Dataplug: Unlocking extreme data analytics with on-the-fly dynamic partitioning of unstructured data

Aitor Arjona
Universitat Rovira i Virgili
Tarragona, Spain
aitor.arjona@urv.cat

Pedro García-López
Universitat Rovira i Virgili
Tarragona, Spain
pedro.garcia@urv.cat

Daniel Barcelona-Pons
Universitat Rovira i Virgili
Tarragona, Spain
daniel.barcelona@urv.cat

This is the author's preprint version of the article available in the proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID) 2024 [1], with DOI 10.1109/CCGrid59990.2024.00069.

***Abstract*—The elasticity of the Cloud is very appealing for processing large scientific data. However, enormous volumes of unstructured research data, totaling petabytes, remain untapped in data repositories due to the lack of efficient parallel data access. Even-sized partitioning of these data to enable its parallel processing requires a complete re-write to storage, becoming prohibitively expensive for high volumes. In this article we present Dataplug, an extensible framework that enables fine-grained parallel data access to unstructured scientific data in object storage. Dataplug employs read-only, format-aware indexing, allowing to define dynamically-sized partitions using various partitioning strategies. This approach avoids writing the partitioned dataset back to storage, enabling distributed workers to fetch data partitions on-the-fly directly from large data blobs, efficiently leveraging the high bandwidth capability of object storage. Validations on genomic (FASTQGZip) and geospatial (LiDAR) data formats demonstrate that Dataplug considerably lowers pre-processing compute costs (between 65.5% — 71.31% less) without imposing significant overheads.**

***Index Terms*—data partitioning, Cloud, object storage, scientific data management, unstructured data, data analytics**

I. Introduction

The seemingly "infinite" compute and storage resources available in the Cloud have paved the way for handling unprecedented amounts of unstructured data, marking the evolution from big data to what is termed as *extreme data* [2]. Extreme data is characterized by its increasing volume, speed, variety, complexity and extreme variations in values, which challenge current distributed computing technologies that struggle to cope with such demanding characteristics [3]. For instance, *The Cancer Genome Atlas* [4] openly offers 2.5 petabytes of a multitude of cancer cell genomic data for researchers to analyze, with different unstructured data formats (VCF, BAM,...) and great variation in size (from KB to TB) and complexity.

To analyze these extreme data repositories, researchers must first deal with data partitioning, which is crucial for efficient workload distribution [5] and to fully harness the scalability and parallelism of Cloud resources [6]. We understand *data partitioning* as chunking a large dataset into smaller logical chunks, so that each chunk (or *partition*) can be processed by a distributed worker, thus increasing the scalability of data-parallel applications. For example, a large dataset of 3D geospatial points that describe a region of the Earth's surface could be partitioned into smaller geographic regions, so that we can distribute the workload to process each sub-region separately, increasing the overall parallelism.

However, partitioning extreme unstructured data at this scale is challenging. A common approach for unstructured data is to partition a dataset into smaller even-sized files or objects, or transform it into a "Cloud-friendly" format [7]. This, however, involves reading, pre-processing, and writing back to storage the whole dataset, which can become extremely costly. Moreover, this issue is further exacerbated in Cloud scenarios, where object storage serves as the primary storage for scientific computing in the Cloud [7]. Since objects are immutable, partitioning a dataset into many objects requires re-writing all the data, making this approach inefficient. Finally, arbitrarily setting a fixed partition size without any hints can result in suboptimal performance [5], given the variation in workload requirements. While distributed computing frameworks like Dask or Spark can partition large files in arbitrary byte ranges, this approach is only applicable to generic formats like CSV and it breaks unstructured data formats.

Thus, we argue that static partitioning is not a viable solution for extreme scientific computing. Instead, we advocate for an alternative approach involving read-only format-aware pre-processing enabling *on-the-fly* dynamic partitioning.

Read-only format-aware pre-processing generates indexes and metadata for existing cold unstructured data without overwriting it. This approach eliminates the need to write back the entire pre-processed dataset to storage, addressing the lack of in-place modifications in object storage. The resulting indexes and metadata are used to query the file structure, allowing to dynamically fetch data partitions of any size *on-the-fly*, directly from object storage, enabling fine-grained parallel access to large unstructured scientific data blobs. These indexes are object storage-aware, specifically built for high throughput and concurrent reads using native object storage APIs.

We present *Dataplug*, an extensible framework that implements the *on-the-fly* data partitioning model. Dataplug hides the complexities of unstructured scientific data pre-processing and partitioning, offering researchers a data-driven

pre-processing and dynamic *on-the-fly* partitioning strategies for diverse unstructured scientific data formats. *Dataplug* aims to build an open community of both data providers and researchers, with the goal of enabling efficient analysis of extreme data in the Cloud. The framework provides open-source implementations of pre-processing and partitioning strategies for formats from different domains, inviting developers to contribute or adapt the code to their needs. *Dataplug* enables efficient parallel access to existing unstructured data blobs in their original scientific format. Currently, we support FASTA, FASTQGZIP, and VCF for genomic data, LIDAR for geospatial data, and imzML for metabolomics data, with plans to include additional formats in the future. Furthermore, *Dataplug* is compatible with Cloud-optimized formats like Cloud-optimized Point Cloud [8] and Cloud-optimized GeoTIFF [9]. These formats only support content queries, whereas Dataplug brings to them partitioning semantics that simplify their processing in data-parallel workloads.

Dataplug's objective is to unlock extreme data processing that was previously unfeasible or prohibitively expensive. Empirical evaluations on geospatial and genomics data reveal a 65.5% to 71.31% reduction in pre-processing compute costs, translating to potentially thousands of dollars in savings at the peta-scale. Finally, dynamic partitioning enables dataset-wide re-partitioning at zero cost, allowing one to utilize different partition sizes and to choose the optimal one for each workload.

Dataplug has been developed as part of the NEARDATA [1] project, which focuses on addressing the challenges arising from extreme data scientific in the Cloud and edge. The project deals with data-parallel workloads that require efficient parallel data access to large-scale repositories in the Cloud: genomics epistasis and transcriptomics, metabolomics metabolite annotation, among others. With these targeted use cases, the NEARDATA project, through *Dataplug*, aims to significantly advance the efficiency and scalability of scientific Cloud computing.

Our contributions are:

1) We propose *on-the-fly* dynamic partitioning to enable fine-grained parallel access to large unstructured scientific data blobs, as a replacement to static partitioning at a fraction of the pre-processing cost.
2) We develop the *Dataplug* framework. *Dataplug* acts as an abstraction layer for data partitioning, allowing researchers to define dynamically-sized partitions using different strategies based on workload-specific criteria. *Dataplug* is open source and is publicly available in Github[2].
3) We validate *Dataplug* with industry-standard data formats from genomics (FASTQGZip) and geospatial (LiDAR) domains. We empirically demonstrate that *Dataplug* significantly reduces pre-processing costs up to 71.31% without adding overhead in partition retrieval.

[1]https://neardata.eu/

[2]https://github.com/CLOUDLAB-URV/dataplug

II. BACKGROUND AND MOTIVATION

The synergy between elastic compute resources and object storage has enabled to scale demanding scientific applications in the Cloud. Thanks to elasticity, we can adapt the compute capacity to the data volume to process, instead of adapting the data to a specific cluster size [6]. As a consequence, data partitioning becomes critical for efficiently parallelizing and scaling data-parallel workloads.

Many Cloud-based data repositories rely on object storage as a cost-effective solution for large scientific data archiving [10]. Object storage is convenient for accessing large data blobs in parallel, as partial reads using HTTP GET byte-range allow workers to remotely access data partitions at high bandwidth. Despite this, dealing with data that cannot be efficiently consumed in parallel poses a potential bottleneck that jeopardizes scalability. Hence, it's crucial not only to enable concurrent partial reads, but also *to know which partial reads* (offsets and ranges) *are necessary for workers to access and process a specific logical partition within a large dataset*. Therefore, for unstructured formats that are not easily partitionable, a prior pre-processing is necessary to determine the appropriate logical data partitions.

Cloud computing frameworks: Spark [11] and Dask [12] are commonly used to deploy scientific workloads in the Cloud. They provide built-in abstractions for reading semi-structured, tabular, or array data from object storage (like CSV or Apache Parquet), which are trivial to partition and access in parallel (e.g., a worker can retrieve a set of rows from a large CSV file). However, unstructured data that cannot be represented as a table or an array must be provided as a single unit (e.g. a file) per worker/executor, which limits parallelism or requires ad-hoc partitioning. For instance, Dask's *bag* abstraction provides operations like `map` over a set of files in S3, but further partitioning of those files is not supported out of the box. We see that there is a lack of standardized support for unstructured scientific data formats in general-purpose distributed computing frameworks.

Cloud-optimized file formats: Cloud-optimized formats [9] allow to retreive specific portions of a remote file in an HTTP server without the need to download and filter the entire content. For example, Cloud-optimized GeoTIFF [9] allows to query and retrieve sub-tiles of raster images (e.g. satellite images). Another example is Zarr [13], which is a compressed and chunked file format and library for N-dimensional array data. To accomplish this, Cloud-optimized data formats rearrange the file contents in a specific indexed structure, with metadata describing the structure stored in extensible headers within the file. Users can query the metadata and utilize HTTP GET byte-range requests in object stores to enable parallel data access. However, not all file formats (e.g. FASTQ) support arbitrary re-arrangement of their content and embedded storage of metadata in optional extensible headers. As a result, those formats cannot benefit from a Cloud optimization upgrade. Additionally, transforming datasets in object stores to become Cloud-optimized is costly, as it requires a complete trans-

formation and rewriting of the entire dataset. Kerchunk [14] addresses this issue by indexing legacy array data formats (netCDF4/HDF, GRIB2, TIFF, FITS) to be compatible with Zarr for parallel access from object storage, without applying format transformation. However, it is arguable whether Zarr is only suitable for array data. There is a wide variety of unstructured scientific data formats that are incompatible with Zarr, such as text-based formats (for instance, Variant Calling Format data in genomics), making them unable to benefit from its advantages.

Where does Dataplug *fit?:* In particular, *Dataplug* adresses the following challenges:

1) Despite the increasing support for general-purpose array-like data formats for Cloud object storage (e.g., Zarr), **many scientific data formats are still not adequately addressed**. With *Dataplug*'s approach, we enable efficient partitioning and parallel data access to Cloud object storage for many domain-specific scientific formats as well. In particular, in this article we focus on two representative formats: FASTQGZip from genomics and LiDAR from geospatial — but *Dataplug* also implements dynamic *on-the-fly* partitioning for VCF genomic data and imzML metabolomic data, with plans to support more formats in the future. ***Dataplug* model allows a broader range of formats and workloads to benefit from the advantages of Cloud-based data processing**.
2) ***Dataplug*'s approach represents an advancement over Cloud-optimized formats**, as it eliminates the need for rearranging file contents and metadata embedding in optional headers. *Dataplug* overcomes these constraints through read-only pre-processing, which avoids modifying existing data by decoupling metadata from raw data. This allows to support formats that cannot re-arrange their content and do not have an option for embedded metadata to also benefit from indexing and partitioning, making them equivalent to Cloud-optimized formats (for example, FASTQGZip and LiDAR data, which are discussed in Section IV). Additionally, existing Cloud-optimized formats (such as GeoTIFF, Zarr or COPC [8]) can also benefit from *Dataplug*, as Cloud-optimized formats only offer queries over the content. ***Dataplug* can incorporate partitioning semantics to Cloud-optimized data formats**, being able to abstract the partitioning logic and adapt it to the requirements of each specific workload.
3) *Dataplug* specifically targets domain-specific unstructured scientific data formats, which pose significant challenges for efficient partitioning and parallel access in Cloud environments. Other generic formats (such as CSV or Parquet) are already supported by general-purpose distributed computing frameworks (like Dask or PySpark). In this regard, ***Dataplug* is designed to act as glue code to integrate with these frameworks to facilitate the Cloud adoption for new scientific workloads**.

The following sections provide a deeper description of *Dataplug*'s model and architecture and how the aforementioned challenges are solved.

III. Enabling dynamic on-the-fly partitioning

This section presents the data model of *Dataplug* which enables parallel data access for unstructured scientific formats. In summary, the model is comprised of two concepts: 1) ***Cloud-aware* read-only pre-processing**, and 2) **dynamic *on-the-fly* partitioning**.

A. Cloud-aware pre-processing and indexing

In order to enable parallel access to unstructured scientific data, a pre-processing phase is required. In *Dataplug*, each supported format is required to define a pre-processing method that must be applied to each raw data blob. This pre-processing method is format-specific, and extracts metadata from the raw data blob, such as internal content structure, indices, and attributes.

For semi-structured or tabular formats like JSON or CSV, generic pre-processing methods like schema inference [15] can be applied. However, scientific unstructured data formats require domain-specific tools and techniques to extract valuable metadata. *Dataplug* aims for an extensible approach, which allows for flexibility in metadata and index structure, as well as in the generation process. This approach ensures support of virtually any file format.

We say our pre-processing and indexing approach is *Cloud-aware*, which means that it is specifically optimized for efficient data access in Cloud object storage. *Cloud-aware* indexing focuses on exploiting the high-bandwidth capability of object storage so that partitions can be retrieved using many concurrent HTTP GET byte-range requests over large data blobs. This approach differs from "traditional" pre-processing in the following key aspects:

Read-only preprocessing: With *Cloud-aware pre-processing*, raw data is pre-processed only once, without modifying it, and the extracted metadata is decoupled from the data, stored externally in another object, storage bucket, or database. This approach maintains the original data unmodified, with metadata automatically handled by *Dataplug*. Additionally, it is effective for pre-processing data in object storage where objects are immutable, meaning that in-place modifications are not possible. As a result, pre-processing that is intrusive to the data, such as transformation to a Cloud-optimized format, is not ideal because it requires a complete data rewrite, significantly increasing pre-processing cost and data movement.

Object storage-aware indexes: In object storage, data access relies on HTTP, which introduces significant latency as each chunked read requires a complete HTTP GET byte-range request, compared to, for instance, parallel file systems. To account for this limitation, *Cloud-aware indexes* optimize data indexing by prioritizing large chunks and concurrent reads, exploiting the object store's high bandwidth. By leveraging concurrent requests, non-contiguous data portions can be efficiently retrieved by requesting many HTTP GET byte-range requests in parallel, partially mitigating the high latency inherent to object storage [7].

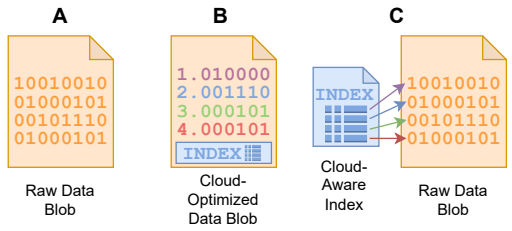


Figure 1: Comparison between Cloud-optimized data formats and *Cloud-aware* data formats. In Cloud-optimized, metadata is embedded in the blob, while *Cloud-aware* pre-processing stores metadata decoupled from the blob.

To summarize, Figure 1 visually represents the indexing difference between Cloud-optimized data formats and our *Cloud-aware* approach. On one side, Cloud-optimized formats 1) re-arrange the blob's content and 2) embed metadata and indexes within the blob. On the other side, *Cloud-aware* indexing 1) leaves raw data unmodified, *as is*, and 2) stores the metadata decoupled from the raw data blob. These two differences are key to why our model is more suitable for Cloud object storage. Since object stores are immutable, transforming files to Cloud-optimized is inefficient, requiring a complete blob rewrite. Instead, our model stores indexes (which are much smaller in volume) in another object, making the pre-processing cost considerably lower.

B. Data slicing: Dynamic on-the-fly partitioning

After pre-processing and extracting metadata, applications may request dynamic *on-the-fly* partitions of scientific datasets that have been pre-processed using our *Cloud-aware* approach. We refer to this process as *data slicing*.

A *data slice* is an entity that represents a lazily-evaluated partition of an unstructured dataset in Cloud object storage. It encapsulates partition metadata (such as byte ranges) and code. When a *data slice* is evaluated, its embedded code runs and fetches the actual partition data content from Cloud object storage using its native APIs, typically through many concurrent byte-range HTTP GET requests. Additionally, the code can apply any necessary corrections to ensure integrity for the data format, such as adding missing headers.

Data slices are generated for a specific dataset by applying a partitioning strategy. In *Dataplug*, each data format can define multiple partitioning strategies, allowing for different criteria or partitioning parameters. Furthermore, users can expand and implement their own strategies to address specific cases for their workloads. Prior pre-processing of the dataset is necessary so that strategies can query the corresponding metadata and indexes for defining *data slices*.

Data slicing does not involve data movement. Instead, partitioning strategies operate on the metadata and indexes generated during the pre-processing stage, rather than directly on the actual data. Ideally, the index data structure is fully loaded in memory for efficient processing.

Once a set of *data slices* is generated, they may be distributed to remote workers. Each worker independently evaluates a *data slice* and loads the corresponding data partition into memory. This parallel and distributed process takes advantage of the synchronization-free parallel access and high bandwidth capabilities offered by object stores. To support remote worker processes, *data slices* must be serializable.

IV. Use Cases

This section presents two unstructured scientific formats as use cases for *Dataplug* from different domains: FASTQGZip for genomic data and LiDAR for geospatial data. We demonstrate the challenges associated with partitioning and parallel access arising from the format's structure combined with the inherent limitations of object storage. We then elaborate on how our data model enables partitioning and parallel access from Cloud object storage for these formats. By using fundamentally different formats, we showcase the flexibility and versatility of *Dataplug*, as it can be adapted to various unstructured scientific data formats across different domains.

A. Compressed genome FASTQGZip data

The FASTQ format [16] is used to represent genome sequence read data and their quality values as text. To reduce data volume, FASTQ files are usually compressed using GZip, resulting in high compression ratios due to the repetitive nature of genome sequence characters.

Challenge: The DEFLATE algorithm used in GZip poses a limitation for parallel access *because it is not possible to decompress arbitrary portions of a GZip file, requiring to process the entire stream from the beginning* [17]. To partition FASTQGZip or any compressed GZip file, a process must create static partitions sequentially as the GZip stream is decompressed.

Solution: We propose the use of *GZip indexing combined with parallel decompression*. Our *Cloud-aware FASTQGZip* utilizes GZiptool [18] to generate an index for a compressed GZip file, allowing for random access decompression. This index provides multiple entry points within the GZip file, each associated with a line number in the uncompressed text. We employ a FASTQGZip *data slicing* strategy based on the number of sequences per partition, leveraging the GZip indexes to determine the appropriate entry points for each *data slice* line range [19]. Our approach is *Cloud-aware*, enabling parallel access to FASTQGZip files stored in the Cloud through concurrent HTTP range requests and parallel decompression, resulting in high throughput.

B. LiDAR point cloud data

The LAS (Laser) binary file format [20] is commonly used for storing LiDAR (Light Detection and Ranging) 3-dimensional point cloud data that represents the Earth's surface morphology. In the LAS format, the point record body is not arranged in any specific order. It typically corresponds to stripe patterns formed by laser measurements during plane or drone sweeps.

Challenge: Geospatial workloads often parallelize data processing by partitioning large surfaces into smaller subregions using bounding boxes [21]. However, *accessing arbitrary land surface regions of a LAS file requires reading and shuffling all point records*, because a point belonging to a particular region can be located anywhere in the file. This limitation prevents parallel access to a LAS file, requiring it to split into multiple files for partitioning. The COPC (Cloud-optimized Point Cloud) format [8] addresses this issue by arranging LiDAR points into nodes forming an octree. The metadata describing the octree and node offsets is stored in extensible LAS headers. Users can then partition a COPC file by issuing multiple spatial queries over the octree to retrieve nodes using byte-range HTTP GET requests. However, adopting the COPC format for existing LiDAR data in object storage requires a complete dataset rewrite. This is because COPC modifies the point record of LAS files and embeds metadata in extensible headers, significantly increasing the complexity and cost of pre-processing.

Solution: Our proposed *Cloud-aware LiDAR* utilizes `lasindex` from LAStools [22] for efficient in-place LiDAR indexing. Our approach uses `lasindex` to pre-process LAS files stored in Cloud object storage to generate a quadtree point index without modifying the original file. By leveraging this index, we implement strategies to define spatial partitions within the LiDAR point cloud and determine the corresponding quadtree nodes for each partition. Unlike LAStools, which requires reading data from a local file system, our *Cloud-aware* approach enables direct retrieval of point nodes from Cloud object storage in parallel. Although *Dataplug* also supports partitioning COPC data, our *Cloud-aware* approach is non-intrusive as it only necessitates read-only pre-processing. This is especially useful for large legacy datasets stored in LAS format since transforming to COPC is more expensive than our read-only pre-processing approach (see Section VI-B1).

V. *Dataplug* Framework Architecture

This section describes the architecture details of *Dataplug*. *Dataplug* allows end-to-end management of the lifecycle of unstructured data: from **data pre-processing and staging** to **data partitioning and delivery** for scientific workloads in the Cloud. *Dataplug* does not require any service to be running, as it only comprises a client-side Python library for users to interact with the framework. We chose Python because it is currently the *de facto* programming language for distributed scientific computing due to its simplicity but also because of its ability to interact as glue code for high-performance low-level scientific applications. The framework is compatible with popular Python distributed computing frameworks such as PySpark, Dask, and Ray, thus achieving portability for many workloads.

A. Dataplug *Cloud architecture and data life-cycle management*

This section describes the *Dataplug* Cloud architecture and the full data management workflow life-cyle. Figure 2 depicts

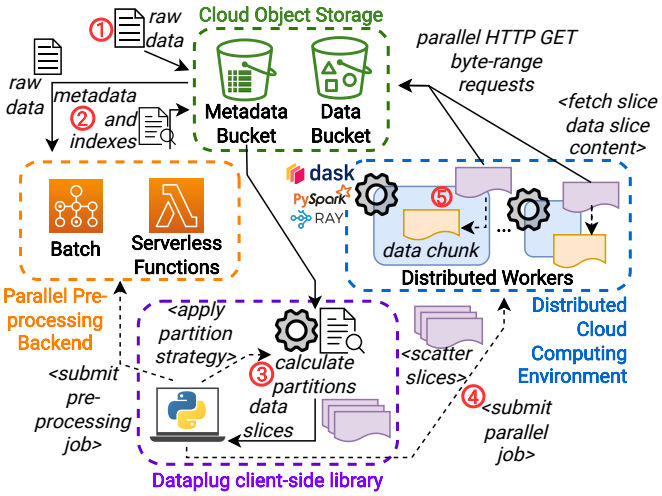


Figure 2: *Dataplug* architecture.

a big picture of *Dataplug*'s architecture.

1) Pre-processing: **Raw data** (1) is stored as objects in a object storage bucket. This data is in its original, unprocessed raw form. Notably, *Dataplug* does not require write permissions to the source bucket, allowing to use read-only public Open Data Repository buckets [10] as data sources.

Dataplug requires a **pre-processing** (2) phase to enable parallel data access to the raw data. When a new object is uploaded, *Dataplug* can leverage storage triggers to automatically launch pre-processing jobs (for instance, using an Amazon S3 trigger to invoke a Lambda function). In this case, *Dataplug* can infer the matching format from the file extension or the `Content-Type` header in order to launch the corresponding *Cloud-aware* pre-processing method for that particular format. Alternatively, users can manually submit pre-processing jobs for cold data, by explicitly specifying which *Cloud-aware* format the raw cold data complies to. During the pre-processing stage, metadata and indexes are generated from the raw data. This extracted information is stored in a separate metadata bucket within the object storage.

Dataplug leverages `joblib` for deploying pre-processing jobs. Regarding pre-processing scalability, *Dataplug* can use Cloud-based scalable `joblib` backends (such as Dask) to deploy and adapt the resources necessary to the raw data volume. Each *Cloud-aware* format can specify pre-processing parameters, like batch or parallel processing, CPU and memory resources and chunk size.

2) Partitioning: After the pre-processing stage, users can leverage *Dataplug* to partition datasets for specific workloads by applying a **partitioning strategy** (3) to define *data slices*. *Dataplug* utilizes the metadata and indexes generated during the pre-processing stage to query and extract the necessary information for defining these *data slices*.

Once the data slices have been defined, the user can proceed to **submit a parallel processing job** (4) using a Python distributed computing framework such as Dask or `ipyparallel`. The user may pass the set of *data slices*

created as input data for the job. The distributed computing framework will handle the deployment of distributed workers and appropriately scatter the *data slices* among them.

Finally, each worker process will be assigned a specific *data slice* as input, allowing them to **fetch the content of the data partition** (5). Leveraging embedded metadata and code, the *data slice* can efficiently perform multiple concurrent byte-range HTTP GET requests over one or more objects in the object store. These requests retrieve data chunks, which are then assembled and processed to perform any necessary corrections. Once the data partition has been retrieved and prepared, it can be passed back to the worker to perform the job.

Regarding partitioning scalability, *dataplug* relies on the underlying distributed computing framework and on Cloud object storage. Distributed workers among many virtual machines or containers can effectively exploit the high bandwidth of object storage by issuing many concurrent byte-range HTTP GET requests when retrieving the data partitions.

B. Dataplug *Workflow: a FASTQGZip format example*

This section shows an overview of *Dataplug*'s programmatic APIs. Using an example for the genomic FASTQGZip format, we illustrate how *Dataplug* interfaces are used to manage pre-processing and partitioning logic using code.

```
# Create Cloud Object reference
co = CloudObject.from_s3(
  FASTQGZip,
  's3://my_bucket/SRR123456.fastqgz'
)

# Preprocess object (this has to be done only once)
backend = AWSLambdaPreprocessor()
co.preprocess(backend,
              sequence_identifier='SRR0000000')

# Apply partition strategy and get slices
data_slices = co.partition(
  partition_num_reads,
  num_seq_partition=1_000_000
)

# Define processing function
def process_sequences(data_slice: GZipTextSlice):
  chunk = data_slice.get()
  ...  # process fastq chunk
  return result

# Submit distributed job with ipyparallel
# Data slices are used as input data
with ipyparallel.Cluster() as cluster:
  view = cluster.load_balanced_view()
  result = view.map(process_sequences, data_slices)
```

Listing 1: Full workflow example for an application using the *Cloud-aware* `FASTQGZip` format with an example of parallel processing using `ipyparallel`.

Listing 1 provides an example of the complete workflow management for the *Cloud-aware* `FASTQGZip` format. Initially, a file is referenced using the key and bucket where it is stored. Here we associate the `FASTQGZip` *Cloud-aware FASTQGZip* format for this particular object.

Pre-processing is then applied to this file. Each supported format in *Dataplug* must implement the pre-processing interface for its pre-processing logic. An implementation of the interface for `FASTQGZip` is shown in Listing 2. Following the example, the referenced file is pre-processed using the `AWSLambdaPreprocessor` backend, which executes jobs on AWS Lambda serverless functions. Here, the job is manually triggered for demonstration purposes, and this step has to be performed once since the generated metadata and indexes are reusable for any partitioning request.

Next, the `partition_num_reads` partitioning strategy is applied to obtain a list of *data slices*, representing lazy-evaluated partitions of the file. The interface for strategies consists of a function with a signature that receives a `CloudObject` as a parameter and returns a list of `DataSlice` objects. The implementation for `partition_num_reads` can be seen in Listing 3. This strategy computes the entry point offsets within the GZip file for each partition, based on the specified number of lines per partition. Alternatively, other strategies can be implemented, such as partitioning based on the total number of chunks rather than the chunk size.

Finally, a parallel job is launched using `ipyparallel`, with the list of *data slices* as the input data for the job. Each user-defined processing function receives a single *data slice*. The interface for *data slices* consists of a class with a `get()` method, which implements the logic to issue multiple byte-range requests to object storage and retrieve the actual data partition. The *data slice* implementation for `FASTQGZip` is shown in Listing 4. A format slice must implement the `get()` method, which will be called by the worker code to evaluate the slice and retrieve the corresponding decompressed FASTQ sequence chunk for that particular slice.

```
@FormatPreprocessor(FASTQGZip)
class FASTQGZipPreprocessor(BatchPreprocessor):
    def preprocess(self, cloud_object: CloudObject)
        -> PreprocessingMetadata:
      blob = self.s3.get_object()
      ...
      return PreprocessingMetadata(gzip_index)
```

Listing 2: FASTQGZip implementation of the *Cloud-aware* pre-processing interface.

```
@PartitioningStrategy(FASTQGZip)
def partition_num_reads(cloud_object: FASTQGZip,
    num_reads: int) -> Iterator[FASTQGZipSlice]:
  gzip_index = cloud_object.metadata
  byteranges = ...
  for byterange in byteranges:
    yield FASTQGZipSlice(byterange)
```

Listing 3: A partitioning strategy function interface implementation for the FASTQGZip format.

VI. Use cases evaluation

In this section, we validate our framework and data model. The main objectives of this validation are 1) demonstrate that ***Cloud-aware* pre-processing significantly reduces costs** both

```python
class FASTQGZipSlice(CloudObjectSlice):
  def get(self):
    blob_chunk = self.s3.get_object(self.byterange)
    fastq_chunk = ...
    return fastq_chunk
```

Listing 4: A *data slice* interface implementation for the FASTQGZip format.

in terms of compute time and storage without introducing partitioning overhead, and 2) demonstrate the benefits of **dyanamic on-the-fly partitioning** (to avoid in-cluster partitioning, and to allow zero-cost re-partitioning to probe for the optimal partition size).

We employ the *Cloud-aware* implementations of the scientific unstructured data formats discussed in Section IV: *Cloud-aware* LiDAR and *Cloud-aware* FASTQGZip.

All validations were conducted on Amazon Web Services in the `us-east-1` (North Virginia) region during 2023. The data is stored in Amazon S3 buckets located within the same region. Further specific configurations, such as EC2 instance type, are detailed for each experiment. Reproducible experiment code can be accessed publicly in Github[3].

A. FASTQGZip compressed genomic data

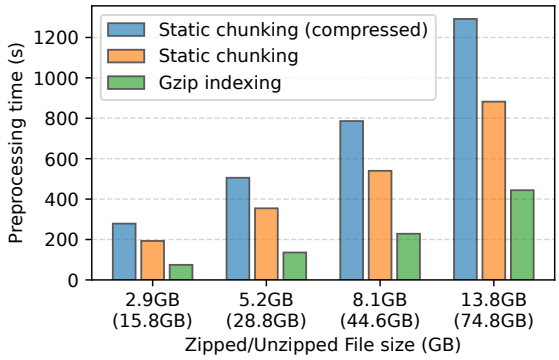


Figure 3: Comparison of pre-processing time to generate static partitions and *Cloud-aware* indexing for different input FASTQGZip data sizes.

1) Pre-processing comparison between Cloud-aware and static partitioing approaches: In this experiment, we want to compare the time required to generate static partitions and the time required to apply *Cloud-aware* indexing for different data volumes of FASTQGZip data.

To generate static partitions, we use a script (Listing 5) that employs the `aws` CLI, `zcat`, and `split` commands. This script reads a compressed file from S3, splits it into partitions with a specified number of lines, and writes the partitions back to S3 in a pipe stream. We also created a modified version of the script that compresses the chunked data using the `gzip` command before uploading it to storage, using the fastest

[3]https://github.com/aitorarjona/dataplug-validation

```
aws s3 cp s3://$BUCKET/$KEY - | zcat -
  | split -l $CHUNK_LINES
  --filter='aws s3 cp - s3://$BUCKET/$KEY-parts/$FILE'
```

Listing 5: Bash script to partition a FASTQ file from S3 in streaming, which writes back the output partitions to S3.

FASTQGZip File Size	GZip Index Size	% of original file size
1 GB	$3.1MB$	0.31%
3 GB	$9.7MB$	0.32%
5 GB	$17.6MB$	0.35%
8 GB	$27.4MB$	0.34%
13 GB	$45.0MB$	0.35%

Table I: GZip index size ratio for each FASTQGZip file pre-processed.

compression ratio. Each partition is approximately 850 MB of decompressed size per partition.

For the *Cloud-aware* pre-processing approach with *Dataplug*, we employ the method described in Section IV. Our pre-processing reads the input file from S3 in a stream and generates a GZip index using `gztool`. This index can be then used to query the GZip file to retrieve specific chunks and decompress them *on-the-fly*.

For the data, we used various samples of FASTQ readings from the dataset published in [23], concatenating several samples to generate large files. For both experiments, we used a `t2.xlarge` instance on AWS EC2.

Figure 3 shows the execution times for pre-processing different input FASTQGZip file sizes using both static partitioning and *Cloud-aware* indexing approaches. The results demonstrate that *Cloud-aware* indexing of GZip files is up to $\times 2.9$ faster than generating static partitions.

While both approaches involve reading and decompressing the entire GZip stream, generating static partitions requires an additional step of writing the chunked data back to storage, which increases the pre-processing time. This effect is further amplified if the partitions are compressed before being uploaded to storage. Specifically, for an input compressed file size of 13.8 GB (74.8 GB decompressed), the pre-processing time is reduced by 65.6% when using the GZip index generation approach compared to generating static partitions.

However, employing *Cloud-ware* FASTQGZip implies an extra cost in metadata storage, as we have to account for the storage of the GZip index file. Table I presents the sizes of the GZip indexes for each pre-processed file. We see that the storage cost of the indexes is nearly negligible compared to the original data volume size, as it accounts for only 0.35% of it.

Since the pre-processing time for static partitioning and indexing exhibit a linear relationship with the file size ($r^2 = 0.9999$), we can extrapolate the results for larger volumes. For instance, for 100 TB of FASTQGZip data, static partitioning would occupy $\approx$ 2498 hours of aggregated vCPU time, while indexing would occupy $\approx$ 876 hours. In other words, processing $100TB$ of data with our *Cloud-aware* approach

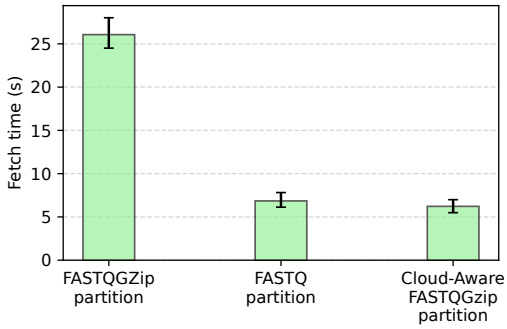


Figure 4: Comparison between fetch time for statically generated compressed and uncompressed FASTQ partitions and *Cloud-aware* FASTQGZip *on-the-fly* partitioning.

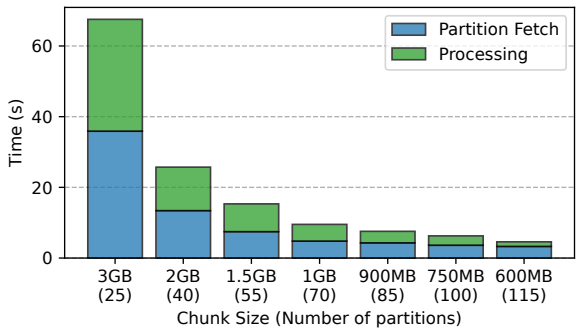


Figure 5: Partitioning and processing runtime for different chunk sizes of a large FASTQGZip file.

would save $\approx$ 1622 hours of compute time for on-demand `t2.xlarge` instance type ($\approx$ 65% less).

2) On-the-fly partitioning overhead: We measure the time taken to fetch a partition of $\approx$ 850 MB using different methods: static FASTQGZip compressed chunks, static FASTQ decompressed chunks, and *on-the-fly* partitioning with *Cloud-aware* FASTQGZip. The partitions are retrieved from serverless functions in AWS Lambda, with a memory configuration of 2048 MB each.

Figure 4 displays the results of the experiment. From it, we can observe that fetching a partition with *Cloud-aware on-the-fly* partitioning and decompressing it significantly lowers the average fetch time (5.81 s) compared to obtaining the static decompressed partition (21.72 s). On the other hand, the fetch time for static compressed partitions is similar in both cases. This difference in fetch time can be attributed to the fact that it is faster to download a smaller compressed payload from object storage and decompress it in memory compared to downloading a larger volume of already decompressed data.

3) Chunk size probing with dynamic partitioning: One of the key benefits of dynamic partitioning is that, after pre-processing only once, a dataset can be re-partitioned at different chunk sizes without any penalty. With this, it becomes trivial to quickly probe many different chunk sizes and to find

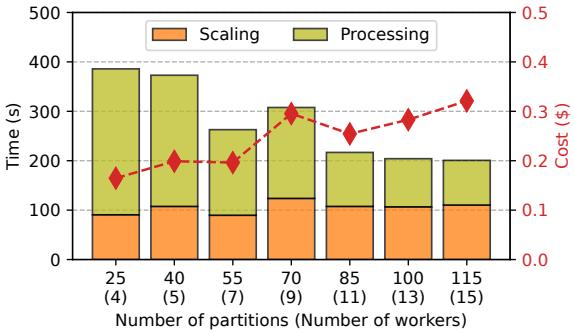


Figure 6: Full workload runtime and cost using Dask for processing a large FASTQGZip file using different number of partitions and workers.

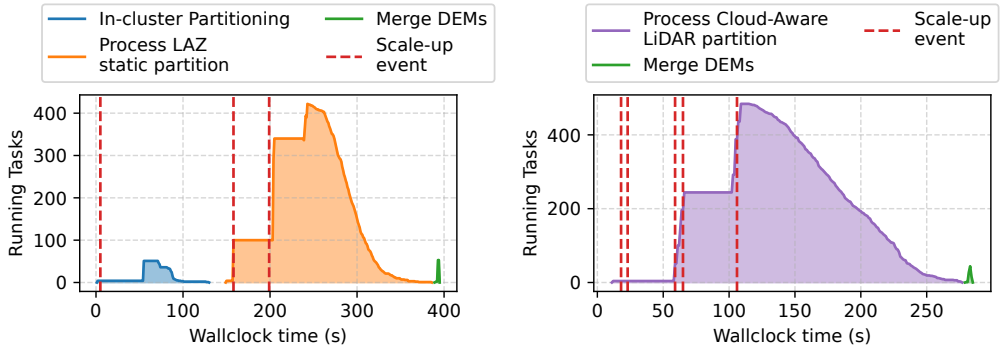
out the most optimal one for a specific workload.

To demonstrate this advantage, we ran a workload consisting of common FASTQ data-parallel processing (filtering using `fastq-filter` [4] and deduplication using `czid-dedup` [5]) for the large 13.8 GB file (74.8 GB decompressed) from the previous experiments. We used Dask with an EC2 cluster and `dask.bag` to distribute the workload in AWS, using `m6i.2xlarge` instance nodes as workers.

For this experiment, we sampled 7 partition sizes ranging from 600 MB to 3 GB (decompressed size) of the 13.8 GB compressed FASTQ file and measured the time elapsed to fetch the partition and process it. The results can be seen in Figure 5. We observe that after 900 MB for chunk size, the performance improvement is no longer substantial. Specifically, it only improves by 3% with respect to the previous largest size, which indicates that from 85 partitions and up, the performance improvement of parallelism is no longer relevant, and using more partitions will only add further overhead. To assess our assumption, we run the full workload with all partition sizes and measure scaling time, processing runtime, and cost. We can see from the results in Figure 6 that, indeed, 900 MB as partition size corresponding to 85 partitions and 11 workers is the configuration that offers the best runtime/cost ratio.

***Main takeaway**: Although both static partitioning and GZip indexing achieve parallel read access for compressed GZip files, the cost of Cloud-aware pre-processing for FASTQGZip is considerably lower compared to static partitioning. Moreover, our evaluation shows that dynamically partitioning and decompressing data on-the-fly is faster than retrieving compressed static partitions for FASTQGZip genomic data, as the compression of partitioned data adds extra computational overhead. Finally, dynamic partitioning of FASTQGZip files allows us to sample different partition sizes with no additional cost, and to probe the performance, which helps in deciding how many resources to deploy and the optimal number of partitions to process a particular dataset.*

[4]https://github.com/LUMC/fastq-filter
[5]https://github.com/chanzuckerberg/czid-dedup



(a) **In-cluster** data partitioning.

(b) ***On-the-fly*** data partitioning.

Figure 7: LiDAR workflow tasks and runtime on a Ray autoscaling cluster, for both data partitioning approaches.

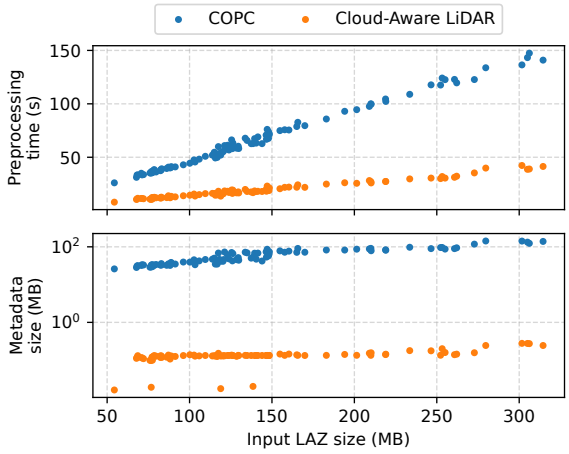


Figure 8: Pre-processing compute time and metadata size overhead comparison for COPC and *Cloud-aware* LiDAR.

B. LiDAR Point Cloud data

1) Pre-processing comparison between Cloud-aware and static partitioing approaches: Our objective is to compare the pre-processing costs of LiDAR data using two different approaches: our *Cloud-aware* pre-processing model based on `LASindex`, and format transformation pre-processing model based on the Cloud-optimized Point Cloud format.

For this experiment, we used a dataset obtained from the United States Geological Survey [6], consisting of 100 different LAZ files, ranging in size from 10 MB to 300 MB, with a total compressed dataset size of 13.8 GB. We utilized a `t2.xlarge` instance type in AWS EC2 to perform the pre-processing jobs.

The results are shown in Figure 8. For both approaches, pre-processing time and metadata size overhead exhibit a linear relationship with the input data file size. However, in comparison, *Dataplug*'s *Cloud-aware* pre-processing approach proves to be faster and requires less metadata storage volume compared to COPC. Specifically, there is a mean difference of 47.63 s in pre-processing time and 59.65 MB in metadata size overhead. In other words, for this dataset, *Dataplug* achieved an average reduction of 71.31% in pre-processing time and 99.78% in metadata volume size.

The reason for these results lies in the nature of Cloud object storage, which does not allow for in-place data modification. Consequently, when transforming a LiDAR dataset to the COPC format, all data has to be copied back to storage after transformation. In contrast, our *Cloud-aware* approach avoids this requirement by only reading the input data and storing metadata separately, resulting in a more efficient approach for Cloud object storage-based workloads.

2) On-the-fly partitioning in auto-scaling clusters: Next, we validate *on-the-fly* partitioning for LiDAR data in a geospatial data-parallel workflow that consists in generating digital terrain elevation models (DEMs), which is a common task in LiDAR data processing [24]. We have implemented the workflow using the PDAL [25] tool to generate DEMs from LiDAR files which are stored in an S3 bucket.

We want to compare the execution time of two workflow runs for the same dataset, one with *in-cluster* partitioning and one with *on-the-fly* partitioning of LiDAR data.

In the *in-cluster* partitioning run, the LiDAR files are read from the S3 bucket in parallel (one task per file), partitioned into equal-sized chunks, and stored within the cluster's temporary storage. *On-the-fly* partitioning avoids performing the partitioning step since partitions can be retrieved directly from Cloud object storage by means of *data slicing*. We consider static partitioning as part of the workload runtime, as the partitioning is specific for this workload. On the contrary, the metadata for *on-the-fly* dynamic partitioning is reusable and can be leveraged in many different workloads with different partition sizes.

To run this experiment, we used Ray [26] with autoscaling enabled on AWS EC2 with `c5.12xlarge` as VM worker nodes (48 vCPUs, 96 GB RAM) and an `m5.xlarge` VM for the head node. We utilized 55 LiDAR files of the dataset mentioned earlier as input, with an average size of 125 MB each.

[6]https://rockyweb.usgs.gov/vdelivery/Datasets/Staged/Elevation/LPC/Projects/CA_YosemiteNP_2019_D19/CA_YosemiteNP_2019/

The total volume of uncompressed data for this experiment was 53.0 GB.

Figure 7a shows the **in-cluster** data partitioning run, and Figure 7b shows the ***on-the-fly*** data partitioning run. We see that *on-the-fly* partitioning of *Cloud-aware* LiDAR provides a clear advantage in runtime compared to the *in-cluster* partitioning approach, as the *in-cluster* partitioning phase is not necessary with *on-the-fly* partitioning. Although Ray takes $\approx 50\,\mathrm{s}$ to deploy and prepare a VM worker to be ready to run tasks, all job tasks are created at the beginning of the workflow, leading to earlier autoscaling events issued by Ray and achieving a sooner and higher degree of parallelism. In total, the *Cloud-aware* LiDAR partitioning approach reduced the workflow runtime by $110.03\,\mathrm{s}$ (27.82% faster).

***Main takeaway**: The adoption of Cloud-aware indexing for LiDAR data offers clear benefits, mainly a significant reduction in pre-processing compute costs and metadata volume. Additionally, on-the-fly partitioning eliminates the need for in-cluster partitioning, resulting in more efficient workloads on auto-scaling clusters.*

VII. Conclusion

Efficient data partitioning is essential in scientific computing to fully benefit from the Cloud's elastic resources, but chunking a large dataset into smaller files is unsuitable in the *extreme data* era. In this article, we propose a *Cloud-aware* data partitioning model, which consists in generating metadata using read-only pre-processing that allows to define dynamic logical partitions of large unstructured datasets. Distributed workers can then retrieve dynamically-sized partitions directly from object storage, exploiting its high bandwidth capability. We implement this data partitioning model with *Dataplug*, an extensible framework with the aim of providing partitioning semantics to many unstructured scientific data formats. We demonstrate, using two representative formats from different domains (genomics and geospatial), that dynamic *on-the-fly* partitioning lowers pre-processing costs without incurring additional overheads. We have open-sourced the framework in order to facilitate the adoption of data-parallel scientific workloads in the Cloud. Future research directions may involve dynamic partitioning and machine learning to determine the optimal partition size based on the workload characteristics and available resources.

Acknowledgment

This work has been partially funded by the European Union through the Horizon Europe NEARDATA (101092644), CLOUDSTARS (101086248), EXTRACT (101093110) projects and by the Spanish Ministry of Economic Affairs and Digital Transformation and the European Union-NextGenerationEU (frameworks PRTR and the MRR), through the CLOUDLESS UNICO I+D CLOUD 2022 project. Aitor Arjona is a URV Martí Franquès grant fellow.

References

[1] A. Arjona, P. García-López, and D. Barcelona-Pons, "Dataplug: Unlocking extreme data analytics with on-the-fly dynamic partitioning of unstructured data," in *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2024, pp. 567–576.

[2] A. Szalay, "Extreme data-intensive scientific computing," *Computing in Science & Engineering*, vol. 13, no. 6, pp. 34–41, 2011.

[3] *Extreme data mining, aggregation and analytics technologies and solutions - EC Funding & tenders*, 2023. [Online]. Available: https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/horizon-cl4-2022-data-01-05

[4] US National Cancer Institute, *The Cancer Genome Atlas Program*, 2023. [Online]. Available: https://www.cancer.gov/ccg/research/genome-sequencing/tcga

[5] A. K. Paul, W. Zhuang, L. Xu, M. Li, M. M. Rafique, and A. R. Butt, "Chopper: Optimizing data partitioning for in-memory data analytics frameworks," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 110–119.

[6] Y. Qi, X. Shen, L. Zhang, and X. Li, "Workload-aware data placement for cloud computing," North Carolina State University. Dept. of Computer Science, Tech. Rep., 2017.

[7] R. P. Abernathey, T. Augspurger, A. Banihirwe, C. C. Blackmon-Luca, T. J. Crone, C. L. Gentemann, J. J. Hamman, N. Henderson, C. Lepore, T. A. McCaie, N. H. Robinson, and R. P. Signell, "Cloud-native repositories for big scientific data," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 26–35, 2021.

[8] Hobu, Inc., *Cloud Optimized Point Cloud Specification – 1.0*, 2022. [Online]. Available: https://copc.io/

[9] *Cloud Optimized GeoTIFF*, 2022. [Online]. Available: https://www.cogeo.org/

[10] Amazon Web Services, *Registry of Open Data on AWS*, 2023. [Online]. Available: https://registry.opendata.aws/

[11] M. A. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[12] *Dask: Library for dynamic task scheduling*, 2022. [Online]. Available: https://dask.org

[13] *Zarr*, 2022. [Online]. Available: https://zarr.readthedocs.io/en/stable/

[14] *kerchunk: Cloud-friendly access to archival data*, 2022. [Online]. Available: https://github.com/fsspec/kerchunk

[15] T. J. Skluzacek, R. Chard, R. Wong, Z. Li, Y. N. Babuji, L. Ward, B. Blaiszik, K. Chard, and I. Foster, "Serverless workflows for indexing large scientific data," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–48. [Online]. Available: https://doi.org/10.1145/3366623.3368140

[16] H. Zhang, "Overview of sequence data formats," in *Statistical Genomics*. Springer, 2016, pp. 3–17.

[17] M. Kerbiriou and R. Chikhi, "Parallel decompression of gzip-compressed files and random access to dna sequences," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019, pp. 209–217. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPSW.2019.00042

[18] circulosmeos, *gztool (Github Repository)*, 2022. [Online]. Available: https://github.com/circulosmeos/gztool

[19] F. D. Maleno Gonzalez, "A compressed file partitioner for scalable genomics analysis with serverless technology," Bachelor Thesis, Universitat Rovira i Virgili (URV), January 2022. [Online]. Available: http://hdl.handle.net/20.500.11797/TFG5628

[20] The American Society for Photogrammetry & Remote Sensing, *LAS Specification 1.4 - R14*, 2022. [Online]. Available: https://www.asprs.org/wp-content/uploads/2019/03/LAS_1_4_r14.pdf

[21] R. S. James W. Hegeman, Vivek B. Sardeshmukh and M. P. Armstrong, "Distributed lidar data processing in a high-memory cloud-computing environment," *Annals of GIS*, vol. 20, no. 4, pp. 255–264, 2014. [Online]. Available: https://doi.org/10.1080/19475683.2014.923046

[22] rapidlasso GmbH, *LASindex – spatial indexing of LiDAR data*, 2012. [Online]. Available: https://rapidlasso.com/2012/12/03/lasindex-spatial-indexing-of-lidar-data/

[23] K. A. Overmyer, E. Shishkova, I. J. Miller, J. Balnis, M. N. Bernstein, T. M. Peters-Clarke, J. G. Meyer, Q. Quan, L. K. Muehlbauer, E. A. Trujillo, Y. He, A. Chopra, H. C. Chieng, A. Tiwari, M. A. Judson, B. Paulson, D. R. Brademan, Y. Zhu, L. R. Serrano, V. Linke, L. A. Drake, A. P. Adam, B. S. Schwartz, H. A. Singer, S. Swanson, D. F. Mosher, R. Stewart, J. J. Coon, and A. Jaitovich, "Large-Scale Multi-omic Analysis of COVID-19 Severity," *Cell Syst*, vol. 12, no. 1, pp. 23–40, Jan 2021.

[24] R. Ma, "Dem generation and building detection from lidar data," *Photogrammetric Engineering & Remote Sensing*, vol. 71, no. 7, pp. 847–854, 2005.

[25] *PDAL - Point Data Abstraction Library*, 2022. [Online]. Available: https://pdal.io/en/latest/

[26] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," *ArXiv*, vol. abs/1712.05889, 2018.