# Spatial Parquet: A Column File Format for Geospatial Data Lakes*


Majid Saeedan
msaee007@ucr.edu
University of California, Riverside
Riverside, California, USA

Ahmed Eldawy
eldawy@ucr.edu
University of California, Riverside
Riverside, California, USA



## ABSTRACT

Modern data analytics applications prefer to use column-storage formats 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 is not readily supported by Parquet. This paper introduces Spatial Parquet, a Parquet extension that efficiently supports geospatial data. Spatial Parquet 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 geometries 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 Spatial Parquet 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 new research directions to further improve geospatial data storage in column format.



## CCS CONCEPTS

• **Information systems** → *Data warehouses*; **Data compression**; **Geographic information systems**.

## KEYWORDS

datasets, geospatial, column store, parquet

**ACM Reference Format:**
Majid Saeedan and Ahmed Eldawy. 2022. Spatial Parquet: A Column File Format for Geospatial Data Lakes. In *The 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '22), November 1–4, 2022, Seattle, WA, USA*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3557915.3561038

*This work is supported in part by the National Science Foundation (NSF) under grants IIS-2046236 and IIS-1954644.




*SIGSPATIAL '22, November 1–4, 2022, Seattle, WA, USA*



ACM ISBN 978-1-4503-9529-8/22/11.
https://doi.org/10.1145/3557915.3561038


## 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. 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, they add 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 some unique advantages over row formats.

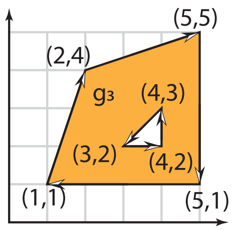
One of the most popular column formats is Parquet [14] which is inspired by Google's Dremel [10] system. Parquet is more geared towards big variety data by allowing nested and repeated attributes such as in JSON files. With the increasing amount of geospatial data, Parquet is a very attractive solution 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 this paper, we present Spatial Parquet, an extension to the Parquet file format that adds support for spatial vector data.

The rest of this paper is organized as follows. Section 2 explains how we structure all standard geospatial data in Spatial Parquet. Section 3 describes the FP-delta encoding method. Section 4 introduces the light-weight spatial index. Experimental evaluation results are detailed in Section 5. The related work is explained in Section 6. Finally, Section 7 concludes the paper.

## 2 THE STRUCTURE

This section describes how Spatial Parquet 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 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. 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 these challenges, we propose the following schema, based on Google's Protocol Buffers Format (PBF):

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




**(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 |
| | 3 | 2 |

**(b) Column Representation**

**Figure 1: Column representation for a Polygon**

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 hole 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 group in the feature.

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. 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.

Figure 1 illustrates an example of a polygon with one hole. All other geometry types are easily represented under this scheme except *GeometryCollection* which needs some tweaks. Interested readers can refer to [13] for the full details.

## 3 THE ENCODING

This section describes how we encode the column-represented geometries in Spatial Parquet to improve the storage efficiency.

The geometry type is stored as an integer in the range $[0, 6]$. In almost all practical cases, all geometries in one dataset have the same type. Therefore, run-length-encoding (RLE) is used to encode the geometry type value. RLE replaces consecutive entries with the same value with two numbers, count and value.

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, and then for subsequent values it stores only the delta between each value and its previous one. 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.

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

**Algorithm 1** FP-delta encoding algorithm*

```
function FP-DELTA-ENCODE(double[] X, BitOutputStream out)
    n* = COMPUTEBESTDELTABITS(X)
    resetMarker = -1 ⋙ (64-n*)
    significantOnes = -1 ≪ (n*)
    out.write(n*, 8)
    out.write(X[0], 64)
    for i = 1 to |X| − 1 do
        delta = cast-long(X[i])-cast-long(X[i − 1])
        zigzag = (delta ≫ 63) ⊕ (delta ≪ 1)
        if (zigzag & significantOnes ≠ 0) or (zigzag = resetMarker) then
            out.write(resetMarker, n*)
            out.write(X[i], 64)
        else
            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

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, every 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 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. Algorithm 1 provides the pseudo-code of the FP-delta encoding algorithm. Refer to [13] for more details about the encoding/decoding steps.

**Table 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 |

## 4 THE INDEXING

Parquet provides a light-weight method for pruning non-relevant records by adding *column statistics*. The structure we propose in Spatial Parquet 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 the Well-Known-Binary (WKB) 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. This can work as a grid index that can be used for range queries.

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, reducing the computational cost.

## 5 EXPERIMENTS

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

We use four datasets for all evaluations. All the dataset are publicly available on UCR-Star [6] and are summarized in Table 1. The versions of these datasets only contain geometry data, and all objects are stripped of any metadata.

We implement Spatial Parquet in Java based on the original Parquet Java repository. We compare Spatial Parquet to three existing baselines: GeoParquet, ShapeFile, and GeoJSON.

First, we evaluate these formats based on the total size without compression. The left part of Table 2 shows the size of data stored in these formats without any compression. For all datasets, Spatial Parquet with delta encoding significantly decreases the size of the data. In the case of the PT, MB and eB datasets, its size is less than half that of the nearest size. Compressing the data using a general purpose compression technique can further reduce the sizes. Note that there are some differences in how the compression is applied to these formats, which is the reason GeoJSON has the smallest size in two cases. Refer to [13] for details and discussion about performance implications.

**Table 2: Output size in GB with/without compression**

| | Uncompressed | | | | Compressed | | | |
|---|---|---|---|---|---|---|---|---|
| **Format** | **PT** | **TR** | **MB** | **eB** | **PT** | **TR** | **MB** | **eB** |
| Spatial Parquet | **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** |

**Table 3: Write/Read time in seconds for uncompressed formats**

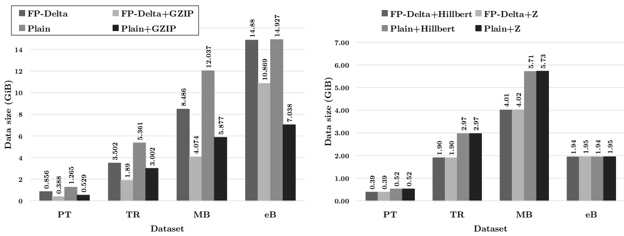
| | Writing Time | | | | Reading Time | | | |
|---|---|---|---|---|---|---|---|---|
| **Format** | **PT** | **TR** | **MB** | **eB** | **PT** | **TR** | **MB** | **eB** |
| Spatial Parquet | **74** | 215 | 544 | **833** | 49 | 143 | 455 | 546 |
| GeoParquet | 226 | **99** | **425** | 1490 | **17** |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

Next, we compare the writing time of Spatial Parquet against the baselines. Table 3 shows the writing time in seconds for the uncompressed files. Spatial Parquet 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 datasets 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 Spatial Parquet, we will further optimize the writing operation to reduce any potential overhead.

Finally, we compare Spatial Parquet to the baselines in terms of the reading time. Table 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 Parquet.

In the following, we delve into Spatial Parquet 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 2a, in all cases, FP-delta results in a smaller size with and without GZIP compression, except for the eB dataset. eB 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 2b. The main difference can be noticed in the eB dataset because it contains unsorted records of type Point.

Both FP-delta and sorting add benefits in reducing the final data size, but they add some performance overhead. In the worst case, it seems that FP-delta adds up to 80% of overhead compared




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

**Figure 2: The effect of sorting on output size in SpatialParquet**

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 overhead can be justified. Moreover, in practice we can sort very big data using distributed sorting/indexing techniques. Beast [3] 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. Also, refer to [13] for more details about this overhead.

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 3 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.

To summarize, Spatial Parquet can significantly reduce the size of geospatial data, improve the query performance, with a minimal overhead. We call on the geospatial community to widely adopt it in real applications.

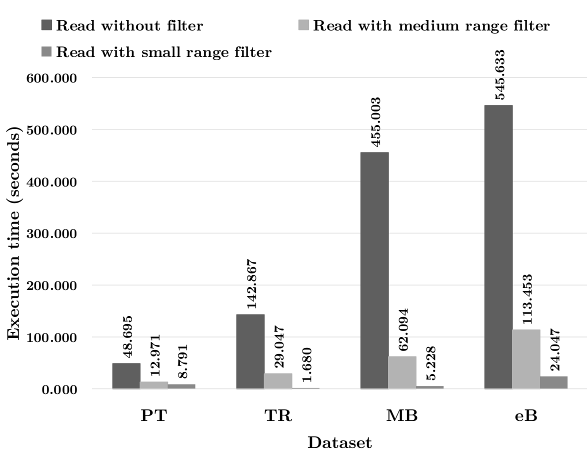



**Figure 3: The performance of the light-weight spatial index**

# 6 RELATED WORK

*Column Formats.* Column stores [14] 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 [10] was introduced by Google which then inspired the open-source Parquet file format [14]. An experimental evaluation [4] showed the efficiency of Parquet with text data compared to ORC. The only existing attempt to provide a column-oriented format for geo-spatial data is GeoParquet [5], 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 [1, 7, 9, 11]. 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 [12] targets time series data. It applies XOR between consecutive values and adds post-processing steps to remove leading and trailing zeros. Similarly, the work in [2] focuses on time series data and improves over Gorilla [12]. Also, [8] focuses on time series data but provides a different approach by encoding similar patterns in time series by mapping them to a dictionary.

# 7 CONCLUSION

This paper introduced Spatial Parquet, a column-oriented file format for geospatial data. Spatial Parquet is designed to store large-scale spatial data in a column format that reduces disk size and improves the performance of analytical queries.

# REFERENCES

[1] Sushila Aghav. 2010. Database compression techniques for performance optimization. In *ICCET*, Vol. 6. V6–714–V6–717.
[2] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time series compression for the internet of things. *IMWUT* 2, 3 (2018), 1–23.
[3] Ahmed Eldawy et al. 2021. Beast: Scalable Exploratory Analytics on Spatio-temporal Data. In *CIKM*. 3796–3807.
[4] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. 2014. SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. *PVLDB* 7, 12 (2014).
[5] geoparquet 2022. GeoParquet: Store Vector Data in Apache Parquet. https://github.com/opengeospatial/geoparquet
[6] Saheli Ghosh et al. 2019. UCR-STAR: The UCR Spatio-Temporal Active Repository. *SIGSPATIAL Special* 11, 2 (Dec. 2019), 34–40.
[7] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. 2017. Practical String Dictionary Compression Using String Dictionary Encoding. In *Innovate-Data*.
[8] Abdelouahab Khelifati, Mourad Khayati, and Philippe Cudré-Mauroux. 2019. CORAD: Correlation-Aware Compression of Massive Time Series using Sparse Dictionary Coding. In *IEEE BigData*. 2289–2298.
[9] Robert Lasch et al. 2019. Fast & Strong: The Case of Compressed String Dictionaries on Modern CPUs. In *DaMoN@SIGMOD*. Article 4, 10 pages.
[10] Sergey Melnik et al. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010), 330–339.
[11] Ingo Müller et al. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems.. In *EDBT*, Vol. 14. 283–294.
[12] Tuomas Pelkonen et al. 2015. Gorilla: A fast, scalable, in-memory time series database. *PVLDB* 8, 12 (2015), 1816–1827.
[13] Majid Saeedan and Ahmed Eldawy. 2022. Spatial Parquet: A Column File Format for Geospatial Data Lakes [Extended Version]. https://doi.org/10.48550/ARXIV.2209.02158
[14] Deepak Vohra. 2016. *Apache Parquet.* Apress, Berkeley, CA, 325–335. https://doi.org/10.1007/978-1-4842-2199-0_8