IMPROVING DATA ACCESS WITH PYTHON AND THE OPENDAP PROTOCOL

MIGUEL ANGEL JIMÉNEZ-URIAS[1]

The explosive growth of data archives has pushed organizations to move their data into cloud object storage, as exemplified by NASA's Earthdata Cloud initiative. Such data migration into the cloud has introduced significant challenges and opportunities, given the preferred choice of metadata-rich and self-describing file formats by data producers for long archival storage such as HDF and NetCDF. These file formats were designed to optimize random access, and reading the file's metadata can lead to many expensive API requests when these files are accessed via remote protocols such as HTTP or S3. To bypass this challenge some data producers are opting to reformat the data in one of a handful of newer *Cloud-Native* formats, but not all datasets can be reformatted, and the cost of hosting multiple copies of datasets in different formats makes this a nonviable option.

In practice, data will remain heterogeneous, and analysis workflows will continue to take place both inside and outside cloud environments, and so continued effort is needed to support efficient workflows across environments and architectures. Data-proximate subsetting, such as that performed by OPeNDAP servers, is an essential tool for scalable and reproducible data workflows, allowing users to download only the data relevant to their analysis while reducing the egress costs associated with bulk data downloads (*Virapongse and Gallagher*, 2025). The purpose of this article is to describe how access has improved over the last few years and outline best practices that can lead to significant speed ups when accessing remote and distributed geospatial data.

There are many excellent open-source Python libraries available for analyzing geospatial data, but few match the community-driven `Xarray`. With its `Dask`-enabled parallelism and its use of "*backend engines*" to abstract read/write operations, a user can access data in different file formats with a single line of code and be ready to analyze data, without having to know beforehand the source file format and independently of whether the data are stored locally or distributed across object storage or remote HTTP servers (*Hoyer and Hamman*, 2017). Moreover, `Xarray` provides the visual and "lazy" representation of datasets that many data scientists have come to rely on. However, getting the most out of `Xarray` and its backend engines can require some expertise, particularly in the case of distributed (remote) data, potentially even becoming a barrier difficult to overcome.

Given the continued development of OPeNDAP, in particular the work of OPeNDAP on NASA funded DMR++ (see *Jimenez-Urias and Gallagher* (2026)), there is a growing need for tools that can efficiently "*speak*" OPeNDAP and efficiently access data via subset-driven workflows (as opposed to full-file downloads). This article presents a *ythonic* approach to accessing data through the `DAP4` OPeNDAP protocol, and outlining guidelines when using Xarray and Pydap that can lead to efficient data access via the OPeNDAP protocol.

Date: April 1, 2026
[1] OPeNDAP, Narragansett, RI
.

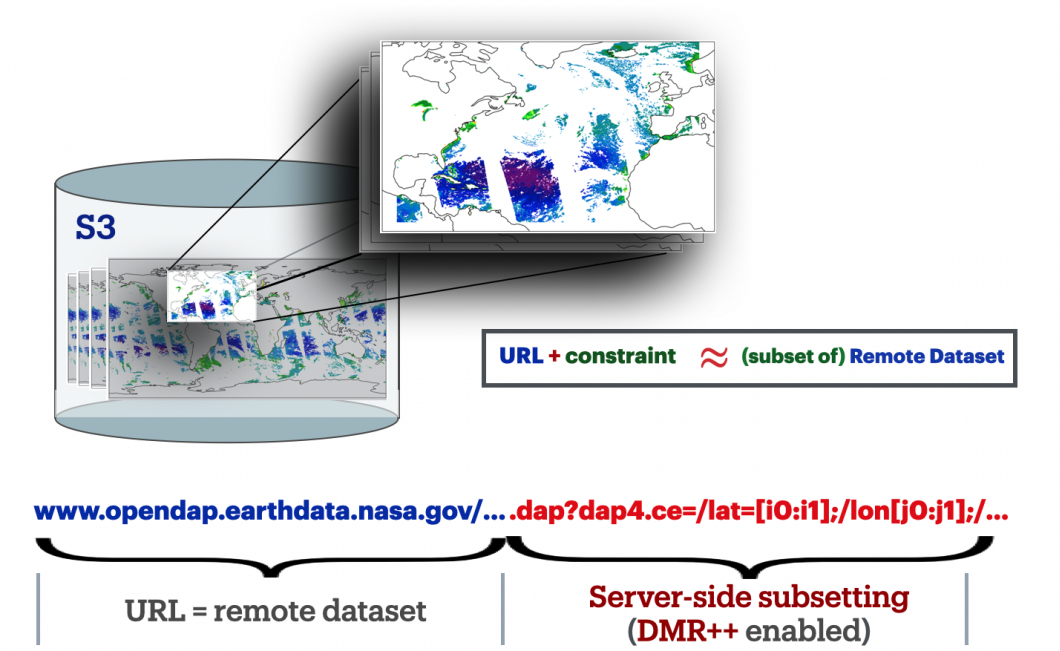

FIGURE 1. A very core concept of OPeNDAP is that 1 URL = 1 dataset and 1 URL + query parameters = a Subset. Parallelizing this approach with Xarray and/or Pydap can lead to an improved performance, while reducing the amount of data streamed compared to full data downloads.

Features	DAP2	DAP4
Pure metadata response	✔ (.das and .dds)	✔ (.dmr)
HTTP transfers	✔	✔
Chunked responses	✖	✔ (.dap)
Streamable response	✖	✔ (.dap)
Hierarchical Data	✖	✔

TABLE 1. Comparison of several features across the two OPeNDAP protocols.

THE DAP4 OPENDAP PROTOCOL

There exist several servers that implement an OPeNDAP protocol, such as with `Hyrax`, `DODS`, `ERDDAP`, or `THREDDS`. Rather than focusing on each individual OPeNDAP server, this article focuses entirely on the OPeNDAP protocol.

There are two versions of the DAP protocol that OPeNDAP servers can implement: the original `DAP2` (previously known as `DODS`, see *Gallagher et al.* (2011)), and the relatively newer `DAP4` (see *Caron et al.* (2016)). `DAP4` was developed over a decade ago to address the growing scale and complexity of data archives, and it is widely used by NASA to provide access to large and heterogeneous data archives for both on-premises and Cloud storage. Currently, two major OPeNDAP server implementations support the DAP4 protocol: the **Hyrax** data server developed by OPeNDAP Inc., and the **THREDDS** data server developed by Unidata. For more information about the `DAP4` protocol, and how it compares with the `DAP2` protocol, you can revisit the DAP4 documentation.

From a user perspective, the DAP4 protocol is a transmission protocol between two distributed systems: the server close to the data, and the user's code, i.e client API, in this case `Xarray` and/or `Pydap`. To safely transmit data across distributed systems, there needs to be an agreement between the two systems regarding what and how many bytes are being sent, and how to reconstruct the received data. The `DAP4` protocol enables sending and receiving data following a subset-driven approach via the use of constraint expression (see figure 1).

How does it work in the background? Given an OPeNDAP URL that points to a remote file, a user "*opens*" the dataset in `Xarray` creating a *Lazy* metadata view of the remote file, describing its content: variable names, sizes, types, etc. This is possible due to a metadata request/response sent in the background between the OPeNDAP server and `Pydap`, i.e. the `.dmr` (see table). The `.dmr` is a *consolidated xml text* file that describes the entire contents of the remote file, which `Pydap` parses to create a *Python Dataset representation*. `Xarray` takes the Pydap object and eagerly downloads all the dimension array data from the remote file. The end result of the `xr.open_dataset` is the visual representation in Figure 2 (note that **lat** and **lon** have been downloaded into memory).

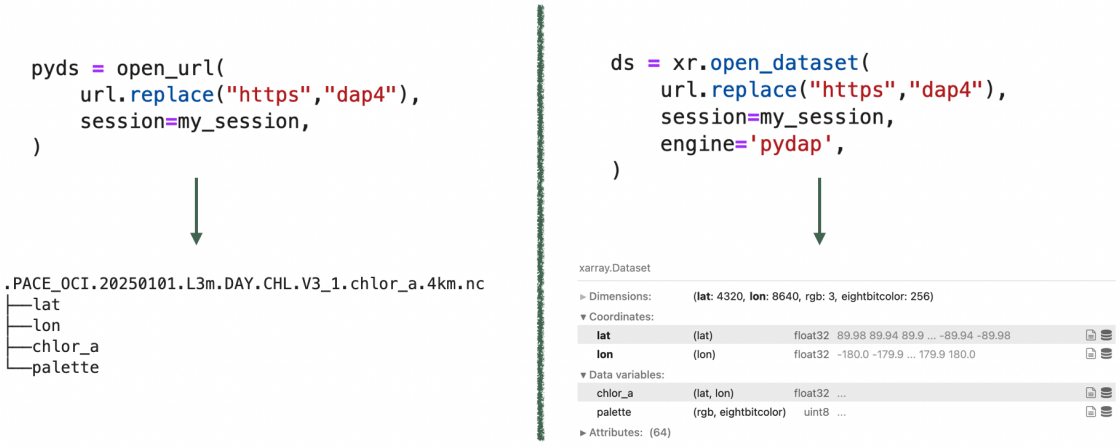

FIGURE 2. Two approaches to *open an opendap url*. With Xarray, the user can specify *pydap* as the engine, or can use *Pydap* directly. In both scenarios, the *.dmr* of the OPeNDAP response will be downloaded which enables to create the Python dataset representation of the remote file. *Xarray* additionally downloads dimension data lat and lon into memory, resulting in slower processing time. **NOTE**: Replacing *https* with *dap4* only instructs the Pydap to treat the url as a DAP4 url. DAP4 is NOT an alternative protocol to https.

When opening a remote dataset using `Pydap` directly, only the metadata `.dmr` is downloaded to create the dataset representation, resulting in faster response times when compared with `Xarray` (figure 2) since no array data is downloaded. But pure `Pydap` can be used to accelerate the `Xarray` dataset generation: With `Pydap`, one can create and add constraint expressions to the original OPeNDAP URL, to send the remote server a request to subset by variable name (`Xarray` has an `drop_variables` method, but these variables are dropped after the entire dataset is created, so there is no actual speed gain. The constraint expression effectively reduces the number of variables to parse by the metadata parser before the Dataset is created. Of course, one needs to know the all the variables first, in order to create the constraint expression. The speed-up from using the constraint expression to filter by variable name when creating the `Xarray` dataset can be significant

when the remote file has $\sim O(1000)$ variables or $\sim O(10)$ dimension arrays.

With `Xarray`, visualizing or saving a file to disk triggers data computation. In the context of remote data access, data is downloaded from a remote OPeNDAP server when a user visualizes a variable, saves to disk, or manually triggers an algebraic computation via `.compute()`. To achieve the data download, Pydap creates and sends a binary data request by appending a `.dap` to the OPeNDAP URL, and *instructing the server* to only send the bytes associated with the desired variable via a **Constraint Expression** (see figure 1). The OPeNDAP server then subsets the data and sends only the bytes requested as a serialized, streamable, and chunked "`.dap`" response. `Pydap` deserializes the binary data sent by the server, turning it into a `NumPy` arrays of the data type and size declared in the Dataset. At this point, the data is accessible by `Xarray` to store, visualize, or operate on. With `Xarray` this process can be parallelized to download data across multiple OPeNDAP urls.

When using `Xarray`, a user gets the most performant data access when the subsetting is done by the OPeNDAP server close to the data of interest, and `Xarray` parallelizes the workflow. However, in many cases, it can be shown that `Xarray` first downloads the entire array from the OPeNDAP server and then subsets, unnecessarily downloading more data that is needed leading to subpar performant access.

To ensure one gets the best performance from an Xarray+OPeNDAP workflow that involves downloading remote data available across multiple URLs, consider the following recommended guidelines:

(1) Keep the number of data requests at a minimum. Every http-request is time-expensive when data is behind authentication. When using `Xarray` and `Pydap`, consider newer versions (see figure 3).
(2) When aggregating multiple files via `Xarray`, make sure to defined a `chunks={**dim_slices}` parameter when creating the dataset, **to make sure OPeNDAP performs the subset and NOT Xarray**. Each dimension is chunked to match the size of the subset along that dimension.
(3) When the remote data has coordinates that are NOT dimension (i.e. **curvilinear coordinates**), avoid aggregating multiple URLs with Xarray. **Xarray will download into memory every coordinate from each URL!**. You can use constraint expressions to drop the coordinates form all but a single remote file, before aggregating into a single Dataset. See benchmarks for `ECCOv4` and `DAYMET` data.

Overall, a data request acts as a single Chunk of data. Too many requests (i.e. too many small chunks) will result in very poor performance. Since the OPeNDAP protocol supports sending multiple or all variables packed along a single request, one could assume that a single request is the most performant approach. However, a single large chunk (i.e. a single dap request) can also be too big and lead to timeout errors. This is the unique scenario that can happen when the URL points to an *NcML Aggregation*, and triggering a single request may download Terabytes of data. In that case is better to parallelize the data download, one request per variable.

Starting with `Pydap=>3.5.4` and `Xarray>=2025.10`, all dimension data is downloaded in a single request! This new feature significantly reduces the time it takes for Xarray to create a Dataset as it can be seen in Figure 3. However, downloading non-dimension data via `Xarray` is done per variable. This is optimal for NcML aggregations but NOT for data across multiple remote URL endpoints. The latest release of `Pydap` now supports downloading a single dap response with multiple variables in it, and can also pass along the slicing along dimensions to the server. `Pydap` does

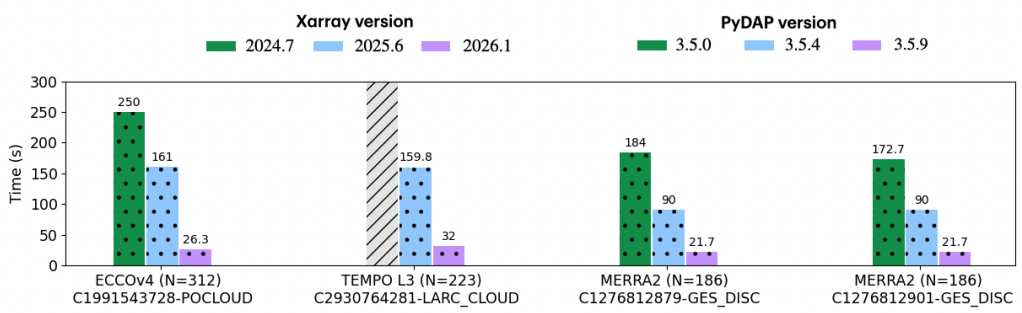

FIGURE 3. Improvements in the way Xarray+Pydap can aggregate N remote datasets over the various software upgrades. Gray bar implies aggregation was not yet supported (older version). See more details in Table 2.

Python	PyDAP	Xarray
3.9	3.5.0	2024.7.0
3.10	3.5.4	2025.6.1
3.13	3.5.9	2026.1

TABLE 2. Several software version installations used.

NOT aggregate, instead it stream earn URL into its own separate NetCDF4 dataset, avoiding the complex logic that is required to aggregate data. Once data is stores local to the user, one case aggregate the files safely and performantly, if the data allows aggregation (swath data cannot in general be aggregated into datacubes).

In the following section, we describe benchmarks that illustrate data access across multiple remote endpoints (URLs) with `Xarray` and `Pydap`.

BENCHMARKS

To best highlight the impact of *Best Practices* described in the previous section when accessing heterogeneous and hierarchical data across many remote OPeNDAP endpoints, we benchmark the total download time it takes to download data of interest. For these benchmarks, the following assumptions are taken

- Data and their OPeNDAP URLs are findable via NASA's CMR.
- Stable network access ($\sim O(100)$ Mb per second).
- The focus is on data-proximate subsetting. The nature of the subset will be either or both by variable name and by spatial region.
- Familiarity with `Pydap` and `Xarray`. Searching for granules, authenticating, and creating constraint expressions do not contribute significantly to net downloading time.

Methodology. For the main benchmarks described below, **only the newer version release will be used**, namely `Pydap=3.5.9` and `Xarray=2026.6.1` (see Table 2). A comparison in performance between different versions of `Pydap` and `Xarray` is shown in figure 3.

As previously described, data download using `Xarray` takes place in two sequential steps:

(1) **Data aggregation**: (`xr.open_mfdataset(urls, engine="pydap", ..., chunks=...,)`). In this step, all metadata and dimension data aggregated. However, not all data can be aggregated, and older software versions do not support aggregating some currently supported

API Approach	Meaning
`Xarray + Pydap`	Use Xarray to generate a dataset, specifying Pydap as engine. No variable selection, no spatial indexing.
`Xarray + Pydap (ce)`	Use Xarray to generate a dataset, specifying Pydap as engine. Use constraint expressions in URL to filter variables, and do '`.isel()` with Xarray.
`Pydap`	Pure Pydap API ($\geq$ 3.5.9) used to stream a dap response into file. Allows for variable selection and indexing by dimension names.

TABLE 3. The three different approaches for downloading data with Xarray and Pydap.

Project	Format	Concept Collection ID	URLs	Total Size (Gbs)	Subset by Variable	Subset by Dim. Index	Down. Size (GBs)
ECCOv4	NetCDF4	`C1991543728-POCLOUD`	313	14	✔	✔	3
TEMPO	NetCDF4	`C2930764281-LARC_CLOUD`	223	346.7	✔	✔	0.8
SWOT	HDF5	`C2799438313-POCLOUD`	150	3.9	✔	✖	0.2
Callipso	HDF4	`C3463063995-LARC_CLOUD`	120	5.14	✔	✖	0.4
TEMPO L2	NetCDF4	`C3685896872-LARC_CLOUD`	340	122	✔	✖	3.3
MERRA2	NetCDF4	`C1276812879-GES_DISC`	186	575.5	✔	✔	28
MERRA2	NetCDF4	`C1276812901-GES_DISC`	186	357.67	✔	✔	23.9
DAYMET	NetCDF4	`C2531982907-ORNL_CLOUD`	99	75	✔	✔	0.4
PACE	NetCDF4	`C3620140255-OB_CLOUD`	363	54	✔	✔	5.7
ECOSTRESS L2	HDF4	`C2076114664-LPCLOUD`	199	147.50	✔	✖	94

TABLE 4. Datasets used for benchmarking and relevant data features.

collections. In this step, providing constraint expressions (CEs) and additional parameters, can result in 10× faster performance even with older software versions.

(2) **Bulk data download**: (`ds.to_netcdf(**args)`). In this step, all relevant non-dimension data download to disk. Performance using `Xarray` is mostly independent of the version used. All data downloaded is stored into a single file on disk. In all cases, the preferred format was `NetCDF4`.

Since we are interested in `Net Download Time`, when presenting the timing results for data download using `Xarray`, we consider the sum of these two data download steps.

Below, we further summarize useful information about the benchmarks:

- All code was performed in Jupyter notebooks executed within SciServer's compute environment, hosted at Johns Hopkins University. This provides network consistency across tests and python environments.
- Timing data represents "*cold reads*" i.e. first read from a remote Hyrax server (NASA hosts several instances of Hyrax across the DAACs).
- Only the `DAP4` protocol was considered.

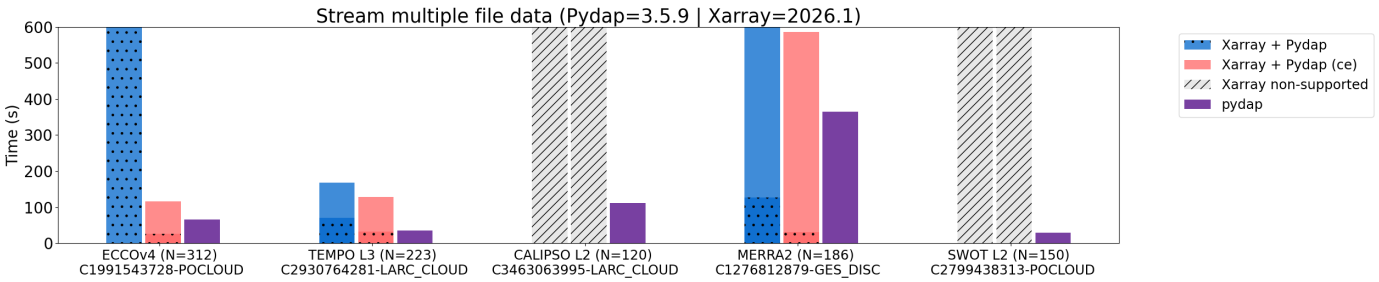

FIGURE 4. Benchmarks for net total download time, including dataset aggregation+generation when using Xarray. The use of constraint expression (ce) in Xarray and Pydap involves variable filtering and chunking when creating the dataset (see step 2 described in the Methodology).

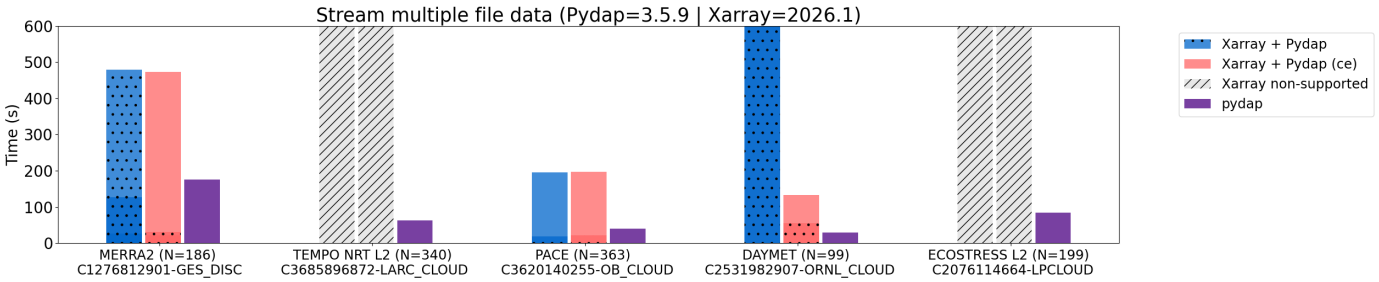

FIGURE 5. Additional benchmarks for total download time, including dataset aggregation+generation when using Xarray. The use of constraint expression (ce) in Xarray and Pydap involves variable filtering and chunking when creating the dataset (see step 2 described in the Methodology).

RESULTS

In every scenario, streaming data available from OPeNDAP via `Pydap` yields the faster downloading times. The `Pydap` workflow parallelizes streaming individual dap responses storing them into individual NetCDF4 files. This workflow does not rely on data aggregation, and is faster because it streams all the variables within an individual dap response. The `Xarray` approach (with `Pydap` as the backend engine) can yield comparable downloading times, but it still relies on data aggregation. In addition, since each non-dimension variable is downloaded as an independent, dap response, the `Xarray` workflow was closest to matching the performance of `Pydap` in the cases where only 2 or 3 non-dimension array data was downloaded.

Figure 4 shows that in some cases, the aggregation step (proportion of each dotted bar) can take a significant amount of time of the total time to download data, even when using newly updated software. In some cases, like in the case of Level 2 (swath) data, the remote granules cannot be aggregated into a single dataset object due to inconsistent dimension sizes (dashed gray bars).

If a remote URL contains 100 non-dimension variables, `Xarray` downloads 100 independent dap responses per granule. When data is accessed across N remote URLs, 100xN dap responses are downloaded, deserialized, and decoded, potentially overwhelming the remote server and the (dask) scheduler running on the client machine. Thus, when using `Xarray`, making use of Constraint Expressions (ce) can lead to improved performance in both steps of the download process. This is exemplified by comparing the Xarray workflow with and without the use of constraint expressions

(blue and red bars in figures 4 and 5).

Summary

There are three aspects that are crucial to download data available through OPeNDAP:

- Update software to latest stable release.
- Use of Constraint Expressions to subset the data of interest before downloading any data.
- Reduce the number of OPeNDAP requests as much as possible.

The last two aspects are not mutually exclusive but in general complement each other. For example, Pydap helps build constraint expressions and also reduces the amount of dap request per URL, by retrieving all data of interest within the same data request. However, this workflow is only recommended for non-NcML aggregations which require parallelizing downloading and deserializing binary dap responses across N URLs.

References

Caron, J., E. Davis, D. Fulker, J. Gallagher, D. Heimbigner, N. Potter, and M. Jimenez (2016), Data access protocol 4 (dap4) v1.0, doi:10.5281/zenodo.14015294.

Gallagher, J., N. Potter, T. Sgouros, S. Hankin, and F. Glenn (2011), The data access protocol — dap 2.0, doi:10.5281/zenodo.10794666.

Hoyer, S., and J. Hamman (2017), xarray: N-D labeled arrays and datasets in Python, *Journal of Open Research Software*, *5*(1), doi:10.5334/jors.148.

Jimenez-Urias, M. A., and J. Gallagher (2026), It's all about dmr++, doi:10.5281/zenodo.19139934.

Virapongse, A., and J. Gallagher (2025), Streaming data for faster access at a reduced cost: Opendap services for data providers, doi:10.5281/zenodo.17418589.