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

Ryan Wong
Northwestern University

Tyler J. Skluzacek (Advisor)
University of Chicago

Kyle Chard (Advisor)
University of Chicago

ABSTRACT

Discovery and use of scientific data is dependent on descriptive metadata. Unfortunately, data lakes often contain compressed data, which are difficult to index and automatically extract metadata from due to storage and I/O constraints when inflating and processing recursively compressed data. Here we describe Abyss, a system capable of indexing large amounts of recursively compressed data. Abyss utilizes a function as a service architecture to execute decompression and crawling functions across multiple compute endpoints, allowing Abyss to index data at scale and overcome storage and I/O limitations. Abyss applies methods for predicting a file's decompressed size and batching files to remote endpoints to optimize indexing. We present a prototype implementation of Abyss and demonstrate that it is capable of indexing real world and synthetic compressed data from a 9TB institutional repository.

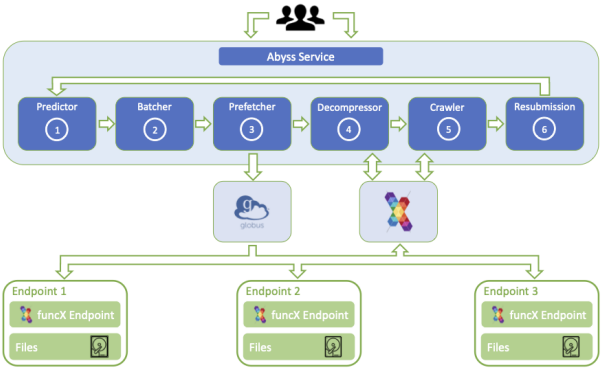

Figure 1: Overview of the Abyss architecture

1 INTRODUCTION

Making scientific data findable, available, interoperable, and reusable (FAIR) requires that data be well described with metadata. At scale, it becomes infeasible for users to manually associate metadata and therefore automated methods are employed to derive metadata [2]. Unfortunately, large data are often compressed in different formats (e.g., *zip*, *tar*, and *gzip*) which can be difficult to process due to storage constraints and unknown extraction size. Further, data are sometimes *recursively compressed* (e.g., *.tar.gz* files).

We present Abyss, a system which provides on-demand indexing of compressed data using the funcX federated function-as-a-service (FaaS) platform [1]. 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 compressed data from a 9 TB institutional repository.

2 ARCHITECTURE

Users interact with Abyss's via REST API, which asynchronously handles requests for indexing repositories. Abyss uses funcX to provision compute resources for execution of decompression and crawling functions. Users specify directories to be processed and target funcX endpoints for processing.

To overcome storage limitations during decompression, Abyss first predicts the decompressed size of a file using user-configurable methods. The File Header method transfers supported formats (*.zip* and *.tar*) to an endpoint to retrieve the decompressed size from the file's header metadata. We additionally developed a machine learning (ML) method to predict sizes from unsupported formats. This method uses linear regression models to estimate a file's decompressed size from its compressed size.

Abyss partitions compressed files into batches and dispatches them to each endpoint, such that each batch's size is less than the endpoint's available space. We explore different methods for allocating batches to endpoints: baseline round robin, MFD (minimize fullness difference), and Knapsack. MFD greedily distributes files evenly across endpoints by minimizing the maximum space utilization difference between all endpoints. Knapsack uses the Knapsack algorithm to determine the subset of files that will maximize the storage used on the endpoint.

Abyss transfers batches to endpoints using Globus transfer, executes a recursive decompression function on the file, and applies a crawling function to determine files for subsequent extraction. Metadata are aggregated and stored in Amazon S3.

3 EVALUATION

We deploy Abyss on a t2.large AWS EC2 instance (8 GB RAM; 2 vCPU), located in us-east-1. We host endpoints on m1.xlarge (24 vCPU, 60 GB RAM) instances on Jetstream [3].

3.1 Latency

To understand the average file processing time for each Abyss component, we process a synthetic dataset containing 1GB files on an endpoint with 200GB disk. Figure 2 shows that end-to-end time scales linearly as we increase compressed size from 1–1024GB (~2700GB decompressed). Figure 3 shows average time spent by a

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

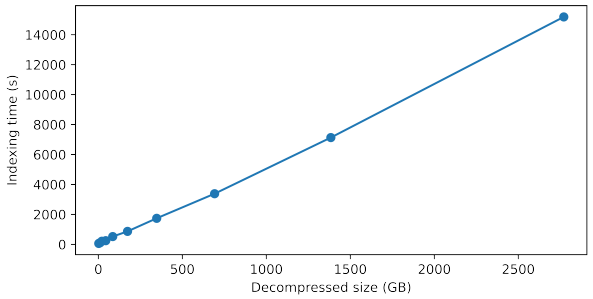

Figure 2: End-to-end indexing times for processing synthetic dataset

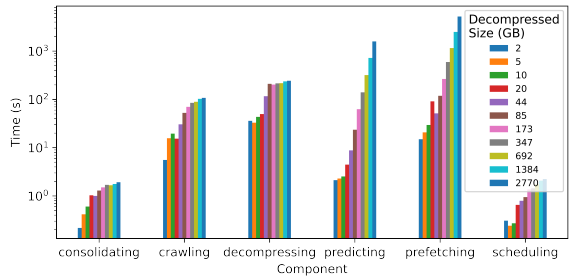

Figure 3: Average processing time per-file for each component when indexing synthetic dataset

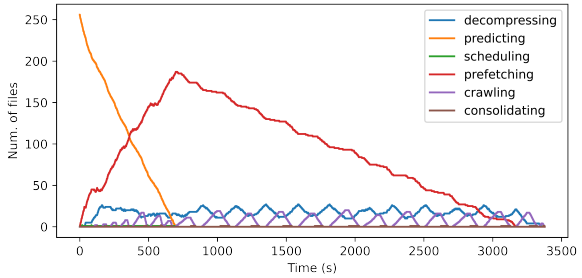

Figure 4: Time series of number of files in each component while processing 267GB of synthetic data

file at each component. We observe that total time is dominated by prediction and prefetching, with scheduling and consolidation being relatively negligible. Figure 4 shows the number of files processed by each component at each point in time. Again we see prediciton and prefetching as bottlenecks.

3.2 Batching

We evaluate the space utilization and time for our batching methods by simulating an indexing job over 130 GB of data, using real decompression and crawl times. We consider 3 space configurations: (1) "constrained" where endpoints have less storage space than required to process all data in one batch (30, 50, 10 GB), (2) "varied" where endpoints have highly heterogeneous space (100, 50, 50 GB) and (3) "even" where endpoints have uniform storage (50, 50, 50 GB).

Table 1 shows that MFD significantly increases performance in the constrained and varied configurations. Round Robin performs best in the even configuration. Knapsack has the slowest indexing time and the lowest average space utilization of all batching methods across all configurations.

Table 1: Comparison of batching methods in relation to Round Robin for various endpoint space configurations

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

3.3 Prediction

We evaluate the performance of the File Header and ML methods by simulating Abyss's workflow using one simulated endpoint with 200 GB storage. Table 2 shows that File Header is substantially slower than ML. This is due to additional transfer overhead from transferring files multiple times as well as increased endpoint usage to process headers.

Table 2: Indexing time speedup of File Header method in relation to Machine Learning method

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

4 CONCLUSION

As data compression is increasingly used to store large volumes of data, scalable methods for indexing compressed data are needed. We have described Abyss, a FaaS-based system which enables the distributed indexing of 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.

REFERENCES

[1] Ryan Chard et al. 2020. funcX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing.*

[2] Tyler J Skluzacek et al. 2020. A Serverless Framework for Distributed Bulk Metadata Extraction. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 7–18.

[3] Craig Stewart, Timothy Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. 2015. Jetstream: A self-provisioned, scalable science and engineering cloud environment. In *XSEDE Conference*. ACM, 29.