the smaller data sizes for this problem, where each set is only tens of thousands of points, whereas in our reference [34] much larger sizes were considered. Also, notice that most of the examples have very low selectivity, due to the range queries associated with the datasets.

Table 4.8: Selectivity estimation (wMAPE)

Selectivity Range	GI-E	GG-E	GV-E	Random
[0.0, 1e-05]	3.34E+03	**5.33E+02**	1.87E+03	4.31E+05
(1e-05, 0.0001]	1.83E+02	**3.72E+01**	8.42E+01	7.92E+03
(0.0001, 0.001]	1.31E+01	**3.81E+00**	9.51E+00	1.04E+03
(0.001, 0.01]	3.52E+00	**1.5E+00**	2.35E+00	1.41E+02
(0.01, 0.1]	**4.32E-01**	5.17E-01	4.86E-01	1.57E+01
(0.1, 1.0]	7.9E-01	**3.18E-01**	5.48E-01	2.36E+00

4.4.5 Walkability Estimation

We summarize the walkability results in Table 4.9. The table shows for each walkability range the percentage of scores that are predicted within 0.1 of their actual value. From the table, GI-W seems to work much better than the other two, which show similar accuracy. We think this can be attributed to three aspects related to each method. First, in this data set, the data is divided by $5km \times 5km$ regions; this results in more sparse histograms and less overlap of points in each histogram pixel. Second, the decrease in accuracy in GG-W could be due to the way the dataset join is performed. While we experimented with several variations in performing the join, there is still more room for improvement. Third, in GV-W, the significant decrease in accuracy could be due to the sampling. However, all three cases still produce reasonably good results and are much faster than the exact method.

Table 4.9: Summary of accuracy for walkability results

Label	Model	[0.0, 0.1]	(0.1, 0.2]	(0.2, 0.3]	(0.3, 0.4]	(0.4, 0.5]	(0.5, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1.0]	Avg
California	GI-W	86%	89%	97%	95%	87%	85%	81%	79%	100%	–	88%
	GG-W	92%	88%	69%	68%	67%	66%	78%	79%	68%	–	77%
	GV-W	94%	86%	85%	84%	64%	65%	78%	61%	56%	–	75%
Florida	GI-W	100%	75%	97%	97%	78%	74%	100%	–	–	–	89%
	GG-W	96%	98%	81%	91%	57%	45%	57%	–	–	–	80%
	GV-W	97%	70%	84%	88%	61%	55%	43%	–	–	–	71%
Italy	GI-W	96%	94%	94%	90%	83%	80%	79%	83%	99%	–	89%
	GG-W	88%	86%	82%	82%	77%	76%	72%	78%	69%	–	78%
	GV-W	97%	89%	87%	84%	51%	68%	72%	67%	67%	–	76%
New York	GI-W	100%	100%	95%	95%	82%	100%	0%	83%	100%	100%	96%
	GG-W	92%	81%	91%	71%	82%	71%	50%	88%	94%	12%	51%
	GV-W	98%	93%	64%	86%	71%	100%	0%	51%	0%	96%	66%
United Kingdom	GI-W	100%	95%	93%	92%	85%	80%	91%	98%	100%	100%	93%
	GG-W	89%	93%	88%	91%	72%	59%	69%	88%	80%	21%	72%
	GV-W	96%	95%	84%	84%	56%	62%	75%	46%	47%	99%	74%

Note: each column represents regions within a walkability range, and scores represent the percentage of regions that received highly accurate estimates.

4.4.6 Discussion

In this part, we discuss the main takeaways from these results. From the data synopsis results, we notice that position-based predictions and predictions that require analyzing points at very small distances, like when the K_{value} is very small, GeoGraph is more suitable. For values that only require a global overview of the data, and where localized values are less relevant, like when computing the box counts, GeoImg is more suitable.

From the clustering results, we notice that all types of models generally work well and estimate clusters efficiently. However, the model may not be accurate if the number of expected clusters is large, especially if the model is parameterized. The results for this problem serve as a motivation for other problems like spatial interpolation, and we expect that GeoGraph would be more suitable.

From the selectivity results, we still notice that there are issues related to scalability that need to be addressed for GeoGraph, especially when data sizes grow to millions of points. Several directions can be investigated, such as partitioning, indexing, and sorting-based methods. We leave this for future work. We also notice that better representations

for geometries other than points are needed. However, a simple representation, such as using the centroid point and pairing it with some features, is sufficient for many use cases.

From the walkability results, we notice that joining multiple datasets under GeoGraph still needs some improvements. Joining datasets under GeoImg is straightforward, since we just align the histograms on top of each other. It worked relatively well in our case, since the area is small ($5km \times 5km$), and there is not much overlap on each pixel, and there is a lot of sparsity. However, in other cases, GeoImg may not be suitable, and a more accurate GeoGraph based approach will be needed. For GeoVec, better sampling or partition approaches can be considered. For example, instead of just taking samples prior to feeding to the model, some aggregations can be used. Furthermore, some hybrid approaches can also be considered. For example, a set abstraction layer can be used to summarize the points in each region, and then embeddings can be passed to a different model architecture. We leave it for future work.

4.5 Related Work

Deep learning for geospatial data. The use of deep learning architectures with geospatial data is improving. Recently, libraries like TorchGeo [193] have been introduced, as well as geospatial foundation models like [153, 13, 205], and others, refer to [238] for a recent survey. However, all of these are based on vision models and tailored for geospatial raster data, which is a different modality than geospatial vector data.

Although there have been successful applications of deep learning with geospatial vector data, full support is still lacking. There have been several proposals for better

representing geospatial vector data for deep learning architectures. These include [147, 222, 149, 188] which all focused on representing a single object, either a point or a polygon, and possibly encoding the context around the object. However, this is very limiting, since a lot of geospatial problems require analyzing a huge set of geometries, such as the problems we consider in this work. We integrated Poly2Vec [188] with the GeoVec architecture, and it is a promising approach. The work in [146] provides a survey on different spatial representations, including models that work with point sets. To the best of our knowledge, this is the first work that studies point set models for geospatial applications, analyzing their effectiveness for various spatial operations and learning about existing limitations.

Point-set architectures. Several architectures have been proposed that take a set of points as input, and they are mainly related to the points-of-cloud domain, where a set of points represents three-dimensional objects. We chose to study PointNet++ [169] due to its suitability for the problems that we consider and the fact that it follows a multi-scale hierarchical approach. While we focused on this architecture, we keep our discussions general enough in a way where it is possible to replace different layer types since this might produce higher quality results. However, our goal is not to provide the best accuracy possible but to learn about the suitability of point-set-based approaches for problems that involve geospatial vector data. Point Transformer [233] is one alternative, and its more recent version PTv3 [219] provides more efficient aggregation by using space-filling curves. Anyway, it does not support processing attributes associated with points, a requirement for all our problems. Also, these new aggregations are not yet integrated with PyTorch

Geometric [68] which is the library of choice. There are also other alternatives, and [146] provides a good comparison among them.

Data synopsis. The K_{value} metric is based on Ripley's K function, used in traffic flow, neighborhood accessibility, clustering, and hotspot detection [101, 135, 199, 90, 141]. To address scalability, Spark-, GPU-, and heuristic-based solutions exist [216, 198, 48]. Box counts [33] have been used in query optimization [209]. Other applications of synopsis include similarity search [224]. Metric selection is application-driven, e.g. [181] develops custom metrics, and we only considered a small subset. **Clustering.** It has a very wide range of applications [18, 46, 49, 187]. There has been work on clustering with deep learning [202, 20, 157, 104]. It involves different architectures, custom clustering loss functions, and other steps. We focused on supervised learning on showing how the different representations can replicate the operations of existing algorithms. We leave the unsupervised learning for geospatial clustering for future work. **Selectivity.** Selectivity estimation is important for query optimization; some related DL-based methods include [84, 34], and [231] provides a survey. **Walkability.** We study walkability because it includes spatial joins. There are many ways to measure walkability [207], and often it involves OpenStreetMap data [83, 133]. Spatial joins are expensive [236, 72, 218, 209]; our aim is to evaluate the effectiveness of different representations in capturing an operation with a spatial join rather than advancing walkability estimation.

4.6 Conclusion

We have presented three different approaches GeoImg, GeoGraph, and GeoVec for working with geospatial vector data and deep learning. We modeled four geospatial problems following the three approaches and learned about their effectiveness and limitations. The results show that all enable modeling complex geospatial operations. Future research directions include improving scalability, designing more generalized models and applications, and enabling models that work with multiple modalities of geospatial data. Which will all take us closer to realizing the use cases that we envision.

Chapter 5

GS-QA: A Benchmark for Geospatial Question Answering

5.1 Introduction

Question Answering (QA) systems are designed to answer free-form questions. Earlier QA works focused on questions based on a given text passage [53], while more recent work focuses on open-domain QA, where a large collection of documents or other data must be searched to find the answer. Recent advancements have shown that Large Language Models (LLMs) are excellent for synthesizing text responses given complex questions, and can be used to build more robust QA systems. Recent work has proposed benchmarks for the evaluation of the performance of LLMs in QA systems [25, 114, 52, 184].

In the area of geospatial data management, QA has the potential to disrupt the way that people look for geospatial information, given the complexity of querying geospatial data for non-experts. As an example, consider the question `'Which four star hotels are within 50km of UCR towards LAX?'` To answer such a question, first, the anchoring locations must be identified, which are the Univesity of California, Riverside (`'UCR'`) and Los Angles International Airport (`'LAX'`). Then, their location coordinates must be retrieved. After that, the spatial predicates must be identified, which are a range query within a 50 kilometers radius and a direction filter based on an angle. Finally, hotels that are in locations that satisfy the spatial predicates are retrieved, as shown in Figure 5.1.

Surprisingly, state-of-the-art conversational AI tools like GPT-4 are unable to correctly answer such questions, because they do not account for all the spatial predicates. For example, passing the example question to ChatGPT returns popular hotels in Riverside and some nearby cities that are not necessarily in the search area; that is, the directional predicate is ignored. Changing the question by replacing LAX with Las Vegas, which is towards the opposite direction, still produces the same answer.

Designing a system for answering such questions is challenging. First, understanding spatial predicates and performing spatial reasoning are required. Current LLMs often respond with popular entities, e.g., the names of popular hotels, which may not necessarily be the best answer. Second, the answers to some questions can change over time, so the LLM may not be up-to-date. Third, some questions may require synthesizing data from multiple sources, which may include unstructured data sources. Building an effective

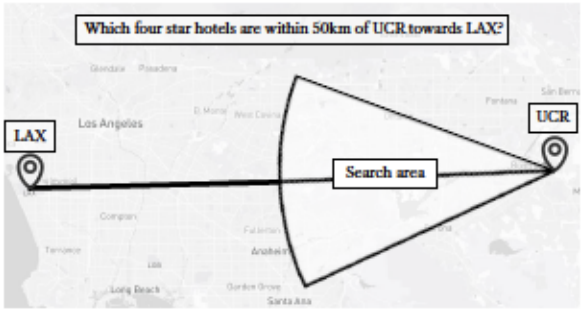


Figure 5.1: Geospatial Question Answering Example

geospatial QA system could enable users and practitioners from many fields to get reliable answers without relying on specialized tools or languages.

There is limited work on benchmarking Geospatial Question Answering (GeoQA) [108, 168, 103]. These have various limitations. First, they include a small number of questions and no mechanism to generate more questions of given types. Second, they assume the existence of a spatial knowledge graph, which is often not available in practice, or it may only store part of the necessary information to answer a question. Finally, they have limited spatial operators or non-spatial conditions.

Evaluating GeoQA is challenging for various reasons. Geospatial data represent many different types of entities, such as points representing restaurants, lines for roads, polygons for region borders, etc. There are many types of questions that can be asked about this type of data, such as asking about directions, the location of an object that satisfies some conditions, and more, including questions that require aggregation and analysis of a large number of records. Further, the generated question-answer pairs must be unambiguous. A question must have a unique answer that can be computed deterministically given reference data. In addition, we need a clear process for evaluating the correctness and quality of the

generated question-answer pairs. Furthermore, we want a benchmark that can be easily updated or expanded given reference data.

In this work, we introduce GS-QA, a benchmark for evaluating open-ended answers on questions that involve geospatial data. It includes 28 question templates, incorporating a variety of spatial objects, spatial predicates, and output types, among others. It also includes multi-source (i.e., multi-hop) questions that require multiple steps to be answered, synthesizing information from multiple spatial and non-spatial sources. We provide one hundred questions for each template, for a total of 2800 questions. We also propose various evaluation strategies appropriate for GeoQA, which go beyond traditional text-based matching used in existing QA work. We complement text-based matching with spatial-based measures, such as relative distance error. We created and complemented a suite of six diverse LLM-based GeoQA baselines, which combine LLMs, retrieval, and structured querying (i.e., text-to-SQL).

In summary, our contributions are as follows:

- We create a GS-QA, a benchmark dataset comprising 2800 questions from 28 templates, with a wide range of spatial objects, predicates, and non-spatial information. We have published GS-QA along with the baselines' results [150].
- We develop a methodology for generating a large number of question-answer pairs based on our templated and a reference database, using corresponding SQL query templates.
- We implement several LLM-based GeoQA baselines, based on state-of-the-art retrieval methods.

- We propose a suite of text-based and geospatial-specific evaluation measures.
- We perform extensive experiments that show that existing baselines have low performance for most of the query templates of GS-QA.

We start by summarizing the related work in the literature and how GS-QA fills a needed gap in Section 5.2. In Section 5.3, we present our methodology used to create the benchmark. In Section 5.4, we discuss the baselines that we use to demonstrate how to use the benchmark. In Section 5.5, we provide experimental results evaluating the baselines. Finally, in Section 5.6 we discuss future work and conclude.

5.2 Related Work

GeoQA Datasets. Existing benchmarks for open-domain GeoQA include GeoQuestions1089 [108] and GeoQuestions201 [168]. Both include natural questions with associated queries that get the answer from a reference knowledge graph, namely, YAGO2geo [103] and one of its variants [40] in the case of GeoQuestions1089. We propose GA-QA over an existing benchmarks such as GeoQuestions1089 for multiple reasons. First, they assume the existence of a knowledge graph, whereas GS-QA is an open-retrieval benchmark, i.e., it operates on top of heterogeneous data sources, including structured spatial databases and document stores. Second, all GS-QA questions reference multiple entities, including the question itself and when computing the answer. This is to ensure that the system is evaluated based on performing spatial operations and not simple retrieval based on string matching. Furthermore, our proposed benchmark includes a larger number of question categories and each category has a larger variety of questions. Also, in being able to automate

the question generation process, our benchmark can be easily extended to include more question categories. These differences are summarized in Table 5.1.

Table 5.1: Comparison between our benchmark and GeoQA1089

Benchmark	**Proposed benchmark**	**GeoQA1089**
Main reference data	OpenStreetMap	Geospatial knowledge graphs
Query language	SQL (PostGIS)	GeoSPARQL
Includes open retrieval	Yes	No
Question categories	28	9
Number of questions	2800	1089
Creation method	Automated template-based	Manual
Easily extensible	Yes	No

GeoQA Systems. Existing systems for GeoQA [51, 124, 108] focus on question answering on a knowledge graph by converting the user question into a dependency parse tree that extracts the final answer from the knowledge graph. Our system is more versatile by utilizing the power of LLM. It can convert the user question into queries that run on any data store, e.g., a database, and use the query result to formulate the final answer.

Closed-domain Spatial Reasoning. There has been work on evaluating spatial reasoning, such as SPARTQA [158]. The problem here is: given a description (e.g. "We have three blocks, A, B, and C. Block B is to the right of block C and it is below block A.") we ask a question (e.g. "Which object is above a medium black square?") that requires spatial reasoning on the provided description. This work is different from our problem, which is open-domain question answering. There is also work on visual spatial reasoning [138], where the input is an image instead of a textual description. Similar works also fall under the domain of spatial reasoning and spatial proximity, including [50, 125]. The work in [107] evaluates spatial reasoning in the context of GeoQA.

Benchmarks for LLM-based QA Evaluation. With the rapid and wide spread of LLMs, researchers have developed several benchmarks to evaluate LLMs on question answering. Some benchmarks rely on multiple choice questions, e.g., MMLU [86] and GPQA [175]. These benchmarks are more challenging and they require deep domain knowledge. The proposed benchmark has more open-ended answers and we take that into account when designing the evaluation strategy as detailed in the paper.

There are also domain-specific benchmarks such as MATH-500 [136] which evaluates LLMs on solving math problems in a step-by-step approach. In this case, all the steps are used to evaluate the model and not just the final answer. Our benchmark focuses on spatial QA and it evaluates only the final answer. We believe that step-by-step evaluation can be useful but we leave this for future work. In this case, the answer will need to change to steps that resemble how GIS analysts approach the question. Another difference is that the MATH-500 dataset was used for both training and evaluation while this paper focuses on evaluation. Our QA dataset generator can also be used to generate data for training but we leave this also for future work.

The evaluation of LLMs is very challenging, and the process is not yet standardized. The work in [117] discusses the challenges and strategies for LLM evaluation. Some of the challenges include contamination of benchmark data in training, tailoring prompts to include examples very similar to the questions being asked, and reproducibility issues. This informs how we designed our baselines.

Multi-hop questions. These types of questions require multiple steps in order to be answered, because they typically involve fact-chaining. There existing datasets for these

types of questions including [225, 196, 89, 204], and [170] provides a survey on this topic. The main difference in our benchmark is that all the facts are related to a spatial object, either identifying a spatial object by its that is referenced in the question or first answering a question by retrieving a spatial object and then retrieving an external fact related to it.

Multi-modal question answering. These types of questions involve working with multiple modalities, such as text and images. Some existing benchmarks for multi-modal question answering include [183, 197]. Our benchmark also falls under this category, since we work text and geospatial data. All our questions require reasoning over text, including retrieving information from text documents, as well as perform some geospatial analysis.

Text2SQL. Converting a natural language question to an SQL query is another active area of research. In the literature, this problem is usually referred to as Text2SQL or NL2SQL. Several methods have been proposed, including LLM-based approaches [121]. There exist several benchmarks for this problem, such as [237, 126], but they are not focused on geospatial queries. Our work can serve as a reference point for the Text2SQL for geospatial data, as well as for building a scalable training dataset.

5.3 GS-QA Benchmark Creation

This section outlines our methodology for creating the GS-QA benchmark's reference data, questions, and answers. First, Section 5.3.1 explains how we extract and prepare geospatial **reference data** from OpenStreetMap. Then, Section 5.3.2 discusses the creation of **question templates** by selecting predicates, spatial entities, and answer types. After

that, Section 5.3.4 describes how to **instantiate these templates** to generate questions. Section 5.3.5 explains the process of **verifying question quality** and answer correctness to ensure question diversity. Finally, Section 5.3.6 discusses how to **keep the benchmark up-to-date** as the reference data is updated.

5.3.1 Reference Database

The first step in creating the benchmark is to prepare the reference dataset that the LLM will use to answer all questions. Our goal is to create a large number of questions that are primarily focused on spatial objects and spatial operations. However, there are four challenges in doing that, as mentioned earlier in the introduction. These include ensuring the questions have a large variety, there is a standardized method to get the correct answers, scalability, and handling data updates. The reference database we create is a structured database instance that we generate from OpenStreetMap (OSM). Building a reference database with a known structure helps us with these challenges. First, it provides us with a clear structure about the type of spatial objects available and the attributes associated with them. This makes it more straightforward to create a diverse set of question templates. Secondly, the support of standardized SQL queries by the database makes it possible to create queries that instantiate the question templates, as well as define their answers. Furthermore, by having standardized queries, it is possible to generate question/answer pairs on a scale. Additionally, there are also advantages in relation to data updates and handling data from multiple sources in the future. The database can also serve as a reference in RAG (Retrieval Augmented Generation) systems.

We get our source data from OSM. We use it as our source because it is the largest publicly available geospatial dataset. However, our methodology can be used on any source data. We don't use OSM directly since it is in a semi-structured format, and converting it to a relational database provides all the benefits mentioned earlier. We obtain the most recent version of OpenStreetMap for the entire United States of America from GeoFabrik OpenStreetMap Extracts [77], which are updated daily. We use the extract produced on February 2, 2024. We show in Section 5.3.6 how to update the generated question-answer pairs using more recent versions.

Then, we use OSMX [191] to extract five different datasets. 1) **Points of interest (POI)**, which includes points for restaurants, shops, hospitals, and many more. 2) **Administrative boundaries**, which contains boundaries for states, counties, cities, etc., at different administrative levels. 3) **Parks** contains boundaries for recreational parks, nature reserves, and sports stadiums, among others. 4) **Water bodies** which includes lakes, rivers, streams, and others. 5) **Roads and walkways** includes roads for cars and pedestrians. Readers can refer to [191] for more details on the extraction process.

Each object in OSM has a geometry attribute, which is the main attribute relevant to creating our questions. Additionally, each object has a set of tags which add more information to each object, e.g., park name or speed limit. Since tags are user-defined, they can be very arbitrary and it would be difficult to store all of them as separate attributes. Instead, we identified some common attributes that are useful for question answering, such as name, a reference to a Wikipedia page, and address. Then, we selected a small subset of other attributes that are related to specific categories like points of interest that provide

Table 5.2: Reference database summary

Table	Columns/Attributes	Geometries	Records
POI	Geometry, OSM ID, Name, Wikipedia, Address, Leisure, Amenity, Tourism, Emergency, Restaurant attributes.	Points	267612
Park	Geometry, OSM ID, Name, Wikipedia, Address, Leisure.	All types	5997948
Lake	Geometry, OSM ID, Name, Wikipedia, Address, Water, Waterway.	All types	7988851
Road	Geometry, OSM ID, Name, Wikipedia, Address, Highway.	All types	36827649
Region	Geometry, OSM ID, Name, Wikipedia ID, Address, Border type, Adminstration level.	LineStrings Polygon	39137

services, or are related to tourism, etc. These columns provide us with enough variety to create a large set of natural questions. We summarize the selected attributes for all tables in the database in Table 5.2. Refer to [163] for details about different attributes and their possible values. Using the schema (columns) in Table 5.2, we create a database instance using PostgreSQL with PostGIS. The statistics of our reference database are shown in Table 5.2.

The name attribute is used to refer to entities in our questions-answer pairs. The columns related to Wikipedia are used to retrieve information related to question entities to create multi-hop questions that contain information from an additional information source. This can provide another dimension for evaluating question answering systems.

The addresses are stored in multiple columns and are used for disambiguation, since an object's name may exist in many locations. Some address attributes like state and county are appended to names to make them less ambiguous. The address attributes are also used as an additional indicator of the importance of a POI, since many records

do not have an address defined, although it is relatively straightforward to translate the coordinates to an address, but it may not always be accurate.

The *amenity* column is associated with points of interest that provide everyday services, like restaurants and coffee shops. Although there are many types of amenities, we selected only a few types that would create natural-sounding questions and still provide a wide variety of values. We only considered amenities related to sustenance such as restaurants and coffee shops, as well as hospitals and universities when we created the questions. When the value for this column is restaurant, other columns can also be used to add additional non-spatial descriptors such as the type of cuisine, availability of drive-through, or outdoor seating. Later, this helps to add more variety to the questions.

From the tourism column, we selected ten categories, including hotels, museums, theme parks, art galleries, etc. The museum column is used to add a non-spatial attribute to the questions when the value in the tourism column is a museum.

In the leisure column, we consider values like park, beach resort, golf course, nature reserve, sports center, among others. For water bodies, we consider lakes, bays, rivers, streams, and a few others. For roads, we consider primary, secondary, residential, and more.

5.3.2 Question Templates

To create a large number of questions that can efficiently benchmark a spatial QA system, we first create *question templates*, and then use these templates to instantiate an arbitrarily large number of questions. Table 5.3 lists all the 28 templates that we use. Each template has one or more *variables* or placeholders that are substituted from the

database to create a question instance. Table 5.4 lists all these variables and how they get substituted. For example, template **T1** can be instantiated into the question '*Can you suggest a* **restaurant** *within* **1 km** *from* **San Diego Zoo***?*'.

A question template is defined by (a) one or more predicates (e.g., "nearest neighbor"), (b) one or more spatial anchor entities (e.g., "Yosemite National Park"), (c) output type (e.g., "location"), and (d) a set of text phrases that can be used to instantiate a question.

Additionally, for each question template, we create an associated SQL query template that computes the answer to the question when applied to the reference database.

The templates are based on a variety of combinations of predicates and output types. The table shows only one text example for each template, but each template has several text phrases to choose from, adding to the richness of the generated questions. They are based on a subset of spatial predicates, metrics, and aggregate functions from all the operations supported by PostGIS. It would be intractable to include all combinations of operations and functions of PostGIS in the benchmark. Instead, we focused on a smaller subset of combinations that can be easily translated to natural language questions that can be asked by everyday users. We note that even this smaller subset is already very challenging to answer using existing LLM based tools, as will be shown later. However, based on the question generation process that we define here. The benchmark can be easily extended to support more operations and variety of questions in later iterations.

Furthermore, for some of these templates, specifically T1, T5, T13, and T17, we created four more templates that include an additional non-spatial predicate, specifically

Table 5.3: Question templates

ID	Text	Output Type	Spatial Predicates
T1	Can you suggest {POI_CAT} within {DISTANCE} from {ANCH_POI}?	Entity name	Range
T2*	Can you suggest {POI_NONSPAT} within {DISTANCE} from {ANCH_POI}?		Range
T3	Which {POI_CAT} is located within {DISTANCE} in the {DIRECTION} of {ANCH_POI}?		Range, Direction
T4	Which {POI_CAT} can I find within {DISTANCE} from {ANCH_POI1} towards {ANCH_POI2}?		Range, Towards
T5	What is the nearest {POI_CAT} from {ANCH_POI}?		Nearest Neighbor
T6*	What is the nearest {POI_NONSPAT} from {ANCH_POI}?		Nearest Neighbor
T7¶	What is the capacity of the nearest {POI_CAT} from {ANCH_POI}?		Nearest Neighbor
T8‡	What is the nearest {POI_CAT} from {ANCH_POI_external}?		Nearest Neighbor
T9	What is the closest {POI_CAT} {DIRECTION} of {ANCH_POI}?		Nearest Neighbor, Direction
T10	What is the closest {POI_CAT} from {ANCH_POI1} towards {ANCH_POI2}?		Nearest Neighbor, Towards
T11	What is the largest {PARK_WATB} in {REGION}?		Intersects
T12	What is the longest {ROAD_WATW} in {REGION}?	Location	Intersects
T13	Where can I find {POI_CAT} within {DISTANCE} from {ANCH_POI}?		Range
T14*	Where can I find {POI_CAT} within {DISTANCE} from {POI_NONSPAT}?		Range
T15	Where can I find {POI_CAT} located within {DISTANCE} in the {DIRECTION} of {ANCH_POI}?		Range, Direction
T16	What location has {POI_CAT} within {DISTANCE} from {ANCH_POI1} towards {ANCH_POI2}?		Range, Towards
T17	Where can I find the nearest {POI_CAT} from {ANCH_POI}?		Nearest Neighbor
T18*	Where can I find the nearest {POI_NONSPAT} from {ANCH_POI}?		Nearest Neighbor
T19	Where is the closest {POI_CAT} {DIRECTION} of {ANCH_POI}?		Nearest Neighbor, Direction
T20	Where is the closest {POI_CAT} from {ANCH_POI1} towards {ANCH_POI2}?		Nearest Neighbor, Towards
T21	In which direction is {POI_CAT} located within {DISTANCE} from {ANCH_POI}?	Angle	Range
T22	What is the direction towards the closest {POI_CAT} from {ANCH_POI}?		Nearest Neighbor
T23	How many {POI_CAT} within {DISTANCE} from {ANCH_POI}?	Count	Range
T24	How many {POI_CAT} are there in {REGION}?		Intersects
T25	How far can I find {POI_CAT} within {DISTANCE} from {ANCH_POI}?	Distance	Range
T26	How far is the closest {POI_CAT} from {ANCH_POI}?		Nearest Neighbor
T27	What is the total area of all {PARK_WATB} in {REGION}?	Area	Intersects
T28	What is the total length of all {ROAD_WATW} in {REGION}?	Length	Intersects

* Includes an additional non-spatial predicate, where POI_CAT is more specific.
¶ A multi-hop question that asks about information retrieved from another source.
‡ A multi-hop question where the ANCH_POI is replaced with unique information from an external source.

T2, T6, T14 and T18, respectively. These are based on some of the attributes we included in the reference database, such as the type of cuisine, the type of museum, etc. Additionally, we created two more templates from T5 as multi-hop questions, which are T7 and T8 that include information that does not exist in the reference database. For T7, the information the question is asking about does not exist in the reference database. Alternatively, for T8, the name of the anchor point in the question is replaced by unique information about it retrieved from another source. Both types of questions add another layer of difficulty in which the QA system must synthesize information from multiple sources to find the final answer. Next, we cover the building blocks of these templates in more detail.

Table 5.4: Question Parameters Summary

Parameter	Description
ANCH_POI	Name of anchoring POI from one of these categories: aquarium, attraction, viewpoint, art gallery, theme park, museum, gallery, zoo, hotel, university, park, nature reserve, garden, stadium, hospital
POI_CAT	POI category name: restaurant, café, fast food, plus all the categories in ANCH_POI.
POI_NONSPAT	A more specific POI category, like cuisine for a restaurant, or museum type.
REGION	Name of region from one of: city, town, village, island, municipality, county, neighborhood, suburb, state
ROAD_WATW	Name of road or waterway type, including: primary, residential, pedestrian, etc., for roads river, stream, and others for waterways.
PARK_WATB	Name of park or water body types, including: nature reserve, park, garden, golf course, etc., for parks lake, bay, etc., for water bodies
DISTANCE	Random distance in the range [1,200] kilometers.
DIRECTION	Direction one of: north, northeast, east, southeast, etc.

Question Parameters

***Anchor POI* (ANCH_POI):** This refers to a POI that is used to anchor the location of the search for the desired entity. The categories we selected for this type are more prominent POIs like attractions, museums, universities, and others. We avoided points that can exist repeatedly within a small geographic area, such as restaurants. To further reduce ambiguity, we attach the city and state to the name of the anchor point in a question.

***POI category* (POI_CAT):** This refers to the type of POI the question asks about. The answer to the question includes a non-spatial filter that selects points from this category. Any type of POI category can be used, but we require that it be from a different category than the anchoring point to increase the chance that the generated question is more similar to questions that could be asked by everyday users.

As an example, we show how the previous two parameters can be used to instantiate a template such as `'Where can I find the nearest {POI_CAT} from {ANCH_POI}?'` (T17 in Table 5.3). The output category could be instantiated with *restaurant* while the anchor point could be *Alaska Pacific University, Anchorage, AK*. Note that it is easy to also use other types of objects such as roads as the anchoring object in a question, using the same text phrase and the same query that obtains the answer.

***More specific POI category* (POI_NONSPAT):** This is similar to the previous variable, but it represents a more specific POI_CAT. For example, instead of just instantiating with *restaurant*, the question can be instantiated with something more specific like *restaurant with outdoor seating*. This is discussed in more detail later in this section.

Administrative regions **(REGIONS):** all the regions in the region table as described in subsection 5.3.1 are used in questions involving topological operations and aggregations.

Parks and water bodies **(PARK_WATB):** The parks and water bodies, like lakes, are used to instantiate questions that ask about areas. An example of such a template is `'What is the largest {PARK_WATB} in {REGION}?'` (T11 in Table 5.3).

Roads and water ways **(ROAD_WATW):** the different types of roads and waterways such as rivers are used to instantiate questions asking about length. For example, a template that includes these object is `'What is the total length of all {RD_WATW} in {REGION}?'` (T28 in Table 5.3).

These parameters are summarized in Table 5.4. These make up a major spatial component in the questions. The other major spatial component is the spatial filtering operations.

Spatial Predicates

All the questions that we generate contain at least one spatial predicate. In addition, the predicates determine what types of spatial objects are used. Next, we define the five spatial predicates that we included.

***Nearest Neighbor*:** this is the simplest spatial predicate, and it filters based on spatial proximity. It requires an anchoring point, such as the user's location, and provides the nearest object that satisfies other predicates in the question. An SQL query for a question that contains this predicate will include the following:

```
ORDER BY  geometry <-> anchor_point ASC LIMIT 1;
```

***Range*:** similar to the nearest-neighbor predicate, but also requires providing a distance limit. In this predicate, there might be multiple possible answers within the provided distance. The SQL will include something like:

```
ST_DWithin(geometry, anchor_point, distance)
```

***Direction*:** This adds another predicate to a question, which filters based on the direction the user is interested in, such as: north, west, northeast, etc. We define eight directions and specify a specific angle range for each as shown in Figure 5.2. For example, the northeast direction will have the following SQL predicate:

```
degrees(ST_Azimuth(anchoring_point, geometry)) BETWEEN 22.5 AND 67.5
```

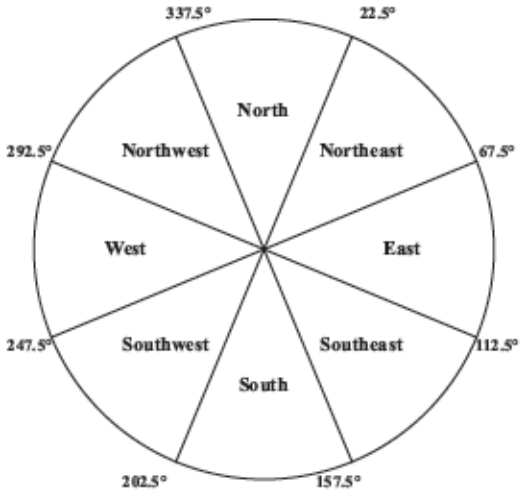


Figure 5.2: Direction angle ranges

Towards: For this predicate, an additional anchor point is added that determines the direction in which the user is interested. The angle is computed with the following SQL:

```
degrees(ST_Azimuth(ancrhor_point1, anchor_point2))
```

The geometries are then filtered similar to the direction predicate, but the angle range is based on the computed angle ± 22.5.

Intersects: This type of predicate takes a defined region as its anchor, like the boundaries of a city or a state, instead of anchoring points as the previous predicates. The intersection predicate selects the objects that intersect with the provided boundary, and we use it in questions that ask for aggregate values.

The *direction* and *towards* predicates are combined with one of the *nearest neighbor* or *range* predicates. As such, in total, we have seven different spatial selectors. Although there are many predicates that we can consider, we selected these five predicates to generate

a diverse set of spatial questions. More variety can be easily added in the future, for example, to consider more topological operations such as contains, overlaps, touches, etc.

Non-Spatial Predicates

These provide an additional aspect to create more complex questions. They are based on other attributes that exist in our tables that we defined in Section 5.3.1. We only selected non-spatial predicates for points representing restaurants, museums, and hospitals. For museums, the predicate is to specify the type of museum in which the user is interested. For restaurants, the predicates are about the cuisine, but also about services such as delivery, drive-through, or outdoor seating. And for hospitals, we added a predicate for filtering for hospitals that provide emergency services. We can easily add many more predicates of this type, but the ones that we selected provide us with enough variety to generate many natural-sounding questions. This component is used to create four additional templates from T1, T5, T13, and T17. These are templates T2, T6, T14, and T18. The difference is in how POI_CAT is instantiated. For example, with the inclusion of a non-spatial predicate that specifies the type of museum, it would be instantiated with the phrase `'art museum'`, and the SQL will include this predicate `museum = 'art'`.

Output Types

The output type or the answer type also adds another aspect to the questions. The user may ask about the name of an entity based on the provided predicates or about the location of the desired entity, such as the coordinates or the address. Additionally, the outputs can also be obtained from spatial operators such as distance, area, and length, which can also be aggregated according to the question.

Entity name: The display name of the spatial object that matches the question.

Location: The coordinates or the complete address of the spatial object that matches the question.

Direction: The azimuth angle in degrees toward which the spatial object that matches the predicates of the question can be found starting from the anchor point of the question and where zero represents the north direction.

Distance: The distance at which the spatial object that corresponds to the predicates of the question can be found, starting from the anchor point of the question.

Count: The number of spatial objects that match the question according to the topological operation it requires.

Area: The area in meters squared of the spatial object or objects that match the predicates of the question.

Length: the length in meters of the spatial object or objects that match the predicates of the question.

External: external responses involve information retrieved from outside the system (out-of-schema) after determining the spatial objects that answer the question. Examples include the capacity, the year it was created, the person who designed, etc.

Aggregate operations

Some templates require an aggregation step before producing the final answer. These templates include those that ask for the total area or length, or the count, as well as the templates that ask about the entity that has the maximum area or length.

5.3.3 Open Retrieval

We also create a set of multi-hop questions, which involve information that is not part of our reference data described in subsection 5.3.1. This adds another layer of complexity to these questions.

We create this by selecting one of the entities in a generated question, which could be an anchor POI or the entity that represents the answer. Then, we retrieve some information from Wikipedia about it, such as the year it was built, who built it, among several other attributes.

We create two types of multi-hop questions. The first type involves asking about out-of-schema information for the entity of the answer, like T7 in Table 5.3. There are several ways to phrase the question in this case, for example, `'What is the nearest hospital from ...?'` can be changed to `'What type of emergency department is available at ...?'`. So to answer such a question, first the nearest hospital must be determined, then out-of-schema information about it must be obtained.

The second type of multi-hop question involves modifying the name of the anchor point with out-of-schema information; see T8 in Table 5.3. One such example is that when the anchor point is a university, then its name can be replaced by `'the university with ... as its mascot'`.

These two types add more complexity, since the answer cannot be directly obtained from the reference data. And require multiple steps and synthesizing information from multiple sources to obtain the correct answer.

Although we only included these two types, it is possible to add more variety based on them. For example, region names can be replaced with descriptions about something that uniquely identifies them. Also, instead of just using Wikipedia, other sources can be used. However, we believe that the two types that we included and the attributes we considered are sufficient to test the ability of QA systems to answer questions that involve multiple steps, especially since selecting suitable attributes and ensuring that they translate to natural-sounding questions involves some manual work. Also, later we show that even at this level of complexity the baselines we consider fail in answering these types of questions.

Next, we discuss the methodology for instantiating a question from a template and identifying the correct answers.

5.3.4 Question Generation

In this section, we discuss how to generate questions from a template. The complete process is shown in Figure 5.3, which shows an example of a question instantiated using the T3 template from Table 5.3. Each template has a fixed set of possible phrases in addition to the one shown in Table 5.3. The first step is to select a text phrase randomly from all the phrases available for the template that were created and stored in a previous step. In this step, we used an LLM to create different phrases for each template of questions. We also manually edit the LLM output to ensure that all text phrases for a template are valid. Each template also has an associated SQL template. Once we have the question phrase and the SQL template, we can start with instantiating them.

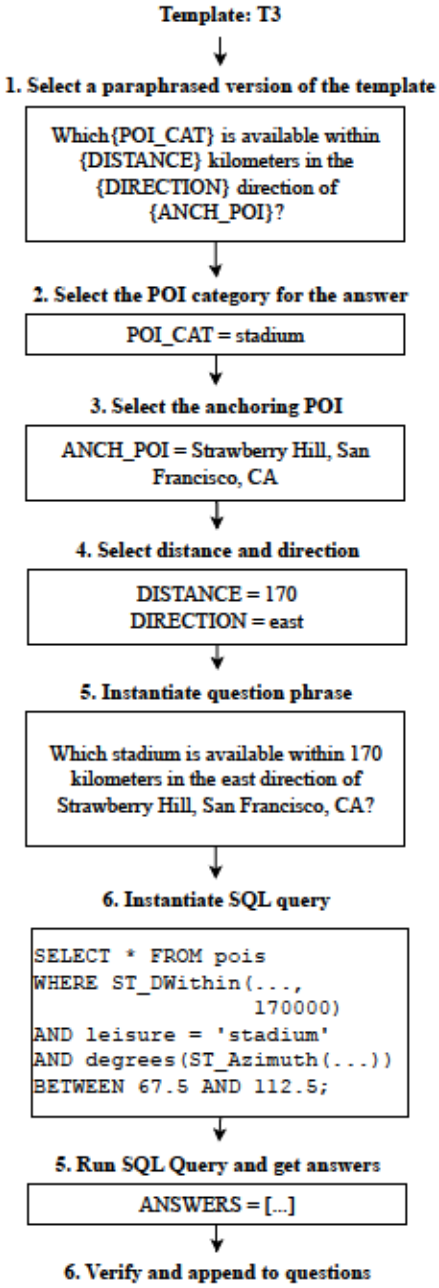


Figure 5.3: Example of Generating a Question from a Template

In the next step, we start by selecting a value for `POI_CAT` that the question asks for. The category is selected randomly from a defined list of possible categories. Additionally, for templates with a non-spatial filter, the filter is appended to the category name and added to the SQL query. Once a category and its filter are determined, a POI that matches it is selected randomly. This POI is used to limit our search in the next step. We do this to ensure anchor points are chosen in locations with existing answers, as we exclude questions without answers. Without limiting the search space, the generator takes a long time to find questions with answers. We excluded questions without answers, though evaluating a QA system on such questions could be insightful. However, generating such questions is simple, as we can create them using random names and coordinates. Next, we randomly select a POI for `ANCH_POI` from a predefined list of categories and within the restricted region identified above. After that, we randomly select a value for `DIRECTION` and `DISTANCE`. All parameters are now defined. Next, both the question phrase and the SQL can be instantiated. The question text is also passed to a grammar checker and correction tool because some templates require pluralizing a name or appending the *'a'* or *'an'* articles.

The next step is to run the instantiated SQL query to get the answers. If the SQL yields no answers, the question is skipped, and the process restarts with new values. Otherwise, a verification and quality check step is applied before the question is appended with all its associated entities to a question bank.

For multi-hop questions, the question is modified after identifying relevant out-of-schema information based on its type. This process was described in subsection 5.3.3.

Other templates are instantiated following a similar procedure based on the type of objects in the template.

For each template, in the question generation step, we generate 1000 questions, with the exception of T7 and T8. A total of 26,000, all of which have passed this quality check. Additionally, a number of 150 questions for each multi-hop type, T7 and T8, are included, since they are more expensive to generate. In the final benchmark, we include only 100 questions of each type, which adds up to 2800 questions. Next, we discuss the process of how these questions were selected from the larger set.

Using a question template that has `POI_CAT` and `ANCH_POI` as parameters, such as T5, we can divide its 1000 questions into hundreds of smaller sets. For example, in this case all the questions that ask about a "restaurant" (`POI_CAT`), and starting from a "park" (`ANCH_POI.category`), will be grouped together. Given the variety in all of these parameters, we will have more than 100 groups, and we select one question from each group that enhances the variety and richness of the questions included in the benchmark. We were able to perform this selection by identifying a set of parameters that allow the questions to be grouped into more than 100 groups for each template.

5.3.5 Quality Checks

This section outlines how we check the quality of benchmark questions. In performing quality checks, we make an attempt to ensure that the questions are reasonable. That they can be asked by average users and are not ambiguous to a human reader. For this reason, we only consider more prominent records when instantiating a question. One reason for requiring this is that it helps us avoid inaccurate or incomplete records, since

OSM is crowd-sourced. This verification step involves multiple types of checks. For POIs, we require that they have a name and some address attributes. While it is possible to geoencode a point's coordinates to an address, we use this as a simple indicator of prominent POIs, since a large percentage of POIs in OSM do not have these fields set. For other types of objects, like regions, we require that they have a Wikipedia page available if they are used in anchoring a question, since a lot of regions are administrative regions, and may not even have a name. A region with a Wikipedia page ensures that we select more prominent regions. Additionally, for questions that have a number as their answer such as length, distance, area, and count, we ensure that the answer is not zero.

Finally, we manually review about 10% of the questions for each template with their answers to ensure their correctness.

5.3.6 Keeping GS-QA Up-to-date

It is important to keep the benchmark based on the most recent reference data, because geospatial data is constantly being updated. Since we are using OSM as our main source, in this section we discuss how the benchmark can be updated to match the most recent version of OSM.

For every question that we included in the benchmark, we store with it the OSM records that were used in building the question. Each OSM record has a unique identifier. The process is straightforward and starts by obtaining the most recent version of OSM, and rebuilding our database using it. Next, for every question in the benchmark, we check if any of its associated records changed and update the values if needed. The next step is rerunning the SQL query again to get the answers and update them if needed. Then, the

question must also pass the quality checks that we described earlier. For questions that do not pass the quality check, we can discard them and include a new generated question from the same template and possibly based on the same categories but associated with new OSM records.

5.4 Baselines

To test the validity of our benchmark, we build six baselines to evaluate how they perform on it. The purpose of building these baselines is three-fold. First, they allow us to understand how the state-of-the-art LLMs behave with the benchmark, which can reveal interesting research problems. Second, the results of these baselines will allow future, more efficient techniques to be compared with these results. Third, they provide an example of how the benchmark can be used and verify its function.

We define *three types* of baselines and each is tested with *two LLMs*, one proprietary and another open-source, for a total of *six baselines*. We also add an additional baseline that is based on a random answer generator. The first type of baselines is based on feeding a question directly to an LLM without any context. The second type of baseline incorporates a Text2SQL step that takes the question and possibly some context about the tables in the database to translate the question into an SQL query first. Then, the SQL query is passed to our database, and the answer is used as additional context for the model to provide the final answer to the question in a textual form. The third type involves Retrieval Augmented Generation (RAG). In this method, a datastore is used to maintain reference data. Before passing a question to the LLM, a retrieval step is performed to get the most relevant records

Table 5.5: Baseline combinations

Label	LLM	Text2SQL	Retrieval
G	GPT4-o	×	×
L	Llama 3.2	×	×
GT	GPT4-o	✓	×
LT	Llama 3.2	✓	×
GR	GPT4-o	×	✓
LR	Llama 3.2	×	✓
R	Random	×	×

for the question from the datastore. The retrieved records are appended to add context to the LLM to help enhance the final answer. In total, we have six baselines summarized in Table 5.5, and each is discussed in detail in the following subsections.

5.4.1 Bare LLM Baselines

We select two popular models for all our baselines, one proprietary and one open-source. These models will help evaluate how state-of-the-art LLMs perform when asked questions from our benchmark, given the various scenarios when different types of context are provided. Next, we briefly describe the models that we use.

GPT-4o: It is a transformer model developed by OpenAI and achieved state-of-the-art results in several domains at the time of its release. It can take as input multi-modal data including text, images, and audio, and can generate all three types [12]. This model's context window is 128 thousand tokens, and its maximum output size is a little over 16 thousand tokens.

Llama 3.2: An open-source model developed by Meta. It achieved state-of-the-art in various natural language tasks. Its context window size is 128 thousand tokens, and we use a distilled version with 3 billion parameters [14].

In the first two baselines, G and L, as shown in Table 5.5, we use prompt engineering to feed the question to the LLM without any additional work. The prompts are designed to provide a concise answer in a standard format that is expected by the benchmark and easy to evaluate. The following is the system prompt:

```
Answer the provided user question while satisfying the following requirements:
1. do not include any parts of the question in the answer you must provide the
↪  answer directly.
2. provide only the property the user is asking for, like name of an entity, its
↪  location, distance, direction.
3. don't provide information the user didn't ask for.
4. any number must be written as words and rounded to the nearest ten.
5. only use metric units.
```

We found that this prompt provides concise answers that are relevant to the questions, which are more suitable for text-based evaluation discussed in Section 5.5.1.

5.4.2 Text2SQL Baselines

This baseline uses an LLM to answer questions in three steps. First, it uses the LLM as a Text2SQL generator to create a SQL query out of the question. This step also feeds the database schema to guide the generation process. Second, it runs the produced SQL query on a traditional database to get the answer. Finally, it feeds the SQL result as context to the LLM to produce the final answer in the desired format.

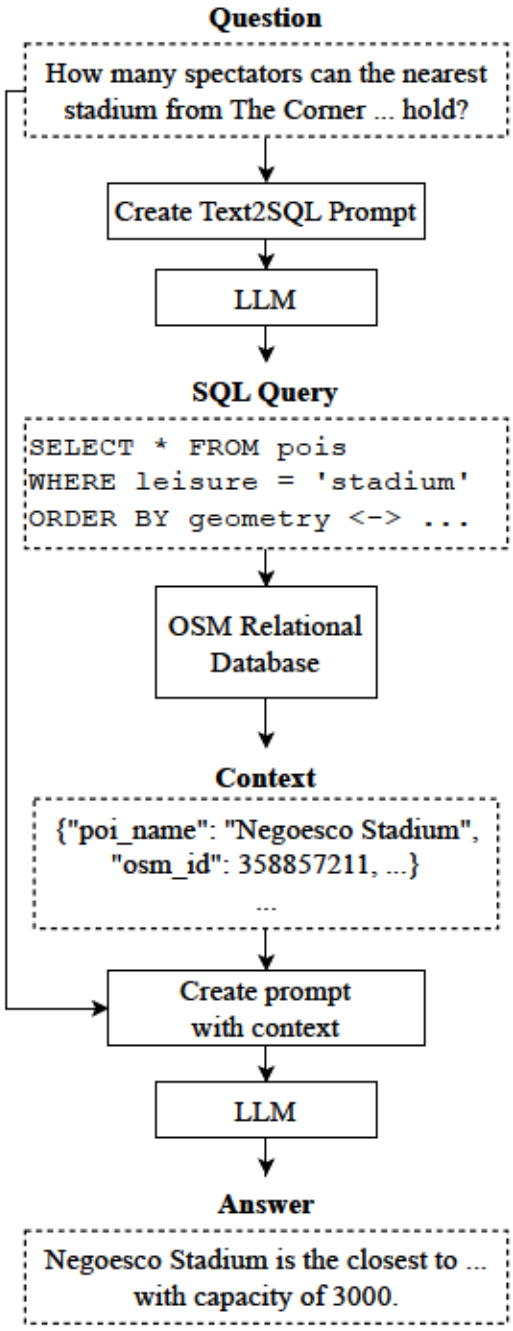


Figure 5.4: Question Answering Pipeline with Text2SQL

The Text2SQL problem using LLMs has recently received a lot of attention. There are various benchmarks specifically for this task, like [74, 127, 237]. Although our objective is not to provide a Text2SQL benchmark, our benchmark can be specifically used to evaluate queries that include spatial components. We will include this evaluation later in the experiments. Figure 5.4 shows the complete QA pipeline when Text2SQL is used. The Text2SQL prompt includes the user's question, and context that includes information about the database schema like the table names, and information about each table and its columns. After passing the prompt to the LLM, we extract the SQL query produced by the LLM, and execute it on our database. If the query is valid and produces an output. This output is used as context to get the final answer to the user's question. The baselines GT and LT are based on this pipeline.

5.4.3 Retrieval Augmented Generation Baselines

One way to improve the quality of the generated answers is to use *Retrieval Augmented Generation* (RAG) [120]. This method works by first running a similarity search to retrieve the most similar documents to the question and then uses them as context to answer the question by the LLM. Note that technically the previous Text2SQL pipeline can also be considered a RAG method, but here we mainly refer to dense-retrieval methods [235]. In dense-retrieval, the search is based on embeddings that represent some text. In our case, we use the distilled LLAMA model to generate our embeddings. A datastore is needed for an efficient RAG-based baseline. We build it using the LLAMA embeddings,

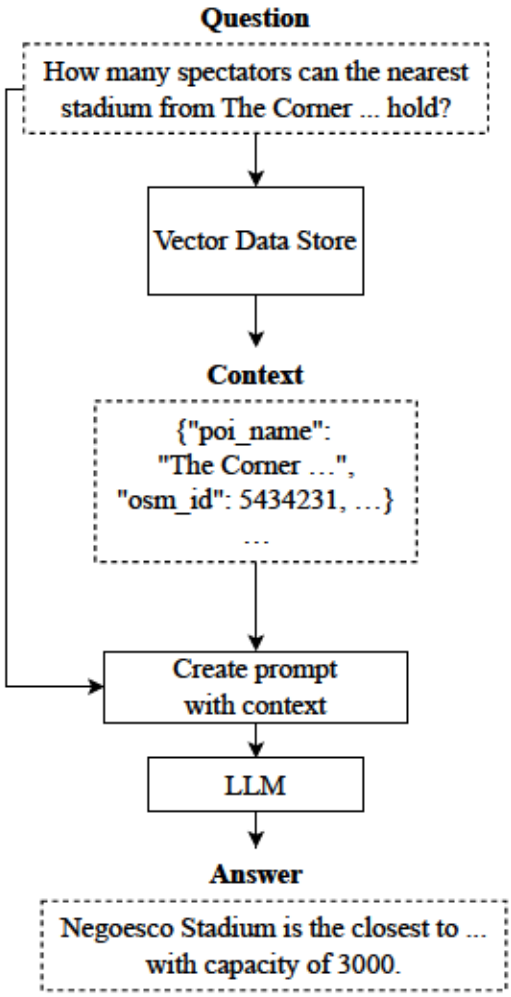


Figure 5.5: Question Answering Pipeline with RAG

and use ChromaDB [16], an open-source database for storing those embeddings, and performing efficient searches. We create our datastore by including all records in all the tables in the reference database that we used when generating the questions, and a small subset of Wikipedia, containing those pages that were considered when generating the multi-hop questions.

To answer a question, we first encode it using the same distilled LLAMA model, and then we use ChromaDB to get the top-10 records with the most similar embeddings. After that, we prompt the LLM with the question and provide the closest documents as context to get the final answer. The baselines GR and LR are based on this pipeline.

5.5 Experiments

In this section, we study the effectiveness of the baselines described in Section 5.4 on the GS-QA benchmark. Simply applying standard text-based evaluation techniques, comparing the answer string to the ground truth string, is not adequate, given the inherent structure of spatial answers. For example, two answers may not share any words, but they may be spatially too close to each other. For that, we consider a comprehensive suite of spatial-aware evaluation measures, in addition to standard text-based measures, as discussed in Section 5.5.1. The results are presented in Section 5.5.2 and a discussion in Section 5.5.3.

5.5.1 Evaluation Strategy

We use two evaluation approaches: *text-based matching*, and *structured spatial-aware evaluation*.

The text-based matching searches for common words between the ground truth and the generated answers. Before matching, a simple preprocessing step is applied to remove punctuation, convert to lower case, and convert numbers to words. Then, three values are computed. Precision is the percentage of common words between the two sentences divided by the number of words in the predicted sentence. Recall is the percentage of common words divided by the number of words in the ground truth answer. The F1 score is the harmonic mean of the two scores. When precision and recall are both zero, we also set F1 to zero. We do not consider scores like BERTScore [230], since when evaluating addresses or entity names exact matching is required as compared to semantic similarity.

For the structured spatial-aware evaluation, the answer is first converted to a JSON document, using the 3B Llama 3.2 model and a prompt specifying the desired JSON schema, as in the following example:

```
The nearest science museum is the Orlando Science Center,
located in Orlando, Florida, approximately fifty kilometers
north of the Central Florida Zoo & Botanical Gardens,
Sanford, FL.
```

Converted to JSON:

```
{
  "name": "Orlando Science Center",
  "address": "Orlando, Florida",
  "distance": 50, "azimuth_angle": 90,
  "count": null, "length": null, "area": null
}
```

Note that inaccuracies might occur in this step, such as setting the angle to 90° for the north direction when north is 0°. Also, additional attributes such as "architect" might be included for multi-hop questions. Based on the JSON fields, we consider spatial-aware measures, customized for each output type, as shown in Table 5.6. Specifically, when the answer type is an entity name, we use the same metrics as in the free-text form. We expect that the scores in this case will be better because of the more compact representation, compared to the text-based matching approach. Similarly, multi-hop answers are evaluated as text using the same metrics.

When the answer type is a location, it is evaluated in two ways. First, the address text is the three text-based measures. In addition, we calculate the distance in meters between the geo-encoded position of the predicted address and the expected position. The distance is divided by a threshold, which we set to five hundred kilometers. If the distance is more than the threshold, the error is set to the maximum value of one. Also, given this threshold, an error of 0.01 is equivalent to five kilometers in distance. We use the Open Street Map free geoencoding service, called Nominatim [15]. This step may not always be accurate.

When the answer type is an angle, we evaluate using both an exact match text approach and a geometric metric. For the text-based approach, the angles are converted to their description, and if both angles fall in the same range, they will be an exact match. Angle ranges are obtained by dividing the 360°, into eight equal ranges, as shown in Figure 5.2. For the angle value, an angle error score is computed using the following equation, where zero is an exact match and one is a 180 degrees difference.

$$\text{Angle Error} = \begin{cases} \frac{|\text{angle1} - \text{angle2}|}{180}, & \text{if } |\text{angle1} - \text{angle2}| < 180 \\ \frac{360 - |\text{angle1} - \text{angle2}|}{180}, & \text{otherwise} \end{cases} \tag{5.1}$$

For all other output types, which are numeric, the relative error is used to measure the quality using the following equation.

$$\text{Relative Error} = \frac{|\text{Prediction} - \text{Actual}|}{\textit{Actual}} \tag{5.2}$$

For all questions in the benchmark, the actual answer for the output types Area, Length, Distance, and Count is never zero, which makes the relative error defined for all questions. For all output types, when the attribute related to it is missing or is out of the expected range, the question is counted as not attempted, such as when the address value is missing or empty and the output type is location. Additionally, the geoencoding step may not produce an output, which might indicate that the address is malformed, and these questions are also counted as not attempted for the distance error score.

For questions that can have multiple valid answers, like questions with the *Range* predicate, each predicted answer is compared to all possible answers, and only the best scores are reported.

There are some considerations for questions that can have multiple valid answers. One consideration is to rank all the possible answers based on some quality metric. For example, when there are many restaurants that fit the criteria of the questions, we give more weight to more popular ones. Furthermore, some types of questions can be interpreted differently. For example, users asking about lakes may only be interested in natural lakes,

Table 5.6: Metrics used for parsed output

Output Type	Recall Full-text	F1 Parsed	Distance Error	Angle Error	Relative Error
Entity name	✓	✓			
Location		✓	✓		
Direction		✓		✓	
Area					✓
Length					✓
Distance					✓
Count					✓

Table 5.7: Percentage of attempted questions in parsed answers

Output type	G	GR	GT	L	LR	LT	R
name	0.97	0.77	0.93	0.95	0.82	0.92	0.96
loc	1.0	0.74	0.97	0.99	0.95	0.89	0.98
angle	0.91	0.87	0.8	0.92	0.72	0.84	0.91
area	0.99	0.89	0.76	0.99	0.44	0.99	1.0
count	0.93	0.87	0.9	0.94	0.74	0.86	0.53
distance	0.99	0.94	0.96	1.0	0.8	0.96	0.97
length	0.49	0.31	0.56	0.83	0.34	0.8	0.85

while the system may consider man-made lakes and seasonal lakes. We leave the evaluation of these special cases for future work.

5.5.2 Evaluation Results

In the experiments, we organize the results by output types, as shown in Table 5.3, because each type may have its own unique evaluation measures (Table 5.6).

Attempted questions We mark a question as attempted if the generated JSON contains a value for the required key, e.g., for the key "address," "name" or "distance." We provide a summary of the percentage of questions attempted in Table 5.7. We see that numeric questions are less likely to be answered, which we discuss in more detail for each output type.

Table 5.8: Evaluation of templates with entity name

TID	Recall on full text output							F1 on parsed output						
	G	GR	GT	L	LR	LT	R	G	GR	GT	L	LR	LT	R
T1	0.07	0.03	0.06	**0.08**	0.07	0.08	0.0	0.39	**0.43**	0.36	0.35	0.29	0.34	0.35
T2	0.09	0.02	0.13	**0.13**	0.12	0.12	0.0	0.26	0.23	**0.39**	0.27	0.24	0.26	0.29
T3	0.11	0.06	**0.14**	0.1	0.08	0.1	0.0	0.3	0.22	**0.35**	0.29	0.22	0.26	0.3
T4	0.03	0.02	0.02	**0.04**	0.03	0.03	0.0	**0.48**	0.45	0.4	0.46	0.36	0.41	0.47
T5	0.18	0.06	**0.42**	0.18	0.13	0.16	0.12	0.17	0.08	**0.42**	0.1	0.11	0.09	0.12
T6	0.14	0.09	**0.46**	0.21	0.14	0.2	0.15	0.14	0.14	**0.45**	0.16	0.12	0.15	0.15
T7	0.01	0.02	0.06	0.21	0.14	**0.23**	0.0	0.01	0.02	0.01	0.03	0.01	**0.04**	0.04
T8	0.15	0.08	0.15	**0.17**	0.15	0.17	0.1	**0.15**	0.11	0.14	0.12	0.11	0.11	0.1
T9	0.15	0.09	**0.18**	0.16	0.12	0.16	0.0	0.13	0.08	**0.17**	0.12	0.07	0.12	0.13
T10	0.12	0.05	0.09	**0.12**	0.09	0.1	0.0	0.09	0.04	0.08	**0.11**	0.07	0.08	0.09
T11	0.38	0.34	**0.39**	0.34	0.28	0.33	0.0	0.45	0.44	**0.46**	0.3	0.25	0.28	0.25
T12	**0.26**	0.26	0.23	0.19	0.12	0.18	0.0	**0.26**	0.26	0.24	0.16	0.14	0.16	0.12
AVG	0.14	0.09	**0.19**	0.16	0.12	0.16	0.03	0.24	0.21	**0.29**	0.21	0.17	0.19	0.20

Return type: Entity name We show the results for twelve templates from Table 5.3 in 5.8. We consider two scores. For the free text output, we look at the recall (R) score. For the parsed output, we use the F1 value to compare the true answer to the extracted entity name, or the multi-hop attribute in the case of T7. This score only includes the attempted questions as defined earlier. We notice that the baseline GT has considerably higher scores for several question categories.

Moreover, we manually checked many of the answers, and noticed a lot of the answers are correct, and they correlate with the scores shown in the table. Also, one observation is that the scores are affected by common words in the entity names. For example, many parks and lakes have the word 'Park' and 'Lake' in their official names. Therefore, if the official name has only two words, the presence of these words would give precision and recall at 50%. On the other hand, some names are spelled a little differently than the reference answer, which results in penalizing correct answers. With regard to

Table 5.9: Evaluation of templates with location

TID	F1 for address text							Distance Error						
	G	GR	GT	L	LR	LT	R	G	GR	GT	L	LR	LT	R
T13	**0.27**	0.19	0.25	0.24	0.22	0.24	0.26	**0.45**	0.53	0.48	0.54	0.58	0.52	0.84
T14	0.24	0.20	**0.25**	0.19	0.17	0.21	0.24	**0.34**	0.54	0.45	0.55	0.63	0.54	0.80
T15	0.20	0.17	0.20	0.14	0.13	0.18	**0.20**	**0.45**	0.52	0.54	0.54	0.60	0.52	0.66
T16	**0.38**	0.35	0.38	0.26	0.28	0.28	0.36	**0.19**	0.39	0.19	0.50	0.68	0.63	0.68
T17	0.19	0.15	**0.24**	0.16	0.16	0.18	0.08	0.46	0.57	**0.39**	0.70	0.58	0.62	0.87
T18	0.10	0.09	**0.17**	0.10	0.09	0.07	0.16	0.64	0.64	**0.54**	0.61	0.67	0.66	0.80
T19	0.14	0.10	**0.16**	0.09	0.11	0.10	0.15	0.49	0.57	**0.44**	0.61	0.57	0.63	0.81
T20	0.13	0.06	0.13	0.14	0.11	0.16	**0.20**	0.34	0.74	**0.32**	0.61	0.77	0.71	0.80
AVG	0.21	0.16	**0.22**	0.17	0.16	0.18	0.21	0.42	0.56	**0.42**	0.58	0.63	0.60	0.78

multi-hop questions, T7 and T8, we noticed that none of the baselines correctly answered any of the questions.

Return type: Location The results are provided in Table 5.9. Looking at the distance error score, we see that several templates have errors noticeably better than the random baseline. We observed that many valid addresses were provided, especially by the baselines using GPT-4o. However, there are a few cases where the baseline did not provide an address, and in the parsing step, the LLM selected the address of the ANCH_POI from the question instead of the answer.

Return type: Direction We show the results in Table 5.10. Since the angle has a limited range of possible values, we first compute the expected error for a random number generator baseline. We find the expected angle error to be 0.5, since on average the angles would be at 90° difference in either direction. Based on this, we notice that for T22, which is based on the k-nearest-neighbor predicate, all the angle errors are around the expected error for a random generator, even though there are quite a few questions where the correct direction was predicted. For T21, the error is less because the score keeps the best matching prediction to any of the possible correct answers. The expected error

Table 5.10: Evaluation of templates with direction

TID	F1 for parsed direction							Angle Error						
	G	GR	GT	L	LR	LT	R	G	GR	GT	L	LR	LT	R
T21	**0.62**	0.54	0.62	0.57	0.45	0.56	0.53	**0.16**	0.23	0.19	0.20	0.26	0.21	0.20
T22	0.16	**0.19**	0.16	0.11	0.13	0.18	0.09	0.51	**0.48**	0.50	0.51	0.49	0.52	0.54

depends on the number of possible correct answers and the number of predictions. For example, if we have only one possible correct answer, and three predictions were provided, the expected error for a random generator is around 0.25. We obtained these expected errors by running a simple simulator. Note that the error is also affected by the number of unattempted questions as defined earlier, since these get an error of 1.0. Similar to the location-based questions, we think a more sophisticated evaluator would first evaluate the predicted entity and that it exists in the reference data, which could be different from the reference data used for building the benchmark, and that it matches all the specifiers in the question. Only after that can we evaluate its direction relative to the ANCH_POI in the question.

Other return types We provide a summary for the remaining templates in Table 5.11. Note that we limit the relative error to a maximum of 1.0. Hence, any baseline with a score of 1.0 provided answers very far from the correct answer.

The baselines performed worse compared to the random number generator for template T25. For template T26, G performs a little better than random, and this happens when the provided answer has a small distance, since most of the correct answers have small distances. We think the evaluation of these types of questions should also evaluate the correctness of the entity where the distance is measured. Otherwise, the LLM may provide some random numbers that may seem correct. Note that we store the entire entity

Table 5.11: Relative error for templates with numeric answers

Output type	TID	G	GR	GT	L	LR	LT	R
Count	**T23**	**0.77**	1.00	0.78	0.74	0.78	0.75	0.96
	T24	0.98	0.95	0.97	**0.83**	0.92	0.86	0.98
Distance	**T25**	0.95	0.93	0.96	0.96	0.94	0.96	**0.81**
	T26	**0.88**	0.92	0.95	0.95	0.93	0.93	1.00
Area	**T27**	0.99	0.99	0.99	0.92	0.99	**0.92**	0.98
Length	**T28**	0.85	0.84	**0.82**	0.87	0.87	0.90	1.00

in the reference answer, and the associated SQL query to the questions gets the entity as well as the distance, in case the user has their own reference data.

T27 and T28 are analytical queries that require aggregating the area or length of many geometries that intersect with a provided search area. Baseline LT for template T27 seems to have a few questions with good answers, but we think this is also due to randomness, rather than an actual quality prediction. For example, it produces 50000 as an answer to many questions, and it happens that some questions have an answer that is close to this value. We also make a similar observation for T28. Another thing for T28, is that many questions were left unattempted, and this is due to the parsing stage, while the baseline provides some number text. The parsing step does not detect this number as a *length* value, and either ignores it or assigns it to the distance value.

5.5.3 Discussion

Analysis of Text2SQL performance The Text2SQL pipeline resulted in significant improvements in the answers for templates T5 and T6. As mentioned, both templates are only based on the nearest neighbor predicate and other non-spatial predicates. We categorize the generated SQL queries into valid and invalid, and summarize the results in Table 5.12. GPT4-o (G) generated valid queries for about 73% of the questions, while the

Table 5.12: Text2SQL error summary

Category	Subcategory	L	G
Valid SQL	Ran successfully	117	1832
	Timed out	528	203
	Total	645	2035
Invalid SQL	Syntax error	743	16
	Function does not exist	692	274
	Operator does not exist	116	2
	Column does not exist	306	405
	Relation does not exist	169	0
	Missing FROM clause	126	3
	Sub-query error	4	37
	Other	43	28
	Total	2199	765

distilled LLAMA (L) generated valid queries for only about 23% of the questions. Note that a valid query does not necessarily mean the query resulted in retrieving the correct answers. A large percentage of the queries timed out. The time limit for each query is set to be twice that of the maximum time it takes to run any of the queries in the corresponding template that we use to get our true answer. Furthermore, the queries that ran successfully may retrieve incorrect answers. Also, we limit the retrieved records to only 20 records, since some of the generated queries result in retrieving a very large set. We further categorize the invalid SQL queries into multiple sub-categories. A lot of errors are caused due to improper use of spatial functions and predicates. There are clearly a lot of improvements needed for a more accurate Text2SQL when it comes to geospatial data.

Analysis of RAG performance From the previous results, it is clear that the dense-retrieval-based baselines GR and LR consistently perform the worst. This is mainly due to the quality of the embeddings, which we found to not give importance to location information. To evaluate this method further, we manually crafted a few examples to test its quality. For example, when we search using non-spatial terms like the type of cuisine,

generally relevant records are retrieved. However, when we search using the name of a city usually it does not retrieve relevant records. When combining non-spatial keywords with the name of a region, usually records relevant to the non-spatial keywords are retrieved. Clearly, a more suitable dense-retrieval method that gives importance to geospatial information is needed, since a keyword-based sparse-retrieval might have performed better.

Summary of results The results show that only questions with *entity name* output type provided relatively good answers and only for those questions with the simplest spatial predicate. However, for other answer types, the baselines mostly provide random answers. We also discussed the existing limitations in the evaluation method.

5.6 Conclusion and Future Work

We created an extensible benchmark, GS-QA, for spatial QA, based on an automated question and answer generator. GS-QA includes a variety of spatial predicates, geospatial and non-geospatial entities, and output types. We proposed spatial-specific evaluation measures that go beyond standard text-based matching. We have shown that existing LLM-based baselines are not sufficient to answer such questions.

We have identified several future research directions. First, while we have proposed an evaluation strategy that takes spatial characteristics into account, a more sophisticated and standardized evaluation is still needed. For example, we may want to quantify how good the answer "California" is if the correct answer is "Los Angeles." Further, instead of just evaluating based on the final answer, we can also evaluate the correct identification of the releant geospatial entities. Moreover, when extracting a spatial entity from an LLM's

response, parsing errors may occur, which need to be quantified. Second, we argue that more advanced GeoQA systems are needed. Such systems must be able to handle a variety of spatial predicates and integrate non-spatial information, like in the case of multi-hop questions that we included. Third, better retrieval methods that take geospatial information into account are needed, as seen in the discussion for the Text2SQL and RAG results. Finally, research is needed to support even more advanced geospatial questions that could be asked by geospatial analysts in areas such as urban planning and epidemiology.

Chapter 6

GeoGen I: Towards General Geospatial Point Data Generation from Text

6.1 Introduction

Spatial data generation is critical for the development, testing, and evaluation of new algorithms, index structures, and systems. Synthetic data offer a key advantage over real-world datasets by allowing researchers to control characteristics such as size, distribution, and location to match their needs and to promote reproducibility and fair comparison. However, most existing spatial data generators rely on simple statistical or fractal-based models that fail to capture the complexity of real-world spatial distributions [213, 32, 73]. As a result, evaluations based on such synthetic data often overlook the challenges posed by

more realistic scenarios. Although the availability of real spatial datasets is improving [79], they remain difficult to find at the desired scale or region, limiting their practical utility. This gap underscores a pressing need for a data generation framework that combines the flexibility of synthetic data with the variety of real-data distributions. Such a tool would not only streamline experimentation but also lead to more reliable and generalizable system evaluations.

An effective spatial data generator should meet three essential criteria. First, it must support a broad range of distributions that go beyond the standard statistical models to better reflect real-world spatial variability. Second, it should be easy to use and avoid complex configurations with many obscure parameters. Third, it should allow grounding the data to specific geographic regions which helps combining the generated data with other datasets.

Recent advances in generative AI have enabled models to synthesize complex data types, including text [44], audio [139], images [179, 172], and even videos [190], with remarkable quality. These developments have opened new opportunities for synthetic data generation across other domains. However, current generative models lack an inherent understanding of spatial context and geographic semantics, rendering them ineffective for producing realistic spatial data [81, 67]. Recent work has explored the use of machine learning to run agent-based simulations [23] or generate trajectory data [92, 137, 130]. While promising, these approaches are tailored to specific data types and require significant overhead to setup and use.

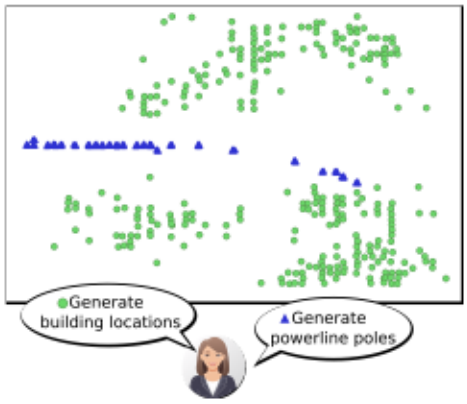


Figure 6.1: Two generated datasets in the same region with two different prompts

This paper introduces GeoGen I, a framework for generating geospatial point data from natural language prompts. For example, given the prompt "generate residential building locations in Irwindale, CA", the system synthesizes a plausible spatial point distribution resembling building placements in the specific region. Figure 6.1 illustrates an example of two datasets generated by the same model in the same region, but with two different prompts, one for power line poles and one for buildings. Even though both are randomly generated data, each one follows a distinct pattern, e.g., a linear distribution for power lines and a clustered distribution for buildings. The use of a generative model has the promise of supporting virtually an endless number of distributions. It is also easy to use since it runs on natural language prompts. Finally, it can be localized to specific locations as we later show in this paper.

Developing GeoGen I as a general-purpose geospatial data generator entails several challenges. 1) **Data availability:** Training requires a large and diverse collection of geospatial datasets paired with descriptive text across various locations. 2) **Representation learning:** The model must learn associations between spatial distributions and

corresponding textual descriptions. 3) **Spatial awareness:** The system must incorporate contextual map data to differentiate between regions and generation location-specific outputs.

GeoGen I addresses the above challenges to produce an end-to-end framework as follows. First, it constructs a large number of data examples from real datasets by applying spatial and attribute-based partitioning to extract a huge number of examples from a few real datasets. Second, it employs contrastive learning to jointly train a text encoder and geospatial encoder, aligning their representations in a common embedding space. Third, it incorporates contextual map data, e.g., land cover or satellite imagery, that guides the generative model and makes it aware of the underlying map data. Finally, it uses a diffusion model to generate the data distribution, which is then converted into sets of geospatial points.

We built a prototype of the proposed model to test its applicability in data generation. We found that the model can generate patterns similar to those of real data across different datasets. We trained multiple version of the generator model, and conducted various evaluations to assess the quality and highlight existing limitations.

The remainder of the paper is structured as follows. First, in Section 6.2 we provide an overview of the proposed system. Second, in Section we discuss the data preparation process and how to curate a dataset and associate it with labels. Then, we discuss the encoders for generating geospatial data embeddings and associating them with text embeddings in Section 6.4. After that, in Section 6.5 we discuss the data generator model, and the steps to convert generated histograms to geospatial points with valid coordinates. Following, in

Section 6.6 we provide our experiments, including a detailed discussion. We discuss the related work in Section 6.7, and provide the conclusion and future work in Section 6.8.

6.2 Overview of GeoGen I

Figure 6.2 provides an overview of, GeoGen I, our proposed spatial data generator. Inspired by text-to-image diffusion models, GeoGen I adapts this generative framework to the geospatial domain. The generation process consists of three main stages, *data preparation*, *embedding*, and *generation* as further detailed below.

Stage 1: Data Preparation: This stage processes input datasets into a training-ready format. It takes as input three types of data. 1) *Region boundaries:* Polygons representing geographic divisions, e.g., states, counties, or cities, that define the spatial scope for training. 2) *Geospatial data:* Point datasets, e.g., building locations, roads, or power poles, that capture various spatial distributions. 3) *Contextual map data:* Auxiliary information about regions, e.g., land use, elevation, or satellite imagery, used to condition generation on physical geography. This stage first applies a three-way spatial join to partition geospatial features into the predefined regions. The data is then grouped by dataset-specific attributes, e.g., building type, to enrich the representation. Finally, we synthesize a textual description based on the region and data grouping to pair the data with natural language input. This stage is further explained in section 6.3.

Stage 2: Embedding: This stage encodes both the text description and spatial data into a shared embedding space. The key challenge is to capture not only individual representations but also their alignment. To address this, we use *contrastive learning* [93, 171]

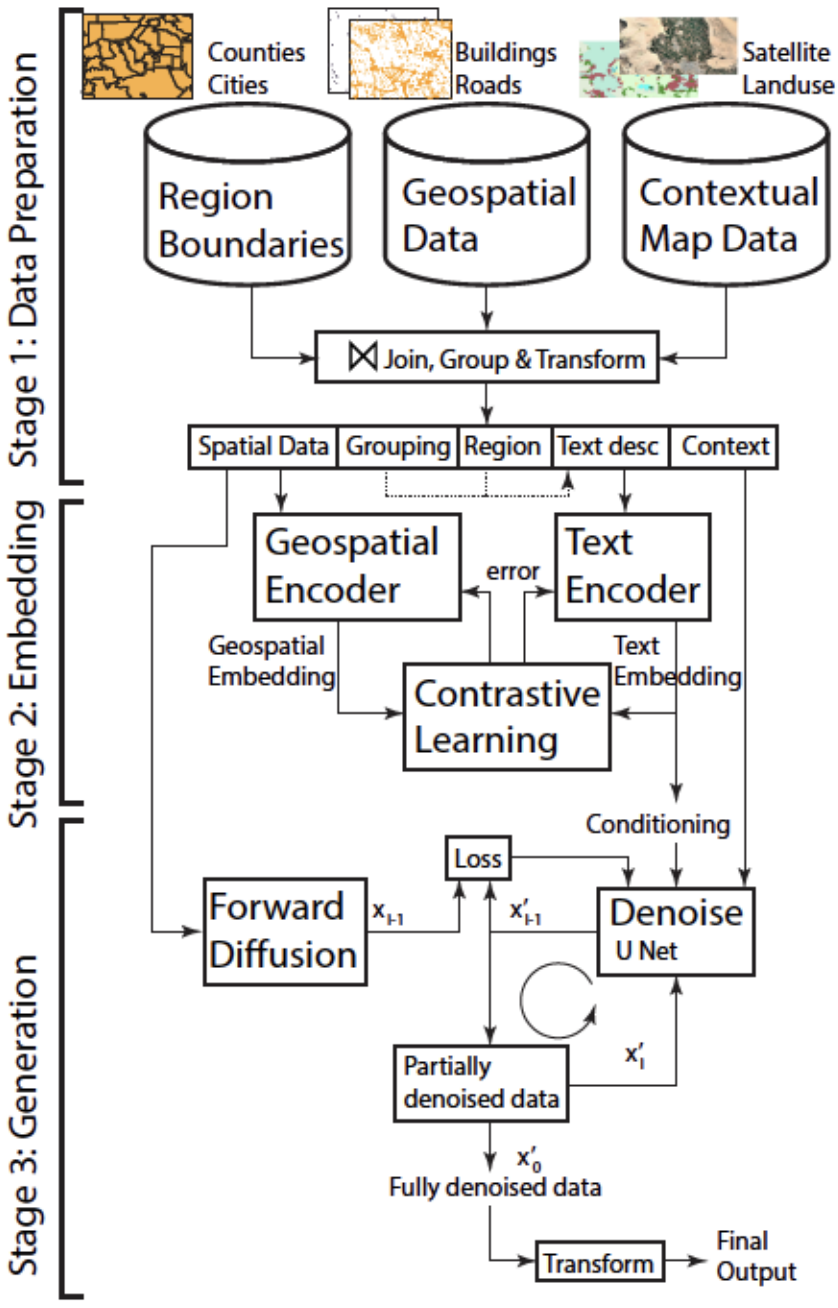


Figure 6.2: GeoGen I System Overview

to concurrently train dual encoders that minimize the distance between the text encoding and its corresponding dataset encoding. This jointly learned embedding enables effective conditional generation. Further details are in section 6.4.

Stage 3: Generation: This stage trains a diffusion model [192] to generate new spatial datasets from text. The *forward diffusion* process incrementally adds noise to an input spatial dataset x_0 to produce a sequence $x_1 \cdots x_n$ with x_n being pure noise. A *denoising model* is then trained to reverse this process, step-by-step, transforming x'_{i+1} to x'_i. Starting from pure noise, this iterative denoising yields a synthesized dataset x'_0, which is decoded into a final set of spatial features. The denoising steps are guided by the embedding used for conditioning to produce the desired output. This stage is described in more detail in section 6.5.

Based on this design, we implement a working prototype of GeoGeo I and conduct a rigorous experimental evaluation. We assess the generated data using both qualitative and quantitative methods. A central challenge addressed in this work is how to objectively evaluate the realism and utility of generated spatial data. The prototype and evaluation are further explained in section 6.6.

6.3 Data Preparation

The first challenge in building a generalized spatial data generator is the lack of large-scale, ready-to-use datasets for training. In particular, we require a large number of real datasets each paired with a descriptive text prompt to effectively learn the mapping between spatial distributions and semantic descriptions. These datasets must capture the

diversity observed in real-world spatial patterns. This stage describes how to take a small number of real datasets and systematically enrich them to produce a large number of training corpus. The core idea is to partition each dataset both spatially, by location, and non-spatially, by other attributes, to generate numerous subsets. For each resulting subset, we synthesize a corresponding text description. Below, we introduce the three types of input data and then describe the preparation pipeline.

6.3.1 Input Data Types

Geospatial Data: This includes real-world point-based datasets such as building footprints or road intersections, which form the core training input for learning spatial distributions. These datasets can be sourced from public repositories, e.g., UCR-Star [79], and may include rich attribute information. We focus on datasets that contain additional metadata, such as building type, to support finer-grained learning and grouping during preprocessing.

Region Boundaries: This dataset defines a set of geographic regions, e.g., cities or states, that serve as spatial units for segmentation. While arbitrary partitions such as grid cells or random rectangles are possible, we choose real administrative or natural regions for three reasons. 1) Real regions often encapsulate meaningful and coherent spatial patterns, whereas random partitions may lack content or context. 2) Named regions can be leveraged to enrich text prompts with place-specific descriptions (e.g., urban vs. rural). 3) Users are more likely to query or evaluate generated data in recognizable regions rather than arbitrary spatial units.

Contextual Map Data: This dataset contains auxiliary contextual information from the map that helps the generator ground the data to spatial features. This can range

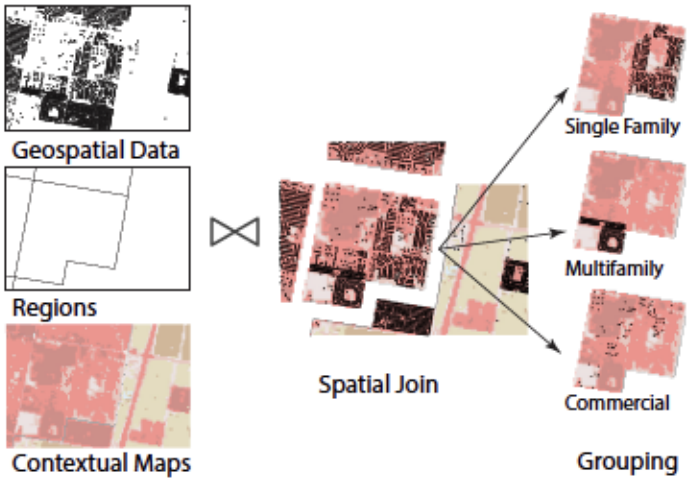


Figure 6.3: Spatial join and grouping

from generic data, e.g., satellite view, to more specific data, e.g., land use, land cover, or elevation. There are two important points to consider for this part. First, the contextual data should be relevant to the spatial data distribution for it to be useful. Second, as the contextual data gets more complex, the model will need more and more data to be able to learn the distinct patterns.

6.3.2 Preparation Process

Figure 6.3 illustrates the first part of the preparation workflow, namely, spatial join and grouping. Given the spatial datasets, region boundaries, and contextual map layers, we first perform a *ternary spatial join* to associate each spatial feature to its context with the appropriate geographic region. This produces a set of region-specific partitions for both the spatial and contextual data as shown in the figure.

Next, we apply a *grouping step* within each region based on non-spatial attributes. For example, buildings can be grouped by usage type, e.g., single-family, multifamily, or

commercial, while other datasets can be grouped by time, e.g., hor, day or season. This breakdown creates smaller, semantically coherent subsets that the model can learn more effectively.

For each resulting subset, we then *synthesize a text description* composed of keywords from all three input sources. From the *geospatial data*, we extract the dataset type, grouping attribute, e.g., multifamily, and the main keywords that appear in other fields, e.g., descriptions or tags. From the *region*, we extract the region type and name and any other high-level classification, e.g., city or rural area. From the *contextual data*, we can optionally extract relevant keywords, e.g., climate or terrain, although our prototype focuses on raster-based features without keyword extraction.

Finally, we rasterize the geospatial data into a fixed-resolution histogram grid, e.g., 64×64, and resize the contextual raster layers to match this resolution. This alignment allows both inputs to be processed jointly by the diffusion model's U-Net architecture, as described in section 6.5.

6.4 Geospatial Data Embeddings

An embedding is a dense vector of floating point numbers that summarizes the information provided to the model that produced it. It is very crucial to have embeddings that accurately capture the information in the input domain. There are several use cases for embeddings. In generative models, they are used to guide the training to produce the desired output. Generally, good quality embeddings result in higher quality outputs as opposed to only providing a text description directly, encoded as a sequence of fixed word identifiers,

for example. Embeddings are also used in dense retrieval methods. In this case, they enable retrieving data based on how similar their embeddings are, using some similarity metrics. They also enable linking information from one input domain to another. For example, embeddings of text descriptions can be used to retrieve images with embeddings similar to that of the provided description. Such a use case is enabled by training two models together using methods like contrastive learning. Where one model takes data from one input domain like text, and one model takes from another domain like images, and the two models are trained together to generate similar embeddings for paired inputs from both the text and image domains. While there is a lot of research on text and image embeddings, there is no existing model for embedding geospatial vector data. In this section, we discuss the different aspects needed for building a geospatial encoder. We discuss the process of contrastive learning, then discuss text embeddings generally and from the perspective of geospatial data. Then, we discuss the topic of geospatial embeddings, and what we expect it can capture, and different types of geospatial encoders that can be designed. Finally, we discuss the applications that can result from having a good quality and general geospatial encoder.

6.4.1 Contrastive Learning

The main idea behind contrastive learning is associating similar samples, positive pairs, and pushing away dissimilar samples, negative pairs, in the embedding space. Contrastive learning also enables working with multi-modal data, by associating data from multiple modalities in the embedding space, such as associating text to images, and vice versa.

It is enabled by using a contrastive loss function like the Info Noise-Contrastive Estimation (InfoNCE). When training for two modalities like text and images, like in CLIP [171], and ALIGN [93], the InfoNCE loss is computed in both directions. First, the loss is computed for each image in relative to all text pairs, as shown in Equation 6.1. In this loss, the similarity between an image and its paired text is measured in the numerator, and its similarity to all other texts in the denominator. The values are also scaled using a temperature parameter τ.

$$\mathcal{L}_{\text{image}} = \frac{1}{N}\sum_{i=1}^{N} -\log \frac{\exp(\text{sim}(x_i, y_i))/\tau)}{\sum_{j=1}^{N} \exp(\text{sim}(x_i, y_j))/\tau)} \tag{6.1}$$

Similarly, the same equation is used in the other direction, by computing the similarity between a given text and all images, as shown in Equation 6.2. Then, both losses are added together for all pairs, which makes the ALIGN loss shown in Equation 6.3. The loss aims to minimize the similarity distance between positive pairs and maximize the distance between negative pairs. This way, text descriptions get embeddings similar to embeddings of images that have content that matches the text.

$$\mathcal{L}_{\text{text}} = \frac{1}{N}\sum_{i=1}^{N} -\log \frac{\exp(\text{sim}(y_i, x_i))/\tau)}{\sum_{j=1}^{N} \exp(\text{sim}(y_i, x_j))/\tau)} \tag{6.2}$$

$$\mathcal{L} = \frac{1}{2}\left(\mathcal{L}_{\text{image}} + \mathcal{L}_{\text{text}}\right) \tag{6.3}$$

This same learning approach can be used to build a model that encodes geospatial data and enables all the applications provided by it. Figure 6.4 illustrates the concept

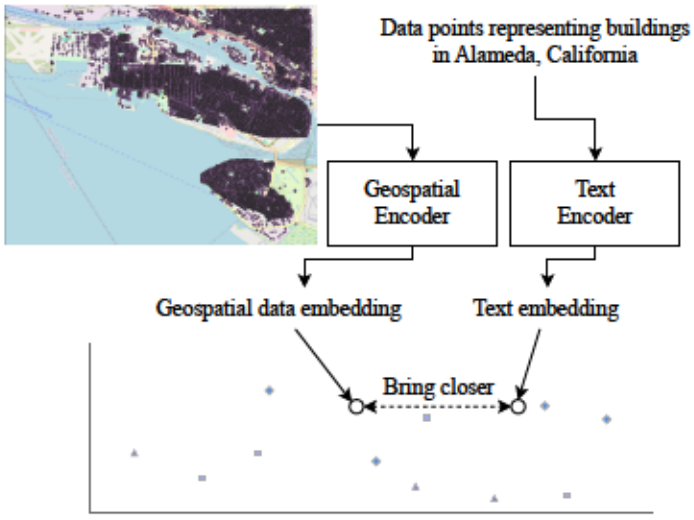


Figure 6.4: Example of contrastive learning

of encoding geospatial data into an embedding space that associates with embeddings of texts that summarize their content. Next, we discuss the text encoder component from the perspective of geospatial applications.

6.4.2 Text Encoder

Having a text encoder that generates embeddings similar to that of embeddings of data from other modalities is very crucial. It enables many applications for translating data between different input domains, such as text-to-image or image-to-text, which can include data generation or retrieval. A good encoder also enables more rich text descriptions without being limited to fixed text descriptions or a fixed set of classes.

There are several choices in the case of text encoders. The first option is using an existing pre-trained text encoder without any finetuning. In this case, the geospatial encoder will be trained to generate embeddings similar to those of the already existing text encoder, which might be limiting its quality. The second is using an existing text encoder

but adjusting it during the training, which might help in improving the text embeddings. This might help produce more representative embeddings and still benefit from the pre-training of the text encoder. The third option is building a new model and training it from scratch. This might be needed if our text descriptions are unique compared to an existing text encoder designed based on text descriptions for images, for example.

In our prototype, we train a custom model from scratch based on the ALIGN [93] architecture, and we compare it to a pre-trained version.

6.4.3 Geospatial Encoder

There are many design choices involved in designing a geospatial encoder. First, the choice for the model architecture, which dictates how to represent the geospatial data. For example, a typical vision encoder using a ResNet [85] can be used. In this case, the data must be represented in an image format such as a rasterized version of the vector data or a summary histogram. Other architectures include graph neural networks, point set-based architectures, and transformer models. The representations for these architectures would require a custom graph representation, a point-set representation which requires minimal processing, or a vector-based representation for each geometry object using encoding techniques like Poly2Vec [189]. In the current prototype, we use a vision model and represent the data as histograms.

Second, deciding what type of regional context to provide with the model. In the case of vision models, we can use a remote-sensing image as context, for example. Other choices also include processed rasters, such as, a raster for land use, road networks, vegetation index, among others. In this prototype, we use remote-sensing images as context.

For other architectures, like graphs and point sets, each node can be associated with some features about its geographic location. It will also still be possible to use raster images in those models by introducing some layers that merge the two representations.

Third, deciding on the data resolution is another challenge. As discussed in Section 6.3 patterns in datasets can be observed at different resolutions. In the case of the geospatial encoder, it is important to decide on specific resolutions and region areas. For vision models, the input is always a fixed-size image. Rescaling the image to very small sizes can result in losing information required to see the patterns in the data. For other architectures, like point sets, this is less of an issue since they take variable-sized input, and inputs are normalized using their MBR.

6.4.4 Potential Applications

While our goal of building these encoders is to build a text to geospatial data model, these encoders can be used in various applications. For example, they can be used in the other direction: building models that take geospatial data as input and generate text descriptions for them. Another potential and important application is building vector databases for geospatial data. We highlight the potential of this use-case in the experiments. However, to build something more general purpose and ready for real-world use-cases, a large effort in data preparation is needed. This involves a variety of datasets, the scale of the datasets. It is also important to study the quality of the datasets to ensure their suitability for such applications.

6.5 Geospatial Data Generator

The geospatial data generator is the last component in this prototype. First, we design a denoising UNet, trained following a diffusion process. Then, we discuss how this model is used to generate the histograms. Finally, we discuss the process for converting the histograms to geospatial data.

6.5.1 Model architecture and training

Since we represent the data as histograms, we can use vision-based approaches to design this model. Typically, a UNet architecture is used to generate images. This architecture takes images as inputs and produces images as outputs. It is comprised of multiple convolutional down blocks that encode the input, followed by multiple convolutional up-blocks for decoding the output of the down-blocks and producing images in the desired shape. When working with multiple modalities, such as generating images from text, text encodings are added to the encoded input after the down-blocks. This conditions the model to generate the desired output. Furthermore, the model is trained following a diffusion process [192]. In this way, the model is not trained to generate the desired output in one step, but following an iterative process. In each step, the model takes some noisy input and is trained to predict the noise that is then subtracted from the noisy input. In the context of geospatial data, there are two choices that we are making.

The first choice involves the choice of histogram representation. In this case, we represent the histogram in two layers. The first layer is a mask with zero/one values indicating whether there are points in the corresponding pixel or not. The second layer is

the actual point count in the corresponding pixel. The reason we make this choice is that the histogram mask helps in discarding noisy pixels in the final generated histogram, as we will discuss later. One alternative to this approach is to divide this into two separate models. The first model is used to only estimate the histogram mask, which can use a Bernoulli diffusion process, which is more suitable for discrete data. Then, the output mask can be used as an augmentation image in the model that generates the histogram.

The second design choice is related to the region context. In this case, it is provided in two ways. First, by providing a raster image augmented with the noisy input histogram. In this case, we used the land cover image, but we also tested with a 3-channel satellite image of the region. Second, the text encodigns also have information about the region.

Figure 6.5 shows the architecture of this model and an example of one training step. In each training step, the noise generator selects a random noise step to add to the input. In this case, $t = 0$ is the original input without any noise added, and $t = 1000$ means the input is complete Gaussian noise. In this step, a random time step $(t + 1)$ is selected, and the noisy example is used as the input. The model is trained to predict the noise that is subtracted from $H_{(t+1)}$ in order to move it to the less noisy example H_t. The noise is not added to the land cover image. Doing this for many iterations results in a model that is good at removing the noise given the conditioning text and the augmented images.

The generation process goes through a full loop of denoising. Every sample starts at step $t = 1000$, which is complete noise, and is gradually de-noised to step $t = 0$, which is the final output histogram.

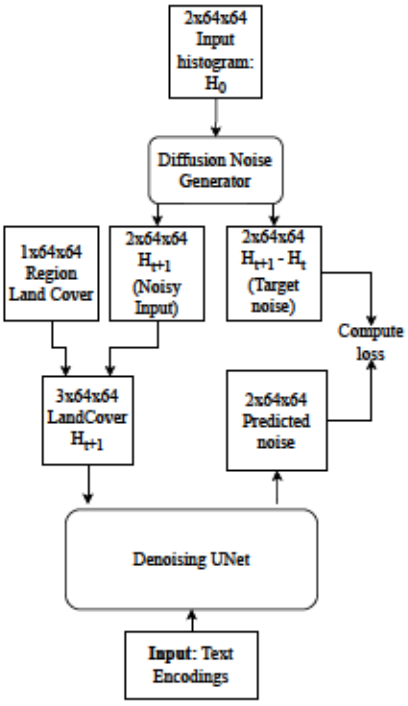


Figure 6.5: Denoising UNet Architecture

6.5.2 Histogram to geospatial points

Given the generated histograms, we want to convert them to their final form as geospatial points in the correct reference space. To do this, first, we do some pre-processing to clean the histogram and remove some noise based on the mask. Then, we scale the histogram and generate the points using a Gaussian generator given a desired cardinality and the density in each histogram cell. Finally, the generated points are translated to the correct reference space given the MBR of the region. This process is shown in Algorithm 4. Since the model produces continuous values, we first use the threshold to convert the histogram mask to binary values. Higher thresholds can be used when we are more sensitive to noise. By default, we set the threshold value to 0.5. Then, all the values in the count histogram are set to zero where the mask is zero. After that, the count histogram is converted to a density

histogram. We do this to control the cardinality of the data which is the parameter k in this procedure. Then, a Gaussian generator takes the density histogram and the cardinality k to produce k points following the density histogram. This is described in Algorithm 5, which generates points around the center of each histogram cell based on its density. These points are then translated to the same reference space as their region using the MBR of the region. This is the final step that completes the pipeline from text description to finally generated points based on the description and on the desired geographic region.

Algorithm 4 ProcessHistogram

Require: *mask*: histogram mask
Require: *hist*: count histogram
Require: t: mask threshold
Require: k: total number of points to generate
Require: *mbr*: minimum bounding rectangle for mapping

procedure ProcessHistogram(*mask*, *hist*, t, k, *mbr*)
    $mask \leftarrow (mask > t)$
    $hist[mask == 0] \leftarrow 0$
    $s \leftarrow \sum hist$
    $density \leftarrow$ **if** $s > 0$ **then** $hist/s$ **else** 0
    $pts \leftarrow$ GaussianGenerator$(density, k)$
    return TranslateToCoordinates(pts, mbr)

Algorithm 5 GaussianGenerator

Require: *density*: density histogram
Require: k: total number of points to generate

procedure GaussianGenerator(*density*, k)
    $points \leftarrow [\,]$
    $counts \leftarrow density \times k$
    for all (i, j) in *density* **do**
        $n \leftarrow \text{round}(counts[i][j])$
        for $l = 1$ to n **do**
            $x \leftarrow \text{Normal}(i + 0.5, \sigma)$
            $y \leftarrow \text{Normal}(j + 0.5, \sigma)$
            $points.\text{append}((x, y))$
    return *points*

6.6 Experiments

In this section, we perform a set of experiments to evaluate the different components in this prototype and evaluate some of the design choices, as well as to highlight future research directions.

6.6.1 Setup

Hardware

We used a server with *AMD EPYC 7543 32-Core Processor* with a total of 64 threads, 256GB of memory and an *NVIDIA A100-SXM4-80GB* GPU.

Datasets

We prepare four datasets based on the ebird dataset, buildings, and two subsets from points of interest in OpenStreetMap, including power lines and traffic signals. We only extract data within the boundaries of California. For the region boundaries, we use the boundaries of administrative regions, like cities, counties, etc. This is summarized in Table 6.1. For the text descriptions, we used a key-value pair JSON representation including the region name and the dataset. For the region context, we considered two alternatives: satellite images with three RGB channels and land-cover images that only contain one channel.

Table 6.1: Prepared Data Count

Dataset	Training	Validation	Total
Buildings	682	164	846
Traffic Signals	578	143	721
Power Lines	500	119	619
Bird Observations	682	151	833
Total	2442	577	

Models

First, we train the encoder models: the text encoder (TE), and the geospatial encoder (GE). The encoders are designed similar to ALIGN [93]. Then, we train three different variations of the denoising UNet. All UNet variations have the same architecture but different inputs. This is summarized as follows:

- **TE**: a text encoder that takes the descriptions of the geospatial data, which we represent as key-value pairs.
- **GE**: a geospatial encoder (convolutional neural network) that takes a raster image and a histogram of geospatial data.
- **U1**: a UNet model that is conditioned on embeddings from our text-encoder (TE) which represent the prompt, and the input histogram is augmented with a raster RGB image.
- **U2**: a UNet model that is conditioned on embeddings from our text-encoder (TE) , and the input histogram is augmented with a land-cover image.
- **U3**: a UNet model that is conditioned on embeddings from the pre-trained ALIGN text encoder, and the input is just the histogram without augmentation.

Next, we go over a set of evaluations. We start by performing a qualitative evaluation where we look at the generated data by all the models. Then, we perform a quantitative evaluation to evaluate the generated data using some quality metrics. Finally, we evaluate the encoders and the quality of the embeddings that they generate.

6.6.2 Qualitative evaluation

We perform a qualitative evaluation by observing how the generated data by all the models compares with data from the ground truth. We show an example for each dataset for each model in Figure 6.6. The blue circles represent data from the ground truth for the text used in the prompt. The red circles represent the data generated by the model. All of these are examples from the validation set. In general, we notice that all three models U1, U2, U3, can differentiate between the four different datasets. For example, power lines tend to follow some line patterns, traffic signals tend to follow a grid-like pattern, while bird observations tend to be more sporadic, and buildings tend to be clustered and dense. However, we notice that the generated data doesn't follow some low details, such as not generating buildings in water areas or generating the traffic signals around road intersections. Furthermore, all models can generate noisy examples. However, we find that the quality of the data is sufficient enough for generating synthetic data for data augmentation for different tasks. Some of the noisy examples can be discarded using some of the quality metrics. We later discuss potential directions for future improvements in Section 6.6.5.

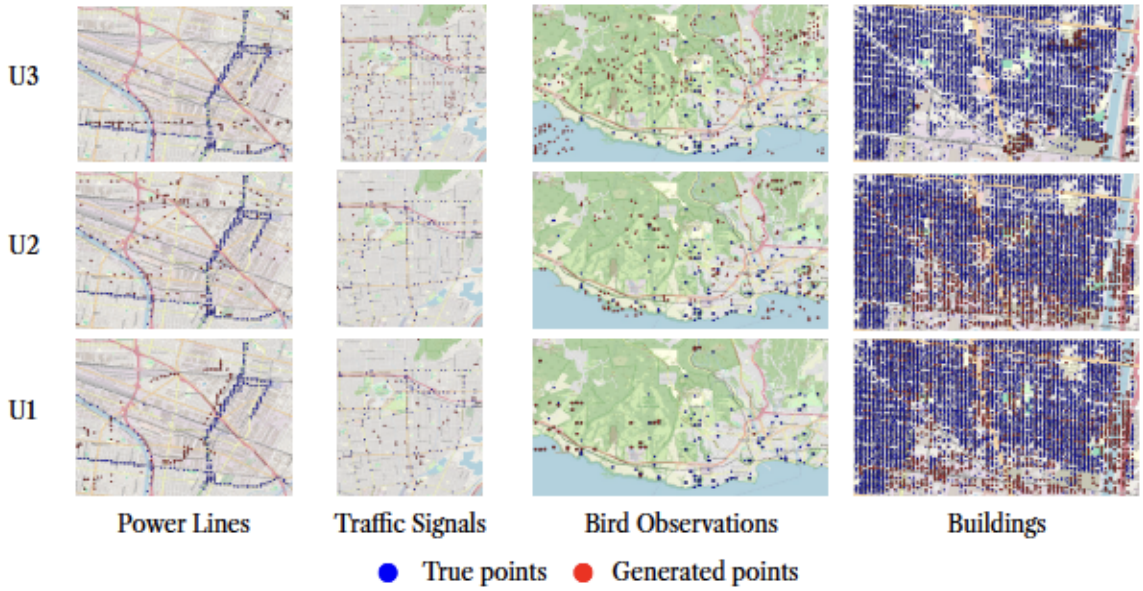


Figure 6.6: Showing one example for generated data for each dataset by each model

6.6.3 Quantitative evaluation

Next, we perform a quantitative evaluation to compare the three UNet models by comparing the generated data for each dataset to the ground truth datasets.

Kernel Geospatial Embedding Distance (KGD)

This evaluation is based on the Kernel Inception Distance (KID) [38], which is used to evaluate the similarity between images based on their embeddings from the Inception [195] image classification model. However, we cannot use such a model to get embeddings for our geospatial data. Instead, we can use the embeddings from our Geospatial Encoder model. This metric includes using a polynomial kernel that takes two embeddings and generates some value. Then, the expected value for this kernel among all pairs in the real data is computed, followed by the expected value for all pairs in the generated data, and finally

the expected value between every real and generated pair. This is shown in Equation 6.4, where x,x' are for real data, and y,y' are for generated data. For this score, lower values are better, indicating more similar distributions. We used this metric because it does not require a large sample size and our validation set for every dataset includes only a few hundred examples.

$$\text{KID} = E_{x,x'}[k(x, x')] + E_{y,y'}[k(y, y')] - 2E_{x,y}[k(x, y)] \tag{6.4}$$

The results for the evaluation using this metric are provided in Table 6.2. The way to read these values is that for each true dataset, its corresponding generated dataset should have the closest score to zero compared to the other generated datasets. Based on this, we can see that only two cases match this criteria. Possible explanations for the closeness of the scores between the different distributions could include the presence of noisy examples in all datasets and the sparse nature of the input datasets.

Geospatial Embedding Based Evaluation

In this evaluation, we use the embeddings of the prompts from the text encoder (TE) and the embeddings of the generated data from the geospatial encoder (GE). This is similar to an evaluation based on the CLIP Similarity [88] score. This evaluation is more easier to interpret compared to the previous one. We evaluate based on the recall based on the closest matching embeddings. For example, given an embedding for a generated histogram for a given prompt, if its closest text embedding is from the same dataset, it will be counted for the score R@1, meaning recall at one. Similarly, R@5 means that one of the

Table 6.2: Comparison of KGD scores by dataset

True Dataset	Gen Dataset	M1	M2	M3
building	building	-0.0195	-0.0138	**0.0018**
building	ebird	0.0291	-0.0018	0.0169
building	power	-0.0222	0.0187	0.0093
building	signal	-0.0030	0.0199	0.0128
ebird	building	0.0065	0.0087	0.0196
ebird	ebird	0.0077	0.0083	**-0.0014**
ebird	power	0.0076	0.0151	-0.0043
ebird	signal	-0.0004	-0.0039	-0.0202
power	building	-0.0056	-0.0088	0.0086
power	ebird	0.0366	0.0131	0.0360
power	power	0.0167	-0.0101	-0.0048
power	signal	0.0104	-0.0135	0.0009
signal	building	0.0071	0.0086	0.0240
signal	ebird	-0.0010	-0.0103	-0.0056
signal	power	0.0047	0.0124	0.0002
signal	signal	0.0116	-0.0182	-0.0052

top five closest text embeddings are from the same data. We also use the Mean Reciprocal Rank (MRR), shown in Equation 6.5.

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{\text{rank}_i} \tag{6.5}$$

The results for this evaluation are shown in Table 6.3. The scores are close to the MRR we get when evaluating the embeddings of training data from the encoder that we discuss later in Section 6.6.4, which is around 70%. This result also shows that M3 is slightly better than M1 and M2. This also corresponds to our observations when performing the qualitative evaluation, although all three models perform somewhat similarly. These results show a relation between the quality of the generated data and the quality of the encoders. We evaluate the encoders in more details next.

Table 6.3: Cross-modal retrieval performance generated data histograms to validation data texts. Dataset matching only.

Model	R@1	R@5	R@10	R@50	MRR
M1	0.5355	0.9324	0.9844	1.0000	0.6984
M2	0.5130	0.9307	0.9913	1.0000	0.6870
M3	0.5962	0.9393	0.9827	1.0000	0.7419

6.6.4 Evaluation of encoder

In this section, we evaluate the two encoder models TE and GE. First, we evaluate the cross-modal retrieval between the image embeddings and the text embeddings. In this evaluation, both the location and the dataset must be matching between the geospatial embedding and the text embedding. We show this evaluation in Table 6.4. We compare our model to pre-trained models for images. Since the pre-trained models are not trained on similar data, they don't work on relating the histograms and the texts. However, our model can relate the two modalities, with 0.89 MRR for the validation data. To learn more about if our model gives more focus to the location as compared to the dataset, we divide the evaluation into two parts. First, we evaluate by matching only on the dataset, and we show the results in Table 6.5. The results for this comparison show similar values for the R@1. The pre-trained model shows values close to a random recall for R@1 which is 25%, since we have four datasets. Second, we evaluate by matching only on the region, and we show the results in Table 6.6. This result shows much higher values compared to the combined evaluation. This indicates that encoders focus more on the region labels as compared to the datasets. There could be many reasons for explaining this behavior. One main reason could be related to the quality and size of the datasets that we use.

Finally, we evaluate the text encoder only. We compare the embeddings of the texts from the validation set to the embeddings in the training set. This evaluation is shown in Table 6.7. While our model worked well in associating the geospatial embeddings to the text embeddings, the text encoder by itself does not work well for associating similar texts between the training and validation data. However, one thing we note is that all the combinations of regions and datasets in the validation data are not seen in the training. For each region, the model only sees a few of the datasets, and the others are kept for training, so the combinations are mutually exclusive. The pre-trained text-encoder models can associate well the dataset labels but not the region labels, since the dataset labels are also common in image datasets, like birds and building, etc.

Table 6.4: Cross-modal retrieval results for datasets & regions

Model	**Dataset**	**R@1**	**R@5**	**R@10**	**R@50**	**MRR**
Our Model	Train	0.4959	0.9967	1.0000	1.0000	0.7129
	Val	0.8146	0.9879	0.9879	0.9896	0.8974
ALIGN	Train	0.0000	0.0020	0.0037	0.0258	0.0035
	Val	0.0017	0.0121	0.0243	0.1127	0.0139
CLIP	Train	0.0008	0.0041	0.0057	0.0299	0.0049
	Val	0.0017	0.0156	0.0225	0.0971	0.0136

Table 6.5: Cross-modal retrieval results for dataset only

Model	**Dataset**	**R@1**	**R@5**	**R@10**	**R@50**	**MRR**
Our Model	Train	0.5057	1.0000	1.0000	1.0000	0.7203
	Val	0.8198	0.9948	0.9965	1.0000	0.9029
ALIGN	Train	0.2068	0.4717	0.6167	0.9828	0.3315
	Val	0.2166	0.6499	0.9497	1.0000	0.4188
CLIP	Train	0.3493	0.5762	0.6691	0.9496	0.4589
	Val	0.2877	0.4437	0.6603	1.0000	0.3816

Table 6.6: Cross-modal retrieval results for regions only

Model	Dataset	R@1	R@5	R@10	R@50	MRR
Our Model	Train	0.9959	1.0000	1.0000	1.0000	0.9977
	Val	0.9827	0.9879	0.9879	0.9896	0.9849
ALIGN	Train	0.0004	0.0078	0.0152	0.0737	0.0089
	Val	0.0017	0.0173	0.0364	0.1542	0.0181
CLIP	Train	0.0016	0.0127	0.0197	0.0786	0.0117
	Val	0.0017	0.0208	0.0364	0.1490	0.0184

Table 6.7: Validation text embedding compared with training text embeddings

Model	Retrieval Type	R@1	R@5	R@10	R@50	MRR
Our Model	Datasets Only	0.1724	0.7502	0.9386	1.0000	0.4091
	Regions Only	0.4189	0.4238	0.4259	0.4263	0.4212
ALIGN	Datasets Only	0.8452	0.9996	1.0000	1.0000	0.9188
	Regions Only	0.1511	0.2179	0.2359	0.2735	0.1833
CLIP	Datasets Only	0.9689	1.0000	1.0000	1.0000	0.9830
	Regions Only	0.0192	0.0622	0.1061	0.2514	0.0465

6.6.5 Discussion

In this section, we discuss the implications of these results and possible future directions.

Dataset preparation

The datasets that we prepared showed clearly distinctive behavior, and the models that we trained can differentiate between them. However, there are still many challenges associated with dataset quality especially when working with datasets from multiple sources. For example, it is challenging to assess that there is a strong correlation between the datasets themselves and the attributes used for creating the text labels. Furthermore, when working with multiple datasets, it can be challenging to ensure that patterns in some datasets have similar behavior in another dataset that was labeled differently. In our case, we looked manually at different datasets and selected ones that clearly show distinctive behavior.

However, this can be challenging to verify at scale. For example, if we partitioned the bird observations dataset by bird species, it is too challenging to verify that the data in each partition shows distinctive behavior. Sometimes distinctive behaviors can only be observed at a higher level and not at the city or neighborhood level. This brings us to another aspect, which is the resolution of the data. In our prototype, we defined the regions based on administrative regions and scaled all of them to 64 by 64 images. A more general approach should probably use a fixed aspect ratio and incorporate different resolutions, while the histograms are built for each dataset based on its appropriate resolution. A more general approach should probably use a fixed aspect ratio, and incorporate different resolutions, while the histograms are built for each dataset based on its appropriate resolution.

Geospatial encoder

The encoders that we trained can associate between the geospatial data embeddings with the embeddings of their text descriptions. The detailed evaluation showed that the embeddings tend to give higher weights on the regional context data as opposed to the geospatial data description. This could also be due to the dataset size and quality discussed earlier. It could also be related to how the data is partitioned since the combinations of text descriptions in the training are different from those in the validation. We still think that multi-modal encoders such as these have many promising applications for geospatial data retrieval and other applications.

Data generation

The data generator model can differentiate between the different datasets and produce similar distributions. The quality of the produced images is sufficient for data augmentation. However, the models do not pick up on small details like generating points for buildings on top of water areas. One potential way to address this is to use masking augmentations to guide the training. For example, we can provide a mask for the road network when generating traffic data or power lines. Or alternatively, provide a mask for other types of land cover, like urban areas for buildings. There are many works for generating data at a specific location in an image using a masking layer [27] is one such example. While we used land cover for augmentation which contains this information, perhaps it is too general, and the model did not pick up on those patterns. It could also be that our dataset is not rich enough to enable the model to pick up on those small details. Furthermore, we only considered histograms of size 64x64. This is perhaps not sufficient to produce high-quality histograms. For image generators, typically there is a set of models, where the first model produces a low-resolution image, and the following models resize the image and increase its clarity. We leave the investigation for this for future work.

6.7 Related Work

Several works have explored image generation from text, including DALL-E2 [172], which proposed a diffusion prior step that translates text embeddings to image embeddings, resulting in higher quality images. Also, Imagen [179] provided state of the art image generation and showed that larger language models for text embeddings result in better

images. Others focused on specific cases like SpaText [27], which focused on generating images based on textual descriptions and specific regions of existing images. Such work can be useful in geospatial data generation, such as generating data around the boundaries of objects like roads. A comprehensive survey presented in [223] covers diffusion models for various data generation tasks such as text-to-image and text-to-video; however, discussions about geospatial applications are still limited.

Some of the recent efforts in geospatial data generation have focused on producing raster data using diffusion-based models. MESA [43] uses a diffusion model trained on global remote sensing data to synthesize high-quality terrain from textual descriptions. DiffusionSat [109] similarly generates satellite imagery from text inputs, incorporating additional conditioning information such as geographic coordinates, timestamps, and sensor specifications. Moreover, RSVQ-Diffusion [75] leverages a diffusion-based architecture to produce satellite images conditioned on textual descriptions. SatSynth [203] generates synthetic satellite images, with a segmentation mask, and highlights the benefits of data augmentation. Other works focused on more application-specific generation. These include SEEDS [129] for weather forecasting, and [118] for rain prediction. MapGen-Diff [201] takes remote sensing images and produces a corresponding map image, and similarly [167].

Some works have discussed the vision for geospatial foundation models [220]. We think the work in this paper takes us a step closer to realizing these visions, including highlighting all the current challenges. Some works have discussed more details like privacy implications when building geospatial foundation models [173].

For works related to geospatial data embeddings, one notable work is SatCLIP [171] which is similar to CLIP [171]. However, this model is only trained on satellite images for the image encoder, and the other encoder only takes the latitude and longitude of the center of the region. It is not related to geospatial vector data and does not incorporate rich textual descriptions. Existing encoders for geospatial vector data typically are application-specific. These include: KnowSite [140], for embedding urban knowledge graphs, and T-JEPA [128], which provides a self-supervised trajectory embedding model, among others.

6.8 Conclusion

We introduced GeoGen I, a framework for generating geospatial point data from text prompts. The models generate data that is distinctive and showing similar behavior to real data. However, several limitations remain. First, dataset preparation is still challenging, like verifying that partitions exhibit distinctive patterns, especially when relying on loosely labeled data. Second, the model resolution is low (64×64), which restricts the detail it can capture. This affects the generator's ability to avoid errors like placing buildings over water. Third, the encoders seem to rely more heavily on regional context than the text descriptions, which may reflect shortcomings in the training data or the encoder alignment.

Future work should focus on improving dataset quality and label consistency, incorporating multi-resolution generation, and exploring masking techniques to enforce spatial constraints. These directions could make the system more robust and allow it to generate higher-quality spatial data suitable for a wider range of applications.

Chapter 7

Conclusions

This thesis presented a set of systems and models aimed at improving the storage, processing, and generation of large-scale geospatial vector data. Each chapter addressed a specific limitation in the current ecosystem, offering practical solutions that scale and generalize.

SpatialParquet introduced a columnar file format tailored for geospatial data, demonstrating how the column representation enabled significant data savings through compression and encodings like FP-delta. The data statistics stored with the column also enable more efficient querying. dsJSON tackled the long-standing inefficiency of distributed JSON processing by integrating selective parsing, robust partitioning, and schema inference directly into Spark. It filled a critical gap in enabling scalable analytics over semi-structured data. While we made some progress, support for geospatial data in data lakes is still lacking, especially with new innovations such as data lake formats like Apache Iceberg, which incorporate storing more statistics and enable more efficient querying.

In deep learning for geospatial vector data, we showed that the three modeling approaches, GeoImg, GeoGraph, and GeoVec, are viable, but come with trade-offs in generalization and scalability. Future works can focus on providing more scalable alternatives, solutions for more complex spatial data such as complex lines and polygons, and also solutions for integrating vector and raster data. The GS-QA benchmark highlighted major deficiencies in current LLM-based systems when dealing with spatial reasoning. It also outlined several areas where more robust evaluation and retrieval strategies are necessary. Finally, GeoGen I demonstrated the feasibility of generating geospatial point data from text but exposed clear limitations in data curation and quality, as well as challenges related to resolution and location context. For future research, focus can be given to data curation and quality assessment, and also generation based on masked regions, to guide the generator to specific parts of the region, such as generating traffic signals around road intersections.

Together, the systems and experiments in this thesis make some progress in providing more support for geospatial data in data lakes and deep learning applications; however, significant open problems remain.

Bibliography

[1] Json encoder and decoder. Available at `https://docs.python.org/3/library/json.html`.

[2] MongoDB. Available at `https://www.mongodb.com`.

[3] Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.

[4] Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[5] Bestbuy developer api, 2021. Retrieved from `https://bestbuyapis.github.io/api-documentation/`.

[6] Jackson, 2021. Available at `https://github.com/FasterXML/jackson`.

[7] Jayway JsonPath, 2021. Available at `https://github.com/json-path/JsonPath`.

[8] RapidJSON, 2021. Available at `https://rapidjson.org/`.

[9] Wikipedia json dumps, 2021. Retrieved from `https://dumps.wikimedia.org/wikidatawiki/latest/`.

[10] geo-arrow-spec, June 2022. original-date: 2020-04-06T21:00:33Z.

[11] Parquet, June 2022. original-date: 2014-06-10T07:00:07Z.

[12] GPT-4o System Card, August 2024.

[13] IBM-NASA Prithvi Models Family (ibm-nasa-geospatial), December 2024.

[14] llama3.2, July 2024.

[15] Nominatim: Open source geocoding with openstreetmap data, 2024.

[16] chroma-core/chroma, February 2025. original-date: 2022-10-05T17:58:44Z.

[17] Sushila Aghav. Database compression techniques for performance optimization. In *ICCET*, volume 6, pages V6–714–V6–717, 2010.

[18] Isam Mashhour Al Jawarneh, Paolo Bellavista, Antonio Corradi, Luca Foschini, and Rebecca Montanari. Locality-preserving spatial partitioning for geo big data analytics in main memory frameworks. In *2020 IEEE Global Communications Conference*, pages 1–6, 2020.

[19] Abdullah Al-Mamun, Hao Wu, and Walid G. Aref. A tutorial on learned multi-dimensional indexes. In *SIGSPATIAL*, pages 1–4, Seattle, WA, Nov 2020. ACM.

[20] Elie Aljalbout, Vladimir Golkov, Yawar Siddiqui, Maximilian Strobel, and Daniel Cremers. Clustering with Deep Learning: Taxonomy and New Methods, 2018. arXiv:1801.07648 [cs].

[21] Wail Y. Alkowaileet, Sattam Alsubaiee, Michael J. Carey, Till Westmann, and Yingyi Bu. Large-scale complex analytics on semi-structured datasets using asterixdb and spark. *Proc. VLDB Endow.*, 9(13):1585–1588, sep 2016.

[22] Sattam Alsubaiee1 Yasser Altowim1 Hotham Altwaijry, Alexander Behm, Vinayak Borkar1 Yingyi Bu1 Michael Carey, Inci Cetindil1 Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova1 Raman Grover1 Zachary Heilbron, Pouria Pirzadeh1 Vassilis Tsotras7 Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source bdms. *Proceedings of the VLDB Endowment*, 7(14), 2014.

[23] Hossein Amiri, Will Kohn, Shiyang Ruan, Joon-Seok Kim, Hamdi Kavak, Andrew T. Crooks, Dieter Pfoser, Carola Wenk, and Andreas Züfle. The patterns of life human mobility simulation. In Mario A. Nascimento, Li Xiong, Andreas Züfle, Yao-Yi Chiang, Ahmed Eldawy, and Peer Kröger, editors, *Proceedings of the 32nd ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2024, Atlanta, GA, USA, 29 October 2024 - 1 November 2024*, pages 653–656. ACM, 2024.

[24] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[25] Sören Auer, Dante A. C. Barone, Cassiano Bartz, Eduardo G. Cortes, Mohamad Yaser Jaradeh, Oliver Karras, Manolis Koubarakis, Dmitry Mouromtsev, Dmitrii Pliukhin, Daniil Radyush, Ivan Shilin, Markus Stocker, and Eleni Tsalapati. The SciQA Scientific Question Answering Benchmark for Scholarly Knowledge. *Scientific Reports*, 13(1):7240, May 2023.

[26] Woody Austin, Grey Ballard, and Tamara G. Kolda. Parallel tensor compression for large-scale scientific data. In *IPDPS*, pages 912–922, 2016.

[27] Omri Avrahami, Thomas Hayes, Oran Gafni, Sonal Gupta, Yaniv Taigman, Devi Parikh, Dani Lischinski, Ohad Fried, and Xi Yin. SpaText: Spatio-Textual Representation for Controllable Image Generation . In *2023 IEEE/CVF Conference on*

Computer Vision and Pattern Recognition (CVPR), pages 18370–18380, Los Alamitos, CA, USA, June 2023. IEEE Computer Society.

[28] Mohamed-Amine Baazizi, Clément Berti, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Human-in-the-Loop Schema Inference for Massive JSON Datasets. In *EDBT 2020 - 23nd International Conference on Extending Database Technology*, pages 635–638, Copenhagen, Denmark, March 2020. OpenProceedings.org.

[29] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive json datasets. *The VLDB Journal*, 28(4):497–521, 2019.

[30] Furqan Baig, Chao Gao, Dejun Teng, Jun Kong, and Fusheng Wang. Accelerating spatial cross-matching on cpu-gpu hybrid platform with cuda and openacc. *Frontiers in Big Data*, 3, 2020.

[31] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Science of Computer Programming*, 112:195–226, 2015.

[32] Norbert Beckmann and Bernhard Seeger. A benchmark for multidimensional index structures, 2008.

[33] Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. Detecting skewness of big spatial data in spatialhadoop. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '18, page 432–435, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. A generic machine learning model for spatial query optimization based on spatial embeddings. *ACM Trans. Spatial Algorithms Syst.*, 10(4), October 2024.

[35] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Mohamed Eltabakh, and Andrey Balmin. Jaql: A scripting language for large scale semistructured data analysis. vldb, 2011.

[36] Derya Birant and Alp Kut. ST-DBSCAN: An algorithm for clustering spatial–temporal data. *Data & Knowledge Engineering*, 60(1):208–221, January 2007.

[37] Enam Biswas. Imdb review dataset, 2021. Retrieved from `https://www.kaggle.com/dsv/1836923`.

[38] Mikołaj Bińkowski, Dougal J Sutherland, Michael Arbel, and Arthur Gretton. Demystifying mmd gans. In *International Conference on Learning Representations (ICLR)*, 2018.

[39] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: Time series compression for the internet of things. *IMWUT*, 2(3):1–23, 2018.

[40] Martin Böckling, Heiko Paulheim, and Sarah Detzler. A planet scale spatial-temporal knowledge graph based on openstreetmap and h3 grid. *arXiv preprint arXiv:2405.15375*, 2024.

[41] Geoff Boeing. Modeling and Analyzing Urban Networks and Amenities with OSMnx, 2024.

[42] Daniele Bonetta and Matthias Brantner. Fad. js: fast json data access using jit-based speculative optimizations. *Proceedings of the VLDB Endowment*, 10(12):1778–1789, 2017.

[43] Paul Borne-Pons, Mikolaj Czerkawski, Rosalie Martin, and Romain Rouffet. Mesa: Text-driven terrain generation using latent diffusion and global copernicus data, 2025.

[44] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[45] Eren Cakmak, Manuel Plank, Daniel S. Calovi, Alex Jordan, and Daniel Keim. Spatio-temporal clustering benchmark for collective animal behavior. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Animal Movement Ecology and Human Mobility*, HANIMOB '21, page 5–8, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Eugenio Cesario, Paolo Lindia, and Andrea Vinci. A scalable multi-density clustering approach to detect city hotspots in a smart city. *Future Generation Computer Systems*, 157:226–236, 2024.

[47] Tsz Nam Chan, Leong Hou U, Byron Choi, Jianliang Xu, and Reynold Cheng. Large-scale geospatial analytics: Problems, challenges, and opportunities. In *Companion of the 2023 International Conference on Management of Data*, SIGMOD '23, page 21–29, 2023.

[48] Tsz Nam Chan, Leong Hou U, Yun Peng, Byron Choi, and Jianliang Xu. Fast network k-function-based spatial analysis. *Proc. VLDB Endow.*, 15(11):2853–2866, 2022.

[49] Ashesh Chattopadhyay, Pedram Hassanzadeh, and Saba Pasha. Predicting clustered weather patterns: A test case for applications of convolutional neural networks to spatio-temporal climate data. *Scientific Reports*, 10(1):1317, 2020.

[50] Jiaqi Chen, Jianheng Tang, Jinghui Qin, Xiaodan Liang, Lingbo Liu, Eric Xing, and Liang Lin. GeoQA: A Geometric Question Answering Benchmark Towards Multi-modal Numerical Reasoning. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 513–523, Online, August 2021. Association for Computational Linguistics.

[51] Wei Chen, Eric Fosler-Lussier, Ningchuan Xiao, Satyajeet Raje, Rajiv Ramnath, and Daniel Sui. A Synergistic Framework for Geographic Question Answering. In *2013 IEEE Seventh International Conference on Semantic Computing*, pages 94–99, Irvine, CA, USA, September 2013. IEEE.

[52] Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. TheoremQA: A theorem-driven question answering dataset. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7889–7901, Singapore, December 2023. Association for Computational Linguistics.

[53] Eunsol Choi, He He, Mohit Iyyer, Mark Yatskar, Wen-tau Yih, Yejin Choi, Percy Liang, and Luke Zettlemoyer. QuAC: Question answering in context. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2174–2184, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[54] Steven Claggett, Sahar Azimi, and Martin Burtscher. Spdp: An automatically synthesized lossless compression algorithm for floating-point data. In *2018 Data Compression Conference*, pages 335–344, 2018.

[55] James Clark, Steve DeRose, et al. Xml path language (xpath), 1999.

[56] Pavel Čontoš and Martin Svoboda. Json schema inference approaches. In Georg Grossmann and Sudha Ram, editors, *Advances in Conceptual Modeling*, pages 173–183, Cham, 2020. Springer International Publishing.

[57] Data.Gov: The home of the U.S. Government's open data, June 2022.

[58] Peter Deutsch. Gzip file format specification version 4.3. Technical report, 1996.

[59] Sheng Di, Dingwen Tao, Xin Liang, and Franck Cappello. Efficient lossy compression for scientific data based on pointwise relative error bound. *TPDS*, 30(2):331–345, 2019.

[60] Michael DiScala and Daniel J Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 295–310, 2016.

[61] Harish Doraiswamy, Eleni Tzirita Zacharatou, Fabio Miranda, Marcos Lage, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. Interactive visual exploration of spatio-temporal urban data sets using urbane. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD, pages 1693–1696. ACM, 2018.

[62] Gabriele D'Orso and Marco Migliore. A GIS-based method for evaluating the walkability of a pedestrian environment and prioritised investments. *Journal of Transport Geography*, 82:102555, 2020.

[63] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[64] Dominik Durner, Viktor Leis, and Thomas Neumann. *JSON Tiles: Fast Analytics on Semi-Structured Data*, page 445–458. Association for Computing Machinery, New York, NY, USA, 2021.

[65] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial Partitioning Techniques in Spatial Hadoop. *PVLDB*, 8(12):1602–1605, 2015.

[66] Ahmed Eldawy et al. Beast: Scalable exploratory analytics on spatio-temporal data. In *CIKM*, pages 3796–3807, 2021.

[67] Markus Endres, Asha Mannarapotta Venugopal, and Tung Son Tran. Synthetic data generation: A comparative study. In *Proceedings of the 26th International Database Engineered Applications Symposium*, IDEAS '22, page 94–102, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[69] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *PVLDB*, 7(12), 2014.

[70] Christian A. Flores-Villamil, Huizilopoztli Luna-García, Mauricio Ramírez-Villegas, Carlos H. Espino-Salinas, Alejandro Mauricio-González, and José G. Arceo-Olague. School Clustering Through Machine Learning and Geospatial Analysis. In *Geographical Information Systems*, pages 86–104, 2025.

[71] Nathaniel Fout and Kwan-Liu Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *TVCG*, 18(12):2295–2304, 2012.

[72] Luis Iribarne Francisco García-García, Antonio Corral and Michael Vassilakopoulos. Efficient distributed algorithms for distance join queries in spark-based spatial analytics systems. *International Journal of General Systems*, 52(3):206–250, 2023.

[73] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 519–530, New York, NY, USA, 2015. Association for Computing Machinery.

[74] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145, January 2024.

[75] Xin Gao, Yao Fu, Xiaonan Jiang, Fanlu Wu, Yu Zhang, Tianjiao Fu, Chao Li, and Junyan Pei. Rsvq-diffusion model for text-to-remote-sensing image generation. *Applied Sciences*, 15(3), 2025.

[76] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. Speculative distributed csv data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*, pages 883–899, 2019.

[77] Geofabrik Download Server, 2024.

[78] GeoParquet: Store Vector Data in Apache Parquet, June 2022.

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

[80] Stefan Goessner. JSONPath - XPath for JSON, February 2007. Available at `https://goessner.net/articles/JsonPath/`.

[81] Mandeep Goyal and Qusay H. Mahmoud. A systematic review of synthetic data generation techniques using generative ai. *Electronics*, 13(17), 2024.

[82] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata. In *International Conference on Database Theory*, pages 173–189. Springer, 2003.

[83] Peiqin Gu, Zhiyuan Han, Zhejing Cao, Yulin Chen, and Yang Jiang. Using Open Source Data to Measure Street Walkability and Bikeability in China: A Case of Four Cities. *Transportation Research Record: Journal of the Transportation Research Board*, 2672(31):63–75, 2018.

[84] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. Deep learning models for selectivity estimation of multi-attribute queries. In *Proceedings of the 2020 ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD '20, page 1035–1050, 2020.

[85] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 770–778, 2016.

[86] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[87] Nicolaus Henke, Jacques Bughin, Michael Chui, James Manyika, Tamim Saleh, Bill Wiseman, and Guru Sethupathy. The Age of Analytics: Competing in a Data-driven World. Technical report, McKinsey Global Institute, 2016.

[88] Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. Clipscore: A reference-free evaluation metric for image captioning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[89] Chi Ho, Bill Yuchen Lin, Xiang Ren Chen, and Xiang Ren. Constructing multi-hop knowledge paths for complex question answering over knowledge bases. In *Proceedings of the 28th International Conference on Computational Linguistics (COLING)*, pages 6302–6318, 2020.

[90] Alexander Hohl, Minrui Zheng, Wenwu Tang, Eric Delmelle, and Irene Casas. Spatiotemporal Point Pattern Analysis Using Ripley's K Function. In *Geospatial Data Science Techniques and Applications*. CRC Press, 2017.

[91] Ionuţ Hrubaru, George Talabă, and Marin Fotache. A basic testbed for json data processing in sql data servers. In *Proceedings of the 20th International Conference on Computer Systems and Technologies*, CompSysTech '19, page 278–283, New York, NY, USA, 2019. Association for Computing Machinery.

[92] Shang-Ling Hsu, Emmanuel Tung, John Krumm, Cyrus Shahabi, and Khurram Shafique. Trajgpt: Controlled synthetic trajectory generation using a multitask transformer-based spatiotemporal model. In Mario A. Nascimento, Li Xiong, Andreas Züfle, Yao-Yi Chiang, Ahmed Eldawy, and Peer Kröger, editors, *Proceedings of the 32nd ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2024, Atlanta, GA, USA, 29 October 2024 - 1 November 2024*, pages 362–371. ACM, 2024.

[93] Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc V. Le, Yunhsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language representation learning with noisy text supervision, 2021.

[94] Hao Jiang and Aaron J. Elmore. Boosting data filtering on columnar encoding with simd. In *DaMoN@SIGMOD*, 2018.

[95] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Scalable structural index construction for json analytics. *Proc. VLDB Endow.*, 14(4):694–707, dec 2020.

[96] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. Scalable processing of contemporary semi-structured data on commodity parallel processors-a compilation-based approach. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2019.

[97] Lin Jiang and Zhijia Zhao. Grammar-aware parallelization for scalable xpath querying. *ACM SIGPLAN Notices*, 52(8):371–383, 2017.

[98] Lin Jiang and Zhijia Zhao. Jsonski: streaming semi-structured data with bit-parallel fast-forwarding. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–211, 2022.

[99] JSON, 2021. Available at `https://www.json.org/`.

[100] Documentation for the json lines text file format, 2021. Available at `https://jsonlines.org`.

[101] Zihan Kan, Mei-Po Kwan, and Luliang Tang. Ripley's k-function for network-constrained flow data. *Geographical Analysis*, 54(4):769–788, 2022.

[102] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Practical string dictionary compression using string dictionary encoding. In *Innovate-Data*, 2017.

[103] Nikolaos Karalis, Georgios Mandilaras, and Manolis Koubarakis. Extending the yago2 knowledge graph with precise geospatial knowledge. In *The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II 18*, pages 181–197. Springer, 2019.

[104] Md Rezaul Karim, Oya Beyan, Achille Zappa, Ivan G Costa, Dietrich Rebholz-Schuhmann, Michael Cochez, and Stefan Decker. Deep learning-based clustering approaches for bioinformatics. *Briefings in Bioinformatics*, 22(1):393–415, 2021.

[105] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense Passage Retrieval for Open-Domain Question Answering, September 2020. arXiv:2004.04906 [cs].

[106] Puloma Katiyar, Tin Vu, Ahmed Eldawy, Sara Migliorini, and Alberto Belussi. Spiderweb: A spatial data generator on the web. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '20, page 465–468, New York, NY, USA, 2020. Association for Computing Machinery.

[107] Mohammad Kazemi Beydokhti, Matt Duckham, Amy L. Griffin, Yaguang Tao, Ross Purves, and Maria Vasardani. Probabilistic qualitative spatial reasoning with applications to GeoQA. *International Journal of Geographical Information Science*, pages 1–30, December 2024.

[108] Sergios-Anestis Kefalidis, Dharmen Punjani, Eleni Tsalapati, Konstantinos Plas, Maria-Aggeliki Pollali, Pierre Maret, and Manolis Koubarakis. The question answering system GeoQA2 and a new benchmark for its evaluation. *International Journal of Applied Earth Observation and Geoinformation*, 134:104203, November 2024.

[109] Samar Khanna, Patrick Liu, Linqi Zhou, Chenlin Meng, Robin Rombach, Marshall Burke, David Lobell, and Stefano Ermon. Diffusionsat: A generative foundation model for satellite imagery, 2024.

[110] Abdelouahab Khelifati, Mourad Khayati, and Philippe Cudré-Mauroux. Corad: Correlation-aware compression of massive time series using sparse dictionary coding. In *IEEE BigData*, pages 2289–2298, 2019.

[111] Konstantin Klemmer, Esther Rolf, Caleb Robinson, Lester Mackey, and Marc Rußwurm. Satclip: Global, general-purpose location embeddings with satellite imagery, 2024.

[112] Meike Klettke, Uta Störl, and Stefanie Scherzinger. Schema extraction and structural outlier detection for json-based nosql data stores. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[113] Fabian Knorr, Peter Thoman, and Thomas Fahringer. ndzip: A high-throughput parallel lossless compressor for scientific data. In *DCC*, pages 103–112, 2021.

[114] Anastasia Krithara, Anastasios Nentidis, Konstantinos Bougiatiotis, and Georgios Paliouras. BioASQ-QA: A manually curated corpus for Biomedical Question Answering. *Scientific Data*, 10(1):170, March 2023.

[115] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 28(6):941–960, 2019.

[116] Robert Lasch et al. Fast & strong: The case of compressed string dictionaries on modern cpus. In *DaMoN@SIGMOD*, 2019.

[117] Md Tahmid Rahman Laskar, Sawsan Alqahtani, M Saiful Bari, Mizanur Rahman, Mohammad Abdullah Matin Khan, Haidar Khan, Israt Jahan, Amran Bhuiyan, Chee Wei Tan, Md Rizwan Parvez, Enamul Hoque, Shafiq Joty, and Jimmy Huang. A systematic survey and critical review on evaluating large language models: Challenges, limitations, and recommendations. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13785–13816, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[118] Jussi Leinonen, Ulrich Hamann, Daniele Nerini, Urs Germann, and Gabriele Franch. Latent diffusion models for generative precipitation nowcasting with accurate uncertainty quantification, 2023.

[119] Benjamin Lewis and Devika Kakkar. Harvard CGA Geotweet Archive v2.0, 2016.

[120] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[121] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. The dawn of natural language to sql: Are we fully ready? *Proceedings of the VLDB Endowment*, 17(11):3318–3331, 2024.

[122] Fei Li, Tan Yigitcanlar, Madhav Nepal, Kien Nguyen, and Fatih Dur. Machine learning and remote sensing integration for leveraging urban sustainability: A review and framework. *Sustainable Cities and Society*, 96:104653, 2023.

[123] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. Llm for data management. *Proceedings of the VLDB Endowment*, 17(12):4213–4216, 2024.

[124] Haonan Li, Ehsan Hamzei, Ivan Majic, Hua Hua, Jochen Renz, Martin Tomko, Maria Vasardani, Stephan Winter, and Timothy Baldwin. Neural factoid geospatial question answering. *Journal of Spatial Information Science*, (23):65–90, December 2021.

[125] Jianing Li, Xi Nan, Ming Lu, Li Du, and Shanghang Zhang. Proximity QA: Unleashing the Power of Multi-Modal Large Language Models for Spatial Proximity Analysis, January 2024. arXiv:2401.17862 [cs].

[126] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[127] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Ma Chenhao, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. *Advances in Neural Information Processing Systems*, 36:42330–42357, December 2023.

[128] Ke Li, Yuxuan Wang, Zhichao Lin, and Yu Wang. T-jepa: Self-supervised trajectory representation learning with joint-embedding predictive architecture. In *Proceedings of the 32nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2024.

[129] Lizao Li, Rob Carver, Ignacio Lopez-Gomez, Fei Sha, and John Anderson. Seeds: Emulation of weather forecast ensembles with diffusion models, 2023.

[130] Siyu Li, Toan Tran, Haowen Lin, John Krumm, Cyrus Shahabi, Lingyi Zhao, Khurram Shafique, and Li Xiong. Geo-llama: Leveraging llms for human mobility trajectory generation with spatiotemporal constraints. *MDM*, 2025.

[131] Xiang Li, Kun Han, Han Hu, Yu Zhang, and Junchi Wang. Trajectorybert: Pretraining spatial trajectories with metric learning. In *Proceedings of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 12657–12667, 2023.

[132] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: a fast json parser for data analytics. *Proceedings of the VLDB Endowment*, 10(10):1118–1129, 2017.

[133] Yunqin Li, Nobuyoshi Yabuki, and Tomohiro Fukuda. Integrating GIS, deep learning, and environmental sensors for multicriteria evaluation of urban street walkability. *Landscape and Urban Planning*, 230:104603, 2023.

[134] Xin Liang et al. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *IEEE BigData*, pages 438–447, 2018.

[135] Zhenjie Liao and Lijuan Zhang. Spatial distribution characteristics and accessibility analysis of characteristic towns in guangdong province based on ripley's k function. *J. Math.*, 2022(1):2873707, 2022.

[136] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

[137] Haowen Lin, Sina Shaham, Yao-Yi Chiang, and Cyrus Shahabi. Generating realistic and representative trajectories with mobility behavior clustering. In Matthias Renz and Mario A. Nascimento, editors, *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2023, Hamburg, Germany, November 13-16, 2023*, pages 107:1–107:4. ACM, 2023.

[138] Fangyu Liu, Guy Emerson, and Nigel Collier. Visual spatial reasoning. *Transactions of the Association for Computational Linguistics*, 11:635–651, 2023.

[139] Haohe Liu, Zehua Chen, Yi Yuan, Xinhao Mei, Xubo Liu, Danilo Mandic, Wenwu Wang, and Mark D Plumbley. Audioldm: Text-to-audio generation with latent diffusion models. In *Proceedings of the 40th International Conference on Machine Learning*, pages 21450–21474. PMLR, 2023.

[140] Wei Liu, Meng Li, Yuchen Zhang, and Enhong Chen. Knowsite: Knowledge-driven site selection for urban retail expansion. In *Proceedings of the 31st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2023.

[141] Yongyi Liu, Yunfan Kang, Ahmed Mahmood, and Amr Magdy. Scalable evaluation of local k-function for radius-accurate hotspot detection in spatial networks. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '23, 2023.

[142] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to xml parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, page 223–230, USA, 2006. IEEE Computer Society.

[143] Minbo Ma, Peng Xie, Fei Teng, Bin Wang, Shenggong Ji, Junbo Zhang, and Tianrui Li. HiSTGNN: Hierarchical spatio-temporal graph neural network for weather forecasting. *Inf. Sci.*, 648(C), 2023.

[144] Ahmed R. Mahmood and Walid G. Aref. Query processing techniques for big spatial-keyword data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD, pages 1777–1782. ACM, 2017.

[145] Gengchen Mai, Weiming Huang, Jin Sun, Suhang Song, Deepak Mishra, Ninghao Liu, Song Gao, Tianming Liu, Gao Cong, Yingjie Hu, Chris Cundy, Ziyuan Li, Rui Zhu, and Ni Lao. On the opportunities and challenges of foundation models for geoai (vision paper). *ACM Trans. Spatial Algorithms Syst.*, 10(2), 2024.

[146] Gengchen Mai, Krzysztof Janowicz, Yingjie Hu, Song Gao, Bo Yan, Rui Zhu, Ling Cai, and Ni Lao. A review of location encoding for geoai: methods and applications. *International Journal of Geographical Information Science*, 36(4):639–673, 2022.

[147] Gengchen Mai, Krzysztof Janowicz, Bo Yan, Rui Zhu, Ling Cai, and Ni Lao. Multi-scale representation learning for spatial feature distributions using grid cells. *CoRR*, abs/2003.00824, 2020.

[148] Gengchen Mai, Krzysztof Janowicz, Rui Zhu, Ling Cai, and Ni Lao. Geographic Question Answering: Challenges, Uniqueness, Classification, and Future Directions, May 2021. arXiv:2105.09392 [cs].

[149] Gengchen Mai, Chiyu Jiang, Weiwei Sun, Rui Zhu, Yao Xuan, Ling Cai, Krzysztof Janowicz, Stefano Ermon, and Ni Lao. Towards general-purpose representation learning of polygonal geometries. *Geoinformatica*, 27(2):289–340, 2022.

[150] Majid Saeedan, Muhammad Shihab Rashid, Ahmed Eldawy, and Vagelis Hristidis. GS-QA, March 2025.

[151] Yuning Mao, Yichong Xu Li, Bo Pang, William Yang Wang, et al. End-to-end beam retrieval for multi-hop question answering. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024.

[152] Sergey Melnik et al. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[153] Matías Mendieta, Boran Han, Xingjian Shi, Yi Zhu, and Chen Chen. Towards geospatial foundation models via continual pretraining. 2023.

[154] Xiangrui Meng et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[155] Matthew J. Menne, Imke Durre, Russell S. Vose, Byron E. Gleason, and Tamara G. Houston. An Overview of the Global Historical Climatology Network-Daily Database. *Journal of Atmospheric and Oceanic Technology*, 29(7):897–910, July 2012. Publisher: American Meteorological Society Section: Journal of Atmospheric and Oceanic Technology.

[156] Microsoft. Computer generated building footprints in all 50 us states., 2020. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?MSBuildings&d`.

[157] Erxue Min, Xifeng Guo, Qiang Liu, Gen Zhang, Jianjing Cui, and Jun Long. A Survey of Clustering With Deep Learning: From the Perspective of Network Architecture. *IEEE Access*, 6:39501–39514, 2018.

[158] Roshanak Mirzaee, Hossein Rajaby Faghihi, Qiang Ning, and Parisa Kordjmashidi. Spartqa:: A textual question answering benchmark for spatial reasoning. *arXiv preprint arXiv:2104.05832*, 2021.

[159] Sobhan Moosavi, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. Accident risk prediction based on heterogeneous sparse data: New dataset and insights. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '19, page 33–42, New York, NY, USA, 2019. Association for Computing Machinery.

[160] Ingo Müller et al. Adaptive string dictionary compression in in-memory column-store database systems. In *EDBT*, volume 14, pages 283–294, 2014.

[161] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: challenges and opportunities. *Proc. VLDB Endow.*, 12(12):1986–1989, August 2019.

[162] Enkhbold Nyamsuren, Haiqi Xu, Eric J. Top, Simon Scheider, and Niels Steenbergen. Semantic complexity of geographic questions - A comparison in terms of conceptual transformations of answers. *AGILE: GIScience Series*, 4:1–10, June 2023. Publisher: Copernicus GmbH.

[163] Map features - OpenStreetMap Wiki, 2024.

[164] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11):1576–1589, 2018.

[165] Christina Pavlopoulou, E Preston Carman Jr, Till Westmann, Michael J Carey, and Vassilis J Tsotras. A parallel and scalable processor for json data. In *EDBT*, pages 576–587, 2018.

[166] Tuomas Pelkonen et al. Gorilla: A fast, scalable, in-memory time series database. *PVLDB*, 8(12):1816–1827, 2015.

[167] Marcin Przymus and Piotr Szymański. Map diffusion - text promptable map generation diffusion model. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Advances in Urban-AI*, UrbanAI '23, page 32–41, New York, NY, USA, 2023. Association for Computing Machinery.

[168] D. Punjani, K. Singh, A. Both, M. Koubarakis, I. Angelidis, K. Bereta, T. Beris, D. Bilidas, T. Ioannidis, N. Karalis, C. Lange, D. Pantazi, C. Papaloukas, and G. Stamoulis. Template-Based Question Answering over Linked Geospatial Data. In *Proceedings of the 12th Workshop on Geographic Information Retrieval*, pages 1–10, Seattle WA USA, November 2018. ACM.

[169] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: deep hierarchical feature learning on point sets in a metric space. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 5105–5114, Red Hook, NY, USA, 2017. Curran Associates Inc.

[170] Yujia Qiu, Xuehai Wu, Yulong Gao, Qiongkai Wu, Bin Zhang, and Baoxun Hu. Multi-hop question answering: Challenges and methods. *arXiv preprint arXiv:2204.09140*, 2022.

[171] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, volume abs/2103.00020, 2021.

[172] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents, 2022.

[173] Jinmeng Rao, Song Gao, Gengchen Mai, and Krzysztof Janowicz. Building privacy-preserving and secure geospatial artificial intelligence foundation models. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '23)*, pages 1–4, 2023.

[174] P. Ratanaworabhan, Jian Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *DCC*, pages 133–142, 2006.

[175] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

[176] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *18th Int.l Conf. on Medical Image Computing and Computer-Assisted Intervention*, MICCAI 2015, pages 234–241, 2015.

[177] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 410–420, 2007.

[178] Majid Saeedan and Ahmed Eldawy. Spatial parquet: A column file format for geospatial data lakes [extended version], 2022.

[179] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes, Tim Salimans, Jonathan Ho, David J Fleet, and Mohammad Norouzi. Photorealistic text-to-image diffusion models with deep language understanding, 2022.

[180] Srikumar Sastry, Subash Khanal, Aayush Dhakal, and Nathan Jacobs. GeoSynth: Contextually-Aware High-Resolution Satellite Image Synthesis . In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 460–470, Los Alamitos, CA, USA, June 2024. IEEE Computer Society.

[181] Christopher Scarpone, Sebastian T. Brinkmann, Tim Große, Daniel Sonnenwald, Martin Fuchs, and Blake Byron Walker. A multimethod approach for county-scale geospatial analysis of emerging infectious diseases: a cross-sectional case study of COVID-19 incidence in Germany. *International Journal of Health Geographics*, 19(1):32, 2020.

[182] Filippo Schiavio, Daniele Bonetta, and Walter Binder. Dynamic speculative optimizations for sql compilation in apache spark. *Proceedings of the VLDB Endowment*, 13(5):754–767, 2020.

[183] Dustin Schwenk, Apoorv Khandelwal, Christopher Clark, Kenneth Marino, and Roozbeh Mottaghi. A-OKVQA: A benchmark for visual question answering using world knowledge. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 641–657, 2022.

[184] Juan Sequeda, Dean Allemang, and Bryon Jacob. A benchmark to understand the role of knowledge graphs on large language model's accuracy for question answering on enterprise sql databases. In *Proceedings of the 7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '24, New York, NY, USA, 2024. Association for Computing Machinery.

[185] Tong Shen, Yang Li, and José M. F. Moura. Forecasting COVID-19 dynamics: Clustering, generalized spatiotemporal attention, and impacts of mobility and geographic proximity. In *39th IEEE International Conference on Data Engineering*, ICDE, pages 2892–2904. IEEE, 2023.

[186] Jia Shi. Column partition and permutation for run length encoding in columnar databases. In *SIGMOD*, page 2873–2874. Association for Computing Machinery, 2020.

[187] N Shobha and T. Asha. Monitoring weather based meteorological data: Clustering approach for analysis. In *2017 Int. Conf. on Innovative Mechanisms for Industry Applications (ICIMIA)*, pages 75–81, 2017.

[188] Maria Despoina Siampou, Jialiang Li, John Krumm, Cyrus Shahabi, and Hua Lu. Poly2vec: Polymorphic encoding of geospatial objects for spatial reasoning with deep neural networks, 2024.

[189] Maria Despoina Siampou, Jialiang Li, John Krumm, Cyrus Shahabi, and Hua Lu. Poly2vec: Polymorphic fourier-based encoding of geospatial objects for geoai applications, 2025.

[190] Amanpreet Singh, Yang Song, Jacob Menick, Quoc Le, Chen-Yu Tan, Dhruv Mahajan, Ting Xu, Zongze Wang, Shiry Ginosar, Han Wang, et al. Make-a-video: Text-to-video generation without text-video data. *arXiv preprint arXiv:2209.14792*, 2022.

[191] Samriddhi Singla, Yaming Zhang, and Ahmed Eldawy. OSMX: spark-based geospatial data extractor from OpenStreetMap. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*, pages 1–4, Seattle Washington, November 2022. ACM.

[192] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics, 2015.

[193] Adam J. Stewart, Caleb Robinson, Isaac A. Corley, Anthony Ortiz, Juan M. Lavista Ferres, and Arindam Banerjee. Torchgeo: deep learning with geospatial data. In *Proceedings of the 30th Int. Conf. on Advances in Geographic Information Systems*, SIGSPATIAL '22, 2022.

[194] Michael Stonebraker et al. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[195] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9. IEEE, 2015.

[196] Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 641–651, 2018.

[197] Alon Talmor, Ori Yoran, Amnon Catav, Dan Lahav, Yizhong Wang, Akari Asai, Gabriel Ilharco, Hannaneh Hajishirzi, and Jonathan Berant. Multimodalqa: Complex question answering over text, tables and images, 2021.

[198] Wenwu Tang, Wenpeng Feng, and Meijuan Jia. Massively parallel spatial point pattern analysis: Ripley's K function accelerated using graphics processing units. *International Journal of Geographical Information Science*, 29(3):412–439, 2015.

[199] Ran Tao, Jean-Claude Thill, and Ikuho Yamada. Detecting Clustering Scales with the Incremental K-Function: Comparison Tests on Actual and Simulated Geospatial Datasets. In Vasily Popovich, Christophe Claramunt, Manfred Schrenk, Kyrill Korolenko, and Jérôme Gensel, editors, *Information Fusion and Geographic Information Systems (IF&GIS' 2015): Deep Virtualization for Mobile GIS*, pages 93–107. 2015.

[200] David Alexander Tedjopurnomo, Zhifeng Bao, Baihua Zheng, Farhana Murtaza Choudhury, and A. Kai Qin. A survey on modern deep neural network for traffic prediction: Trends, methods and challenges (extended abstract). In *39th IEEE International Conference on Data Engineering*, ICDE, pages 3795–3796. IEEE, 2023.

[201] Jilong Tian, Jiangjiang Wu, Hao Chen, and Mengyu Ma. Mapgen-diff: An end-to-end remote sensing image to map generator via denoising diffusion bridge model. *Remote Sensing*, 16(19), 2024.

[202] Kai Tian, Shuigeng Zhou, and Jihong Guan. DeepCluster: A General Clustering Framework Based on Deep Learning. In *Machine Learning and Knowledge Discovery in Databases*, pages 809–825, 2017.

[203] Aysim Toker, Marvin Eisenberger, Daniel Cremers, and Laura Leal-Taixe. SatSynth: Augmenting Image-Mask Pairs Through Diffusion Models for Aerial Semantic Segmentation . In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 27685–27695, Los Alamitos, CA, USA, June 2024. IEEE Computer Society.

[204] Harsh Trivedi, Matt Gardner, Wen-tau Yih, Tom Kwiatkowski, and Oyvind Tafjord. Musique: Multi-hop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics (TACL)*, 10:648–662, 2022.

[205] Aristeidis Tsaris, Philipe Ambrozio Dias, Abhishek Potnis, Junqi Yin, Feiyi Wang, and Dalton Lunga. Pretraining Billion-scale Geospatial Foundational Models on Frontier, 2024.

[206] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, 2017.

[207] Alessandro Venerandi, Hal Mellen, Ombretta Romice, and Sergio Porta. Walkability indices—the state of the art and future directions: A systematic review. *Sustainability*, 16(16), 2024.

[208] Deepak Vohra. *Apache Parquet*, pages 325–335. Apress, Berkeley, CA, 2016.

[209] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. A learning-based framework for spatial join processing: estimation, optimization and tuning. *The VLDB Journal*, 33(4):1155–1177, 2024.

[210] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldway. Using deep learning for big spatial data partitioning. *ACM Trans. Spatial Algorithms Syst.*, 7(1), 2020.

[211] Tin Vu and Ahmed Eldawy. R*-Grove: Balanced Spatial Partitioning for Large-Scale Datasets. August 2020.

[212] Tin Vu, Ahmed Eldawy, Vagelis Hristidis, and Vassilis Tsotras. Incremental partitioning for efficient spatial data analytics. *Proc. VLDB Endow.*, 15(3):713–726, 2021.

[213] Tin Vu, Sara Migliorini, Ahmed Eldawy, and Alberto Bulussi. Spatial Data Generators. In *1st ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2019)*, pages 13–24. ACM, 2019.

[214] Fusheng Wang, Rubao Lee, Dejun Teng, Xiaodong Zhang, and Joel Saltz. High-performance spatial data analytics: Systematic rd for scale-out and scale-up solutions from the past to now. *Proc. VLDB Endow.*, 17(12):4507–4520, 2024.

[215] Jianguo Wang, Eric Hanson, Guoliang Li, Yannis Papakonstantinou, Harsha Simhadri, and Charles Xie. Vector databases: What's really new and what's next? (vldb 2024 panel). *Proc. VLDB Endow.*, 17(12):4505–4506, 2024.

[216] Yuan Wang, Zhipeng Gui, Huayi Wu, Dehua Peng, Jinghang Wu, and Zousen Cui. Optimizing and accelerating space–time Ripley 's K function based on Apache Spark for distributed spatiotemporal point pattern analysis. *Future Generation Computer Systems*, 105:96–118, 2020.

[217] Johannes Welbl, Pontus Stenetorp, and Sebastian Riedel. Constructing datasets for multi-hop reading comprehension across documents. *Transactions of the Association for Computational Linguistics (TACL)*, 6:287–302, 2018.

[218] Randall T. Whitman, Bryan G. Marsh, Michael B. Park, and Erik G. Hoel. Distributed spatial and spatio-temporal join on apache spark. *ACM Trans. Spatial Algorithms Syst.*, 5(1), 2019.

[219] Xiaoyang Wu, Li Jiang, Peng-Shuai Wang, Zhijian Liu, Xihui Liu, Yu Qiao, Wanli Ouyang, Tong He, and Hengshuang Zhao. Point transformer v3: Simpler, faster, stronger. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4840–4851, 2024.

[220] Yiqun Xie, Zhaonan Wang, Gengchen Mai, Yanhua Li, Xiaowei Jia, Song Gao, and Shaowen Wang. Geo-foundation models: Reality, gaps and opportunities (vision paper). In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '23)*, pages 1–4, 2023.

[221] Yan Xu, Huicheng Zheng, Yu Wang, and Yong Yu. Spatial-temporal transformer networks for traffic flow forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1234–1241, 2020.

[222] Bo Yan, Krzysztof Janowicz, Gengchen Mai, and Song Gao. From itdl to place2vec: Reasoning about place type similarity and relatedness by learning embeddings from augmented spatial contexts. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '17, 2017.

[223] Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. Diffusion models: A comprehensive survey of methods and applications, 2024.

[224] Wenzhe Yang, Sheng Wang, Yuan Sun, and Zhiyong Peng. Fast dataset search with earth mover's distance. *Proc. VLDB Endow.*, 15(11):2517–2529, 2022.

[225] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[226] Milo Yip. Native JSON Benchmark, 2021. Available at `https://github.com/miloyip/nativejson-benchmark`.

[227] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*, pages 1–4, 2015.

[228] Xumin Yu, Lulu Tang, Yongming Rao, Tiejun Huang, Jie Zhou, and Jiwen Lu. Point-BERT: Pre-training 3D Point Cloud Transformers with Masked Point Modeling . In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19291–19300, Los Alamitos, CA, USA, June 2022. IEEE Computer Society.

[229] Matei Zaharia et al. Fast and interactive analytics over hadoop data with spark. *Usenix Login*, 37(4):45–51, 2012.

[230] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*, 2020.

[231] Xin Zhang and Ahmed Eldawy. Spatial query optimization with learning. *Proc. VLDB Endow.*, 17(12):4245–4248, 2024.

[232] Yaming Zhang and Ahmed Eldawy. Openstreetmap all map points, 2021. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?osm21/all_nodes&d`.

[233] Hengshuang Zhao, Li Jiang, Jiaya Jia, Philip Torr, and Vladlen Koltun. Point transformer. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 16239–16248, 2021.

[234] Kai Zhao et al. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *ICDE*, pages 1643–1654, 2021.

[235] Wayne Xin Zhao, Jing Liu, Ruiyang Ren, and Ji-Rong Wen. Dense text retrieval based on pretrained language models: A survey. *ACM Trans. Inf. Syst.*, 42(4), February 2024.

[236] Xujun Zhao, Jifu Zhang, and Xiao Qin. k nn-dp: Handling data skewness in knn joins using mapreduce. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):600–613, 2018.

[237] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[238] Yue Zhou, Litong Feng, Yiping Ke, Xue Jiang, Junchi Yan, Xue Yang, and Wayne Zhang. Towards vision-language geo-foundation model: A survey, 2024.

[239] Yuanshao Zhu, Yongchao Ye, Shiyao Zhang, Xiangyu Zhao, and James J.Q. Yu. Difftraj: generating gps trajectory with diffusion probabilistic model. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.