Author Comment (AC1) in response to Referee Comment RC1

Dear Dr. Jelenak,

We would like to thank you for your comments. They will help us improving the clarity and completeness of our paper. Below, we address each of your comments individually.

Our answers are in **green** below.

Section 2.3 would benefit from a clear explanation of the software stack employed and how the data interpretation differs from one layer to another. First six access methods, top to bottom labels of the y-axis in Figure 8, all depend on the HDF5 library for actual I/O operations but this is not mentioned as background explanation.

We agree that a clearer explanation of the software stack is needed. We will add the following figure in Section 2.3 to clarify the layered architecture:

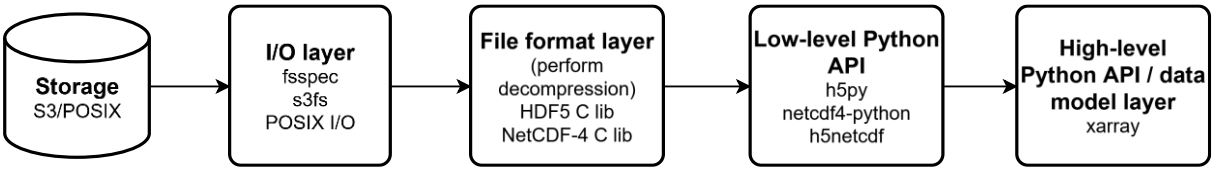

Figure 2: Data access and file-format layers for netCDF-4 over S3/POSIX

This figure will be accompanied by the following text:

"All Python libraries evaluated in this study for reading netCDF-4 files rely on the HDF5 C library for actual I/O operations and decompression. The Fig 2 shows the software stack:

At the lowest level, the I/O layer depends on the storage backend, the HDF5 C library handles all I/O operations, chunk retrieval, and decompression. Built on top of HDF5, the netCDF-4 C library adds the netCDF data model, including dimensions, coordinate variables, and attribute conventions.

This study evaluates the following four python libraries to read datasets in the netCDF-4 format:

- NetCDF4-python (Unidata): a Python interface to the netCDF-4 C library (version 1.7.2 used in our test), which itself depends on the HDF5 C library. It exposes the netCDF data model directly to Python while delegating all I/O and decompression tasks to the underlying C libraries;

- H5py library (Collette, 2013): is a Python interface to the HDF5 library (version 3.12.1 used in our test) Since netCDF-4 files are valid HDF5 files, h5py can read them, although it does not automatically interpret netCDF conventions;
- H5netcdf is a pure Python package that use h5py as its backend while adding netCDF aware functionality (dimension handling, attributes). It avoids the dependency on the NetCDF-4 C library;
- Xarray library (Hoyer and Hamman, 2017) provides a high-level data model interface (including labelled arrays and datasets) and delegates I/O operations to backend engines such as h5netcdf or netcdf4-python. Xarray itself does not perform any I/O or decompression; it only interprets the data returned by its backends (version 2024.9.0 used in our test);
- Chunkindex: our custom chunk-indexing package described in section 3.

"

Line 143: h5py is a Python interface to the HDF5 library, not directly to the HDF5 file format.

We will revise Line 143 from:

"H5py library (Collette, 2013): is a Python interface to the HDF5 binary data format"

to:

"H5py library (Collette, 2013): is a Python interface to the HDF5 library"

Line 144: Mention of h5netcdf would benefit from the more background context. What functionality brings h5netcdf Python package given that both Netcdf4-python and h5py are already used?

We agree that additional context is warranted. We will expand Line 144-145 to explain that h5netcdf is a pure-Python alternative to netcdf4-python for reading netCDF-4 files. It uses h5py as its backend and provides netCDF-aware functionality (dimension handling, attributes, etc.) while avoiding the dependency on the NetCDF-4 C library.

Line 155: Perhaps the labels in the results figures should be "xarray_nczarr" to accurately reflect the software used for this case because xarray adds an overhead for its interpretation of the data.

Thank you for pointing this out. There was an error in our text: Xarray was not used to read the ncZarr file, the ncZarr test used the Zarr library directly. The text has been corrected.

Depicting the 32 kB blocks in Figure 2 would help readers understand the zran concept easier.

We will revise Figure 2 to explicitly depict the 32 kB windows associated with each index point.

Line 295: There is no explanation why the Deflate compression was not used for the Zarr version of the netCDF-4 data because it is available in Blosc. LZ4 decompression is faster than Deflate so the Zarr runtime results are unnecessarily less comparable with those getting data from the netCDF-4 files.

In our experiments, we used Zarr's default compression settings (Blosc with LZ4 at level 5), which is the typical configuration users would encounter. We acknowledge that this introduces a confounding variable when comparing with deflate-compressed netCDF-4 data.

We will add the following discussion in Section 4:

"This choice reflects typical real-world usage, as most users rely on default settings when converting data to the Zarr format. However, it should be noted that LZ4 decompression is generally faster than deflate decompression, which may contribute to Zarr's performance advantage in some of our experiments. Using deflate compression within Blosc would have provided a more direct comparison with the deflate-compressed netCDF-4 files."

Line 332: Suggest to pick one spelling of the word "connexion"/"connection" since both are currently present in the paper.

Thank you for catching this inconsistency. We will standardize the spelling to "connection" throughout the manuscript.

Using fsspec via h5py and HDF5 library involves reading data by a fixed amount, with 5 MiB as the fsspec's default (I think). This means the data from the netCDF-4 files were either read more than needed per single request, or less per single request, hence, requiring multiple requests. Both cases contribute to a larger total runtime. Fsspec also has several different caching mechanisms for already accessed parts of a file, which can also impact runtime. Sections 5.2 and 5.3 do not discuss the configurations of fsspec or s3fs and their effect on the measured results.

The following text will be added to Section 5:

"For the remote and S3 experiments, netCDF-4 and ncZarr datasets were accessed using fsspec (version 2024.9.0) with default configuration settings. The default block size for fsspec is 5 MiB, meaning data is read in 5 MiB increments regardless of the actual amount requested. This can lead to over-fetching (reading more data than needed) or multiple sequential requests when accessing data spanning multiple blocks, both of which contribute to increased runtime.

Fsspec offers several caching strategies (e.g., read-ahead caching, block caching, whole-file caching) that can significantly impact performance. Readers should note that the results presented here are specific to the default fsspec configuration, and optimal settings may vary depending on the use case, access pattern, and network conditions."

There is no mention of HDF5 library's Read-Only S3 (ROS3) virtual file driver which could have been used for the netCDF-4 files in the S3 store. Did the authors consider it and decided not to use? It would be good if this was noted in the paper.

Thank you for bringing up the ROS3 virtual file driver. At the time we began the implementation of chunkindex, we were not aware of the ROS3 driver. We will add a note in Section 2.3 mentioning this alternative approach.

The following text will be added to Section 2.3:

" It should be noted that the HDF5 library also provides a Read-Only S3 (ROS3) virtual file driver, which enables direct access to HDF5/netCDF-4 files stored in S3-compatible object stores without requiring fsspec or s3fs. Although the ROS3 driver was not evaluated in this study, it represents a promising alternative for accessing netCDF-4 files on cloud storage."

Author Comment (AC2) in response to Referee Comment RC2

Dear Dr. Jones,

We greatly appreciate your detailed and constructive review of our manuscript that has helped us identify areas for improvement. We are pleased that you found our contribution valuable and recommend it for publication. Below, we address each of your comments.

Our answers are in **green** below.

Major Comments

I think it's worth being more specific about how the components of the benchmarking options fit together. For example, Xarray does not do any I/O or decompression so it doesn't make sense to include it as analogous to netcdf4-python, h5py, or chunkindex. A figure such as the one at the start of https://tutorial.xarray.dev/intermediate/remote_data/remote-data.html could help for distinguishing which libraries perform I/O, which perform decompression, which provide a Python wrapper, and which only provide a data model.

We agree that a clearer explanation of the software stack is needed. We will add the following figure in Section 2.3 to clarify the layered architecture:

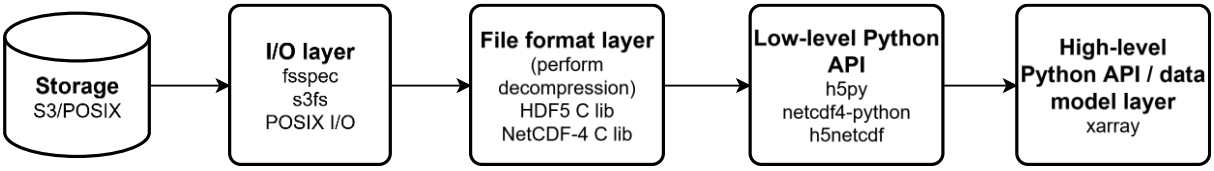

Figure 2: Data access and file-format layers for netCDF-4 over S3/POSIX

This figure will be accompanied by the following text:

"All Python libraries evaluated in this study for reading netCDF-4 files rely on the HDF5 C library for actual I/O operations and decompression. The Fig 2 shows the software stack:

At the lowest level, the I/O layer depends on the storage backend, the HDF5 C library handles all I/O operations, chunk retrieval, and decompression. Built on top of HDF5, the netCDF-4 C library adds the netCDF data model, including dimensions, coordinate variables, and attribute conventions.

This study evaluates the following four python libraries to read datasets in the netCDF-4 format:

- NetCDF4-python (Unidata): a Python interface to the netCDF-4 C library (version 1.7.2 used in our test), which itself depends on the HDF5 C library. It exposes the netCDF data model directly to Python while delegating all I/O and decompression tasks to the underlying C libraries;
- H5py library (Collette, 2013): is a Python interface to the HDF5 library (version 3.12.1 used in our test) Since netCDF-4 files are valid HDF5 files, h5py can read them, although it does not automatically interpret netCDF conventions;
- H5netcdf is a pure Python package that use h5py as its backend while adding netCDF aware functionality (dimension handling, attributes). It avoids the dependency on the NetCDF-4 C library;
- Xarray library (Hoyer and Hamman, 2017) provides a high-level data model interface (including labelled arrays and datasets) and delegates I/O operations to backend engines such as h5netcdf or netcdf4-python. Xarray itself does not perform any I/O or decompression; it only interprets the data returned by its backends (version 2024.9.0 used in our test);
- Chunkindex: our custom chunk-indexing package described in section 3.

"

It's worth mentioning that the benchmarks only explore a small part of the possible parameter space for loading NetCDF data over the network, but still represents an important step forward.

We agree and will add the following text to the discussion in Section 5:

"It is important to acknowledge that the results presented here explore only a small portion of the possible parameter space for reading netCDF-4 data over the network. The performance of I/O operations in cloud environments is influenced by numerous factors, including software versions, configuration settings (e.g., fsspec caching strategies, concurrency limits, NetCDF caching strategies), network conditions, and server-side configurations. Similarly, newer versions of libraries like Zarr-python may exhibit significantly different performance characteristics compared to the versions tested here. While our benchmarks provide a valuable snapshot of performance for common configurations, readers should consider these variables when extrapolating results to other contexts."

The configuration for fsspec has a massive impact on performance, for example see https://agu2025.workshops.geojupyter.org/modules/data-in-the-cloud/#local-vs-cloud-

performance-comparison which shows a 10x difference between default (read-ahead) caching vs. block caching.

The following text will be added to Section 5:

"For the remote and S3 experiments, netCDF-4 and ncZarr datasets were accessed using fsspec (version 2024.9.0) with default configuration settings. The default block size for fsspec is 5 MiB, meaning data is read in 5 MiB increments regardless of the actual amount requested. This can lead to over-fetching (reading more data than needed) or multiple sequential requests when accessing data spanning multiple blocks, both of which contribute to increased runtime.

Fsspec offers several caching strategies (e.g., read-ahead caching, block caching, whole-file caching) that can significantly impact performance. Readers should note that the results presented here are specific to the default fsspec configuration, and optimal settings may vary depending on the use case, access pattern, and network conditions."

The version of software libraries would also have a profound difference on the results. For example, https://earthmover.io/blog/i-o-maxing-tensors-in-the-cloud shows a 10x improvement from Zarr-Python 2 to Zarr-Python 3). The concurrency configuration for Zarr-Python and whether obstore or fsspec is used to interact with obstore can also have a >3x impact.

It's a good remark, the following text will be added to the discussion at the end of Section 5:

"The performance of I/O operations in cloud environments is influenced by numerous factors, including software versions, configuration settings (e.g., fsspec caching strategies, concurrency limits, NetCDF caching strategies), network conditions, and server-side configurations. Similarly, newer versions of libraries like Zarr-python may exhibit significantly different performance characteristics compared to the versions tested here. While our benchmarks provide a valuable snapshot of performance for common configurations, readers should consider these variables when extrapolating results to other contexts."

The specific compression used would also likely impact the results; it's unclear why the compression used for the Zarr data differs from the NetCDF4 data.

We acknowledge that using different compression algorithms (Blosc with LZ4 for Zarr vs. deflate for netCDF-4) introduces a confounding variable. We used the default compression settings for each format to reflect typical real-world usage. We will add the following discussion in Section 4:

"Within Zarr, the default compression method is activated: the meta-compression Blosc with LZ4 compressor at level 5 and shuffle activated. This choice reflects typical real-world usage, as most users rely on default settings when converting data to the Zarr format. However, it should be noted that LZ4 decompression is generally faster than deflate decompression, which may contribute to Zarr's performance advantage in some of our experiments. Using deflate compression within Blosc would have provided a more direct comparison with the deflate-compressed netCDF-4 files. "

People may want to optimize chunking exactly for a specific use-case, which may warrant data transformation and would be worth mentioning. For example, sub-chunking would not be able to provide the same performance for time series analyses as having a single chunk aligned perfectly along the region of interest.

This is an important clarification that is missing in the current manuscript. We will add the following text to the conclusions:

"It is important to emphasize that sub-chunking is primarily designed as a practical solution for accessing existing archived datasets without the need for costly reformatting or re-chunking. In scenarios where the access pattern is known in advance and consistent, optimal re-chunking of the data (e.g., aligning chunks with time series for temporal analysis) will always yield better performance than sub-chunking.

However, for large archives already stored with suboptimal chunking strategies (such as large spatial chunks for time-series access), sub-chunking offers a significant performance improvement over reading full chunks, while avoiding the prohibitive costs of duplicating and restructuring petabytes of historical data."

I recommend mentioning the downsides as well as the benefits of the sub-chunking approach, for example:

- chunk indexes can get out of sync if the data in the original file is modified.

- Many libraries coalesce adjacent range requests to match the optimal get request size for S3 APIs (~4 MB), which is not supported when using sub-chunking.

Thank you for identifying these important limitations. We will add a new subsection (Section 3.6 "Limitations") with the following text:

"

3.6 Limitations

While the sub-chunking strategy offers significant performance benefits for specific access patterns, it also has certain limitations that users should be aware of:

- Index synchronization: The external index file is tightly coupled to the contents of the compressed chunks in the original netCDF-4 file. If the original file is modified (e.g., appended to or rewritten), the index will become out of sync, potentially leading to corrupted data retrieval or decompression errors.
- Request coalescence: Many cloud-optimized libraries (e.g., fsspec, s3fs) implement request coalescing, where multiple adjacent or nearby small read requests are merged into a single larger HTTP GET request to optimize throughput and reduce latency overhead. The current implementation of chunkindex issues individual range requests for each sub-chunk. While this minimizes data transfer volume, it may result in a higher number of HTTP requests compared to reading full chunks with coalescing, which could be suboptimal in high-latency network environments or when many adjacent sub-chunks are requested sequentially.
- Index overhead: Creating the sub-chunk index requires reading and decompressing the entire dataset once, representing an initial computational cost. Additionally, the index files consume storage space. In our experiments, the index size was approximately 16% of the original data volume. This overhead must be weighed against the expected savings in data transfer and access time.

"

Minor Errors

S3 APIs allow range requests, so it is not required to read entire objects (Line 135)

We will revise Line 135 from:

"Unlike POSIX storage, which offers byte-level access, S3 provides object-level access, meaning entire objects must be read or written."

to:

"Unlike POSIX storage, which offers byte-level access, S3 provides object-level access. However, S3 APIs support HTTP range requests, allowing partial object reads without downloading the entire object."

The Zenodo upload for the performance tests is missing the configuration file that was used for the paper.

All configuration files that was used for the paper are in test_perfo directory.

Potential Next Steps

It should be possible to only use the sub-chunking for requests where it offers a performance benefit (small chunks) and use the full chunks for larger requests via a coalescence mechanism.

We will add this suggestion as an outlook in Section 6 :

"A future evolution of the library that could optimize performance may be a hybrid approach that automatically selects between sub-chunking and full-chunk access based on the request and chunk size. "

It's still necessary to adapt workflows to use this technology, because it is not a drop-in replacement for h5py, etc. An Xarray backend, integration with a NetCDF-4, and/or integration with Kerchunk or VirtualiZarr would greatly improve accessibility.

We will add the following text to the outlook in Section 6:

"To facilitate adoption, future development will aim to integrate chunkindex into Kerchunk or VirtualiZarr, providing a unified solution for optimized cloud access to legacy archives."

Support for Zarr V3 in the chunkindex store should dramatically improve performance as a fully async implementation.

Thank you for the suggestion. The following text will also be added to the outlook in Section 6:

"The current chunkindex implementation relies on synchronous I/O operations provided by h5py. Adopting a fully asynchronous approach, compatible with the Zarr V3 specification, could deliver significant performance gains, especially for parallel and concurrent access patterns in high-latency cloud environments."

Ignorable Nits

It'd be helpful to have self-describing labels on the histograms (e.g., NetCDF via h5py with shuffling)

Thank you for your suggestion regarding self-describing labels on the histograms. At this time, implementing such a feature (e.g., NetCDF via h5py with shuffling) is not planned. While it could be a nice improvement, the time required to redraw the figures would be too significant relative to the minor gain in readability.

Figure 2 would be more legible at a higher resolution

This figure has been recreated.

It would be worth mentioning around L84 that OPeNDAP also implements chunk indexing internally via DMR++

We will add a mention to DMR++ (https://opendap.github.io/DMRpp-wiki/DMRpp.html) as a related approach for chunk indexing in the OPeNDAP context.

Citing the Kerchunk library (Durant et al.; https://github.com/fsspec/kerchunk) may be more appropriate for Kerchunk than Sterzinger et al., 2021

We will include the Kerchunk citation to reference Durant et al. as the primary citation for the library, alongside Sterzinger et al. (2021) for the AGU presentation that introduced the concept.

"Objects are analogous to files in a POSIX file system" would be more technically accurate than "Objects are equivalent to files in a POSIX file system"

We will revise Line 129 to use "analogous" instead of "equivalent."

---

https://github.com/pauldmccarthy/indexed_gzip preceded zran and is likely worth a citation

We will add a citation to indexed_gzip (McCarthy, P.) as prior work in the area of random access for compressed files.

Although it likely came out after this paper was started, it may be worth mentioning Icechunk as a higher-performance alternative to Kerchunk, and possibly an integration point for chunkindex.

We will add a brief mention of Icechunk as a recently developed alternative that could serve as a potential integration point for chunkindex in future work.

It could help to explain in Line 390 that the netCDF4 library cannot read from buffers and therefore requires an open file using operating system machinery.

We will add an explanation noting that the netcdf4-python library requires a file path and cannot read from in-memory buffers, which limits its compatibility with fsspec-based virtual file systems.

It'd be nice to at some point include a sentence or table describing exactly what subset of files chunkindex can be used for (e.g., all HDF5 or just NetCDF4, all compression levels, the specific compressions implemented by zran)

We will add a new subsection in Section 3 to explicitly define the library's applicability scope. The text will be as follows:

"

3.7 Scope of Applicability

The chunkindex library currently supports the following features:

- File Formats: The library supports standard netCDF-4 files and valid HDF5 files;
- Compression: Support is currently limited to the deflate (zlib) compression algorithm, which is the standard for netCDF-4. All compression levels (1-9) are supported. Other compression algorithms (e.g., SZIP) or third-party filters (e.g., Blosc, Zstd, LZF) are not currently supported;
- Shuffle: The HDF5 shuffle filter is supported. However, as discussed in Section 3.2, shuffling changes the byte alignment of the compressed data, which can reduce the precision of the sub-chunk index.

"