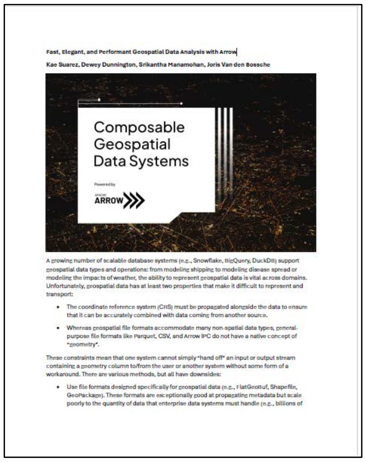
VOLTRON DATA

carahsoft.


Fast, Elegant, and Performant Geospatial Data Analysis with Arrow

Kae Suarez, Dewey Dunnington, Srikantha Manamohan, Joris Van den Bossche

A growing number of scalable database systems (e.g., Snowflake, BigQuery, DuckDB) support geospatial data types and operations: from modeling shipping to modeling disease spread or modeling the impacts of weather, the ability to represent geospatial data is vital across domains. Unfortunately, geospatial data has at least two properties that make it difficult to represent and transport:

- The coordinate reference system (CRS) must be propagated alongside the data to ensure that it can be accurately combined with data coming from another source.
- Whereas geospatial file formats accommodate many non-spatial data types, general-purpose file formats like Parquet, CSV, and Arrow IPC do not have a native concept of "geometry".

These constraints mean that one system cannot simply "hand off" an input or output stream containing a geometry column to/from the user or another system without some form of a workaround. There are various methods, but all have downsides:

- Use file formats designed specifically for geospatial data (e.g., FlatGeoBuf, Shapefile, GeoPackage). These formats are exceptionally good at propagating metadata but scale poorly to the quantity of data that enterprise data systems must handle (e.g., billions of

CI·ǎŰ 9fŜƎI·yŰ I·yŘ tŜNJŦƵNJYI·yŰ DŜƵǎLJI·ŰʎI·f 5I·ŰI· !yЛ·fȅǎʎǎ øʎŰK !NJNJƵø

Thank you for downloading this Voltron Data resource. Carahsoft is the distributor for Voltron Data's AI and ML solutions available via NASA SEWP V, ITES-SW2, NASPO ValuePoint, and many more contract vehicles.

To learn how to take the next step toward acquiring Voltron Data's solutions, please check out the following resources and information:

For Voltron Data overview:
carah.io/Voltron-Data

For upcoming events:
carah.io/Voltron-Events

For additional resources:
carah.io/Voltron-Resources

For additional Artificial Intelligence:
carah.io/ai-solutions

To set up a meeting:
VoltronData@carahsoft.com

To purchase, check out the contract vehicles available for procurement:
carah.io/procurement

Fast, Elegant, and Performant Geospatial Data Analysis with Arrow

Kae Suarez, Dewey Dunnington, Srikantha Manamohan, Joris Van den Bossche

A growing number of scalable database systems (e.g., Snowflake, BigQuery, DuckDB) support geospatial data types and operations: from modeling shipping to modeling disease spread or modeling the impacts of weather, the ability to represent geospatial data is vital across domains. Unfortunately, geospatial data has at least two properties that make it difficult to represent and transport:

- The coordinate reference system (CRS) must be propagated alongside the data to ensure that it can be accurately combined with data coming from another source.
- Whereas geospatial file formats accommodate many non-spatial data types, general-purpose file formats like Parquet, CSV, and Arrow IPC do not have a native concept of "geometry".

These constraints mean that one system cannot simply "hand off" an input or output stream containing a geometry column to/from the user or another system without some form of a workaround. There are various methods, but all have downsides:

- Use file formats designed specifically for geospatial data (e.g., FlatGeoBuf, Shapefile, GeoPackage). These formats are exceptionally good at propagating metadata but scale poorly to the quantity of data that enterprise data systems must handle (e.g., billions of

rows). There is typically a high cost to read/write these formats, most of which were not designed to stream large volumes of data to/from other systems.

- Use a binary representation like a well-known binary (e.g., a blob column). This method has a serialization/deserialization cost and requires a dedicated tool to encode and/or decode the content of the column. This can happen many times in the course of a data pipeline and the compute cost of this serialization/deserialization can add up. It also does not propagate metadata without modification.
- Use multiple columns to encode a point (e.g., longitude, latitude). This method avoids serialization/deserialization but doesn't scale to other geometry types (e.g., polyline, polygon). It also does not propagate metadata.

For the workarounds that do not propagate metadata, many ways have been invented to minimize the chance that it will be ignored by a consumer: "sidecar" files (i.e., files with a similar name in the same directory) with geospatial metadata and file-level key/value stores (e.g., GeoParquet) are two examples; however, there is a cost to these workarounds: consumers must special-case the scan of every file type that might contain a geometry column.

What if geospatial data could be a first-class citizen of the modern composable data system? Whenever we see systems like this, where integration and interoperability seem impossible, our first thought is always Arrow.

Arrow

Arrow is a standard for representing data in-memory in columnar form. Like with any standard, the description can seem dull, but the results are exciting. As a standard for in-memory data, Arrow enables interoperability between applications and enables tools to focus more on their objectives instead of wrangling data into shape. Furthermore, with a high-performance implementation, and the computational benefits of columnar data, Arrow offers not only interoperability but acceleration.

No need to just take our word for it — Arrow is popular, and more so by the month. For example, as of version 2.0, pandas can use Arrow as the backend for its tables, which supplies notable gains. ClickHouse uses Arrow for data import and export, Polars is built on Arrow, Ibis supports Arrow — and more! For an initial list of where Arrow is in use, check out the Arrow site.

Arrow is already in use in many of these enterprise data systems to enable standardized data representation and interoperability. This support enables composable data pipelines through the modularity offered by standards. However, none of these advantages have been in reach for geospatial data. Due to the complex-but-vital metadata, and various ways to represent shapes, geospatial data requires support beyond what Arrow offers by default for tabular data.

Thus, we present that support.

GeoArrow

Enter GeoArrow. The newly proposed GeoArrow specification defines a set of Arrow extension types and memory layouts that allow Arrow-compatible components to represent geospatial data like any

other data type. This includes ADBC, Parquet, Flight, IPC for interprocess communication, and more. Wherever Arrow lives in your composable data pipeline, your geospatial data and metadata can go along for the ride with no modification to the intermediary components.

With this, you can work between systems and set up pipelines without unwieldy glue code. Furthermore, as with all things Arrow, you aren't giving up performance for compatibility. Instead of just saying that, let's take the time to demonstrate.

We'll show how the addition of GeoArrow to your stack enables you to leverage tools such as Parquet, extend your existing Dataframe toolbox with additional functionality instead of additional tools, and enable interoperability with Flight. This showcase will use the experimental Python bindings for GeoArrow available on Github.

Faster and Smaller with Parquet

For this exploration, we're going to use the Microsoft U.S. Building Footprints data set repackaged in GeoArrow format as a single point for each building—about 130 million points.

Parquet and Arrow pair well together, with the former allowing effective storage of columnar data in binary form, and the latter helping you once your data is in-memory. Of vital importance is Parquet's ability to faithfully represent most Arrow data types—including the new GeoArrow extension types.

When it comes to getting compression, ease-of-use, and performance all at once, Parquet is often a good choice. This trend extends to geospatial data. To start, let's see how the Microsoft Buildings dataset looks on disk if we convert from the FlatGeoBuf format to Parquet:

BASH

```
  ls -lh microsoft-buildings-point.fgb microsoft-buildings-point.parquet

9.7G Sep 28 16:21 microsoft-buildings-point.fgb
1.9G Oct  3 10:03 microsoft-buildings-point.parquet
```

9.7 GB cut straight down to 1.9 GB, just by using Parquet for geospatial data instead of FlatGeoBuf — and FlatGeoBuf is a binary format, just like Parquet.

And how does performance compare? For assessing this, we'll use the pyogrio package to read the FlatGeoBuf file. We chose this package because of its notable performance, ability to leverage Arrow and, of course, the ability to read FlatGeoBuf. For comparison, we'll use pyarrow to read the Parquet file:

PYTHON

```
import pyogrio
%timeit table_fgb = pyogrio.raw.read_arrow("microsoft-buildings-point.fgb")
#> 17.7 s ± 133 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

import geoarrow.pyarrow as ga
import pyarrow.parquet as pq
%timeit table_geoarrow = pq.read_table("microsoft-buildings-point.parquet")
#> 1.14 s ± 36.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)}
```

15x faster performance, just by using Parquet. For a whole-file load-into-memory scenario, the Parquet format is much faster. Furthermore, the GeoArrow encoding of the points in the Parquet version means we can operate on the data without a deserialization step! This means we can, for example, extract arrays of the longitude and latitude coordinates of each building instantaneously (and without copying) for the GeoArrow-encoded version, whereas for the version read from FlatGeoBuf we incur an additional 2 seconds of deserialization and an additional copy in memory:

```python
%timeit lon, lat = ga.point_coords(table_geoarrow["geometry"])
#> 13.5 ms ± 214 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
fgb_geometry = ga.wkb().wrap_array(table_fgb["wkb_geometry"])
%timeit lon, lat = ga.point_coords(fgb_geometry)
#> 2.02 s ± 80 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)}
```

Geospatial data supported everywhere in the Arrow universe

You may have noticed above that there was nothing special about the call to pyarrow.parquet.read_table(): as long as the extension type is loaded (in this case using import geoarrow.pyarrow), you can read and write Parquet files that contain geometry without special-casing any of your existing read or write code. In fact, this applies to *any* PyArrow code in your existing pipeline. For example, if you wanted to use pyarrow.dataset to store our example as a series of partitioned Parquet files, pyarrow.dataset.write_dataset() works out-of-the box:

```python
from pyarrow import dataset

dataset.write_dataset(
    table_geoarrow,
    "buildings_dataset",
    format="parquet",
    partitioning=["src_file"]
)
```

Similarly, pyarrow.dataset.dataset() can read the dataset like any other:

```python
dataset.dataset("buildings_dataset").head(5)
#> pyarrow.Table
#> geometry: extension<geoarrow.point<PointType>>
```

Arrow IPC, zero-copy sharing using the Arrow C Data interface, Flight, and any other Arrow-based component in your pipeline *just work*.

GeoArrow in Flight

One of these components deserves a special highlight: Arrow Flight. Flight is a framework for high-throughput data transport that leverages gRPC and the Arrow IPC format to stream data from one place to another. Whether it's transferring portions of a dataset among nodes in a cluster, sharing data between a host process and a Docker container, or streaming a database result to a client, Flight abstracts away the details of optimizing the transport layer so your code can focus on the logic specific to your application. You can write a Flight server in a few lines of Python:

```python
from pyarrow import flight
import geoarrow.pyarrow as _

class FlightServer(flight.FlightServerBase):
    def do_put(self, context, descriptor, reader, writer):
        uploaded_table = reader.read_all()

if __name__ == "__main__":
    server = FlightServer(location="grpc://0.0.0.0:8815")
    server.serve()
```

...and connect to it with just a few more:

```python
import pyarrow.flight as flight

client = flight.connect("grpc://0.0.0.0:8815")
upload_descriptor = flight.FlightDescriptor.for_path("uploaded")
```

For our buildings example, this allows us to send the entire dataset to another local process in 700ms:

```python
def put_flight(tab):
    upload_descriptor = flight.FlightDescriptor.for_path("uploaded")
    writer, _ = client.do_put(upload_descriptor, tab.schema)
    writer.write_table(tab)
    writer.close()

%timeit put_flight(table_geoarrow)
#> 700 ms ± 118 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

An alternative to this type of transfer would be to write the data to a file from one process and read it from another. In the best-case scenario where we're both reading and writing from the same solid-state drive, Flight is over twice as fast:

```python
from pyarrow import feather

def put_file(tab):
    feather.write_feather(tab, "temp.arrow", compression="uncompressed")
    feather.read_feather("temp.arrow")

%timeit put_file(table_geoarrow)
#> 2.03 s ± 234 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Like the other pieces of PyArrow, adding geospatial support is as easy as import geoarrow.pyarrow. Definitely boring, but that's exactly how we want our interoperability to work — no fuss, no specialization, and like you've changed nothing at all.

Future of GeoArrow

The 0.1.0 release of GeoArrow—like the Python, C, Rust, and WebAssembly implementations that accompany it—is a preliminary release intended to solicit feedback from users and developers such that we can collectively converge on a standard representation for the movement and storage of all things geospatial in Arrow. With feedback from the community we hope to move towards a stable 1.0 version of the specification in the next few months.

Voltron Data helps companies integrate tools within the Arrow ecosystem like ADBC, Arrow Flight, nanoarrow, and more. To learn more about our approach, check out our Product page.