Delving Into the Abyss: A Distributed Decompression System for Indexing Compressed Repositories

Ryan Wong, Tyler Skluzacek (Advisor), Kyle Chard (Advisor)

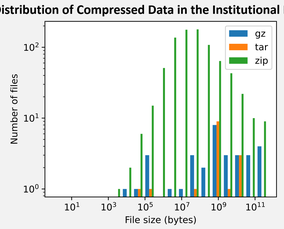
Abstract

- Making scientific data findable, available, interoperable, and reusable (FAIR) requires that data be well described with metadata. Automated methods of metadata extraction are needed when done at scale.
- Large data are often in various compressed formats (e.g. *zip, tar,* and *gzip*) which are difficult to process due to storage constraints and unknown extraction size. The addition of *recursively compressed* data (e.g. *.tar.gz*) further complicates extraction.
- We present Abyss, a system which provides on-demand indexing of compressed data using the funcX federated function-as-a-service (FaaS) platform. funcX allows Abyss to perform decompression across disparate compute systems, providing supplemental resources for these storage and I/O heavy tasks.
- We evaluate Abyss's performance on a 9TB institutional repository.

Background

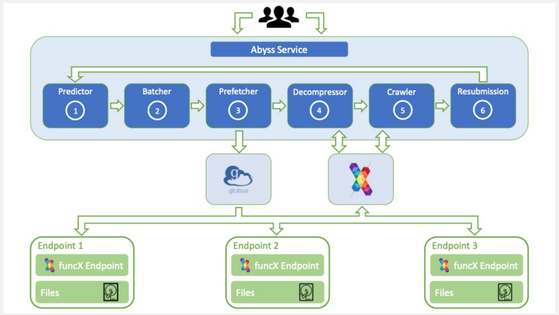
- **Recursively Compressed Data** is data which has been compressed multiple times. While these files may be explicit (e.g. *.tar.gz* files), they sometimes may not be (e.g. a child *.zip* file in a parent *.zip* file), and may require a complex decompression plan.
- Our 9TB institutional repository contains scientific data, teaching data, and large compressed data.

Distribution of Compressed Data in the Institutional Repository



Architecture

Abyss Architecture Diagram



- **Abyss Service:** The Abyss Service is a REST API, which asynchronously handles requests for starting an indexing job, querying job statuses, and fetching job metadata.
- **Endpoints:** Endpoints are the compute units that allow Abyss to process files in a distributed fashion. Endpoints consist of local storage and a compute layer (powered by funcX).
- **Predicting:** To overcome storage limitations during decompression, Abyss employs user-configurable methods to first predict a file's decompressed size before batching.
- **Batching:** Abyss utilizes various user-configurable methods to distribute compressed files to endpoints for decompressing and crawling.
- **Prefetcher:** The prefetcher manages and initiates the transfer between the data source and endpoints using Globus transfer.
- **Recursive Decompression:** After a file has been transferred, Abyss executes a recursive decompression on the file via funcX. The function recursively decompresses data and handles any decompression and out-of-storage errors.
- **Crawling:** After decompressing a file, Abyss executes a crawling function to index the compressed data, collecting basic file metadata such as paths, file sizes, and file extensions.
- **Resubmission and Consolidation:** Once the Abyss service receives a file's metadata, it examines it for recursively compressed files. If Abyss identifies any files, they are resubmitted to Abyss for indexing. Otherwise, Abyss aggregates the metadata and writes it to disk to be pushed to Amazon S3.

Prediction Methods

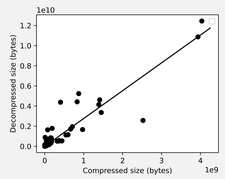
- **File Header:** Abyss transfers files to an endpoint to retrieve the decompressed size from the file's header if it is available.

Bytes of Decompressed Size Located Inside *.zip* File Header

```
b'PK\x03\x04\x14\x00\x00\x00\x08\x0
0\xcadZN2\xcez\xa2"&\x05\x00\xd7&\
x05\x00U\x00\x00\x00junk/image.png
\x9c\xb9g\\\x13O\xf4>\x1aE@AD\x10
P:RT\xba\xd2;
```

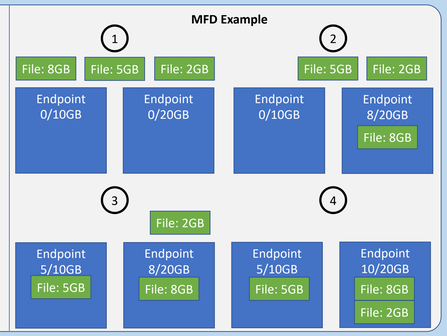
- **Machine Learning:** Abyss utilizes a linear regression to estimate the decompressed size of a file using its compressed size.

***.gzip* Regression Model with Training Data**



Batching Methods

- The **Round Robin** method cycles through endpoints and batches a file to each endpoint.
- The **MFD** (minimize fullness difference) method is a greedy method that minimizes the maximum "fullness" (% space utilization) difference between endpoints.
- The **Knapsack** method utilizes the Knapsack algorithm to determine the subset of files that will utilize the most space on an endpoint.



Experimental Testbed

- We evaluate Abyss's latency as well as Abyss's batching and prediction methods using data from a 9TB institutional repository.
- The Abyss service is deployed on a t2.large AWS EC2 instance with 8GB of RAM and 2 vCPUs, located in the us-east-1 availability zone.
- We utilize Jetstream, a cloud environment, to host our endpoints. Our endpoints are Jetstream m1.xlarge (24 vCPU, 60 GB RAM) instances located in the TACC cluster.

Batching Evaluation

- We evaluate our batching methods by simulating an indexing job over 130GB of data, using real decompression and crawl times from indexing part of the institutional repository, and observing the space utilization and indexing time for each method for various space configurations.
- We consider 3 space configurations: (1) The "constrained" configuration where endpoints have significantly less storage space than required to process all data in one batch (30, 50, 10GB), (2) the "varied" configuration where endpoints have highly heterogeneous space (100, 50, 50 GB), and the "even" configuration where endpoints have uniform storage (50, 50, 50GB).
- MFD performs the best in constrained and varied configurations; Round Robin has a slight performance increase in even configurations; and Knapsack has the lowest space utilization and indexing time of all configurations.

Comparison of Batching Methods in Relation to Round Robin for Various Endpoint Space Configurations

Metric	Configuration	MFD	Knapsack
Indexing Time Speedup	30, 50, 10 GB	+7%	-2%
	100, 50, 50 GB	+8%	-0.3%
	50, 50, 50 GB	-3%	-8%
Average Space Utilization	30, 50, 10 GB	+9%	-2%
	100, 50, 50 GB	+7%	-0.1%
	50, 50, 50 GB	-3%	-8%

Prediction Evaluation

- To efficiently distribute files amongst endpoints, Abyss must first accurately and quickly predict the decompressed size of each file.
- We evaluate the performance of Abyss's prediction methods by simulating Abyss's workflow using real header processing, decompression, and crawling times gathered from indexing data from the institutional repository. Each experiment was run with one simulated endpoint with 200 GB of local storage.
- The Header method is significantly slower than the Machine Learning method, due to additional transfer overhead (from transferring files multiple times) and increased endpoint usage (as endpoints need to allocate cores to header processing in addition to decompressing and crawling).

Indexing Time Speedup of File Header Method in Relation to Machine Learning Method

Compressed Size (GB)	1	2	4	8	16	32	64	130
Header	-186%	-199%	-226%	-255%	-233%	-68%	-88%	-53%

Latency Evaluation

- Abyss's FaaS architecture utilizes multiple components. To understand the average file processing time of each component, we process a synthetic dataset containing 1GB files from the institutional repository on one Jetstream endpoint with 200GB storage space.
- Figure 1 shows that the end-to-end indexing time scales linearly as we increase the compressed size of the dataset from 1-1024GB (~2700GB decompressed).
- Figure 2 shows a breakdown of the average processing time per-file for each component. The total indexing time becomes dominated by predicting and prefetching as we increase the compressed size of the dataset, with scheduling and consolidating remaining relatively negligible.
- Figure 3 is a time series which depicts the number of files being processed by each component at each point in time. Figure 3 confirms that Abyss is primarily bottlenecked by prediction and prefetching.

Figure 1

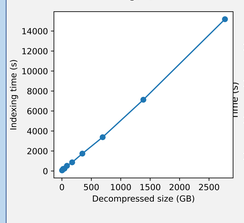


Figure 2

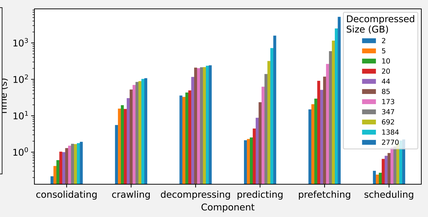
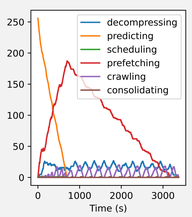


Figure 3



Conclusion

- This project describes Abyss, a FaaS-based system which enables the distributed indexing of recursively compressed data in storage constrained environments.
- We have demonstrated that Abyss is capable of indexing terabytes of synthetic data, indexing recursively compressed data, and can efficiently batch files to operate within constrained, distributed storage environments.
- In future work, we will explore Abyss's scalability and ability to index all of our 9 TB institutional repository. We will additionally extend Abyss to explore dynamically batching files to reduce required data transfers and processing common lossy compression formats.

Acknowledgements

I would like to thank Tyler Skluzacek and Dr. Kyle Chard for advising and mentoring me for this project. I would also like to thank Globus Labs and Argonne National Laboratory for making this research possible.

Abyss Demo Video