THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Data Science Program

Capstone Report - Spring 2024

GRAG - Open Sourced Python Package Implementation of Retrieval-Augmented Generation

Arjun Bingly,
Sanchit Vijay,
Erika Pham,
Kunal Inglunkar

supervised by
Amir Jafari

Abstract

This report introduces GRAG (Good RAG), an open-sourced Python package providing an end-to-end implementation of Retrieval-Augmented Generation (RAG). The package provides easy integration with various LLMs locally, and support for vector databases such as Chroma and DeepLake. It also provides a simple GUI implementation. This report details GRAG and its features. Future work includes enhancement of the PDF parsing features, possible integration for other document types, as well as testing GRAG performance on graphs versus a traditional vector database and producing an evaluation suite. Our documentation can be accessed at https://g-rag.org/ and Git repo at https://github.com/arjbingly/grag.

Contents

1. Introduction

Figure 1 shows a basic Retrieval-Augmented Generation (RAG) pipeline. As the name implies, the process is two-part: retrieval and generation. The input query and documents are first pre-processed into vectors through the embedding process. The pipeline then retrieves data relevant to the query, performing a similarity search in the vector database. Once the retrieval process is complete, RAG utilizes an LLM to understand and preserve context. Then, RAG system integrates the retrieved information with the original query to provide a richer context for the generation phase. In the generation step, the augmented query is processed by a large-language model (LLM), which synthesizes the information into a coherent and contextually appropriate response. The final output is then post-processed, if necessary, to ensure it meets the required specifications, such as correctness, coherence, and relevance.

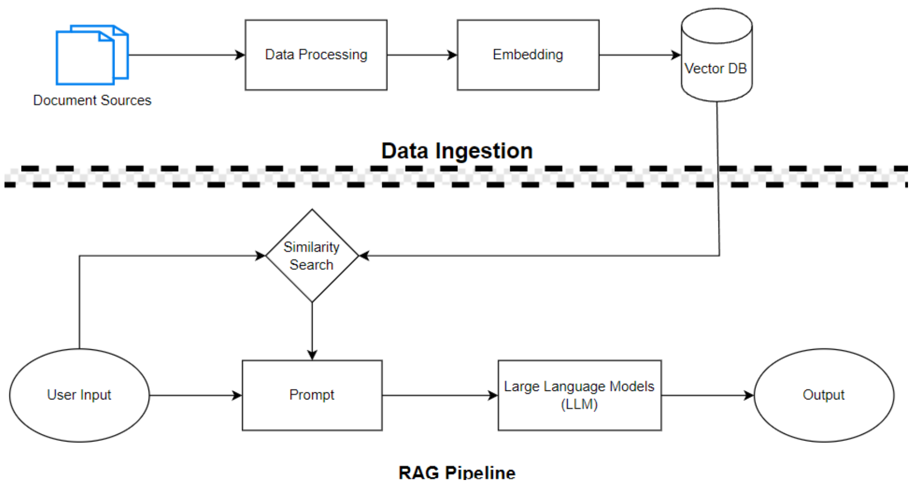


Figure 1: Basic Retrieval-Augmented Generation (RAG) Pipeline

RAG provides several advantages and solutions to LLMs caveats:

- Empowering LLM solutions with real-time data access:

  LLMs are typically trained on vast datasets that may quickly become outdated as new information emerges. RAG technology addresses this limitation by allowing LLMs to access and incorporate real-time data into their responses. Through the retrieval component, RAG systems can query up-to-date databases or the internet to find the most current information, ensuring that the generated output reflects the latest developments.

- Preserving data privacy:

  RAG can retrieve information from a controlled, secure dataset or environment rather than relying on direct access to private data. By designing the retrieval component to operate within privacy-preserving parameters, RAG can ensure that the LLM will not directly access or expose sensitive data.

- Mitigating LLM hallucinations:

  "Hallucination" in the context of LLMs refers to the generation of plausible but inaccurate or entirely fabricated information. This is a known challenge with LLMs, where the model might confidently produce incorrect data or statements.(cite) RAG helps mitigate this issue by grounding the LLM's responses in retrieved documents that are verified or deemed reliable. By leveraging external sources of information, RAG reduces the model's reliance on potentially flawed internal representations and biases, leading to more accurate outputs.

RAG has become very popular since its introduction in Lewis et al. 2020 [1], with its most-known implementation, ChatGPT (powered by GPT-4), creating a sizeable and long-term impact in all industries and academic institutions. Its versatility in several fields has fueled interest in research and development to develop more RAG implementations. Our package, GRAG (Good RAG), aims to add to the RAG literature.

2. Problem Statement

While RAG APIs such as OpenAI are very powerful, they usually have usage limits, which is a barrier for extensive commercial use. Dependence on APIs could limit customization and integration to existing software, which is not ideal for institutions or individuals who have specific needs for their application. Furthermore, sensitive data being stored on external servers, on top of data usage policy dependent on API providers; raise concerns of data privacy.

GRAG aims to provide a customizable, easy-to-use, end-to-end, open-sourced solution that resolves cost and data privacy issues. The package could be implemented locally, allowing for control over data storage and maximizing personalization.

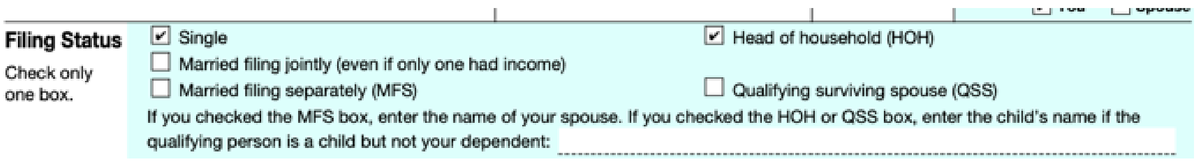
3. Features

3.1. PDF Parser

Parsing PDF documents presents a significant challenge due to their complex structure. PDFs often contain unstructured data, which lacks a predefined organization, making accurate recognition and processing arduous. A notable difficulty arises when handling tables, as PDFs do not inherently understand table columns, complicating the task of recognizing table layouts. This complexity is particularly evident in documents like tax forms, which feature intricate nested table structures. Additionally, scanned PDFs require Optical Character Recognition (OCR) tools to convert images back into text, introducing another layer of complexity.

Initially, we tried only using *unstructured.io*[2] and *pdfplumber*[3], respectively. Neither libraries could consistently parse all PDF files with high accuracy. The current strategy is to primarily use the *unstructured.io* library for partitioning and parsing. For documents containing more complex table structures, such as nested tables or tax forms, *pdfplumber* and *pytesseract*[4] are deployed. The table structures on the documents are detected, then cropped out before the contained text is extracted from the tables.

Below is an example of parsing a tax form, with a complex layout. We see that the output is fairly accurate in this case. The parsing tool is able to identify the layout and extract the information well.



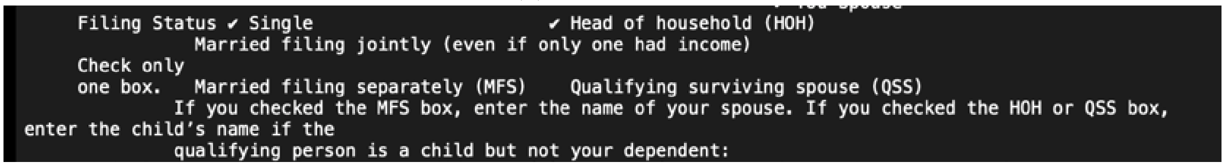
Filing Status

Check only one box.

- [x] Single
- [ ] Married filing jointly (even if only one had income)
- [ ] Married filing separately (MFS)
- [x] Head of household (HOH)
- [ ] Qualifying surviving spouse (QSS)

If you checked the MFS box, enter the name of your spouse. If you checked the HOH or QSS box, enter the child's name if the qualifying person is a child but not your dependent:

(a) Original Text



```
     Filing Status ✔ Single                    ✔ Head of household (HOH)
                Married filing jointly (even if only one had income)
     Check only
     one box.   Married filing separately (MFS)    Qualifying surviving spouse (QSS)
               If you checked the MFS box, enter the name of your spouse. If you checked the HOH or QSS box,
enter the child's name if the
               qualifying person is a child but not your dependent:
```

(b) Parsed Text

Figure 2: Tax Form Parsing Example

This result is not consistent across all types of PDFs, however. Figure 6 (Appendix) is an example of how the tool extracts and parses a two-column paper. The layout is not accurate, and the text position is not in the right place. More experimentation is needed for a robust, consistent parsing tool for PDFs.

3.2. Vector Stores

Vector store or vector database is a type of database that stores data in high-dimensional vectors. This is a crucial component of RAG, storing embeddings for both retrieval and generation processes. Currently, GRAG supports Chroma[5] and DeepLake[6]. By default, our embedding model is instructor-xl, but any HuggingFace embeddings can be used.

3.3. LLMs

As explained above, cost and data privacy concerns mean we could not use OpenAI APIs. To run models locally, *llama.cpp* is the best implementation as it uses a low-level language, with extensive backend support, such as CUDA. Currently, GRAG provides two options to run LLMs locally:

1. Run LLMs using HuggingFace[7]:
This is the easiest way to get started, but does not offer as much flexibility. If using a config file (*config.ini*), simply need to change the *model_name* to the HuggingFace repo id. If the models are gated, user would need to provide an authentication token.

2. Run LLMs using llama.cpp [8]:
Llama.cpp offers great flexibility, but its repository is not user-friendly. To streamline the process, we have a script/cookbook for ease of implementation.

GRAG has been tested with Llama2 7b & 13b, Mixtral 8x7b, Gemma 13b, but could support any model from HuggingFace.

3.4. Multi-vector Retriever

We built a retriever, which employs a Langchain tool called *multi-vector retriever.* Instead of representing each document as one vector, one document is represented by multiple vectors, with each vector being a different aspect of the text. For example, the tool could split and then embed smaller chunks of a document, which would help embeddings to better carry semantic context. Our retriever is used to retrieve multiple chunks and return the most similar chunks from a vectorDB.

3.5. RAG Features

3.5.1. RAG Document Chains

Document chains are used in Retrieval-Augmented Generation (RAG) to effectively utilize retrieved documents. These chains serve various purposes, including efficient document processing, task decomposition, and improved accuracy.

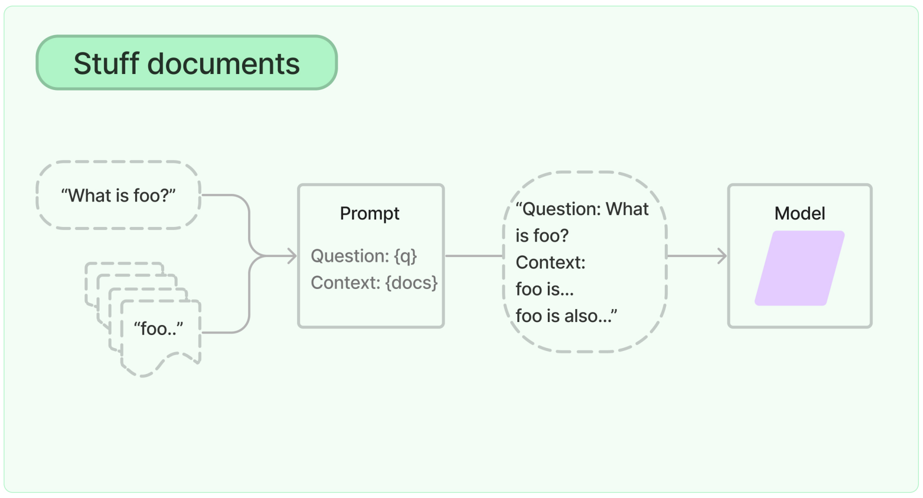


Figure 3: Illustration of Stuff Chain (LangChain)

Stuff Chain

This is the simplest form of document chain. It involves putting all relevant data into the prompt. Given n documents, it concatenates the documents with a separator, usually `\n\n`. The advantage of this method is *it only requires one call to the LLM*, and the model has access to all the information at once.

However, most LLMs can only handle a certain amount of context. For large or multiple documents, stuffing may result in a prompt that exceeds the context limit. Additionally, this method is only suitable for smaller amounts of data. When working with larger data, alternative approaches should be used.

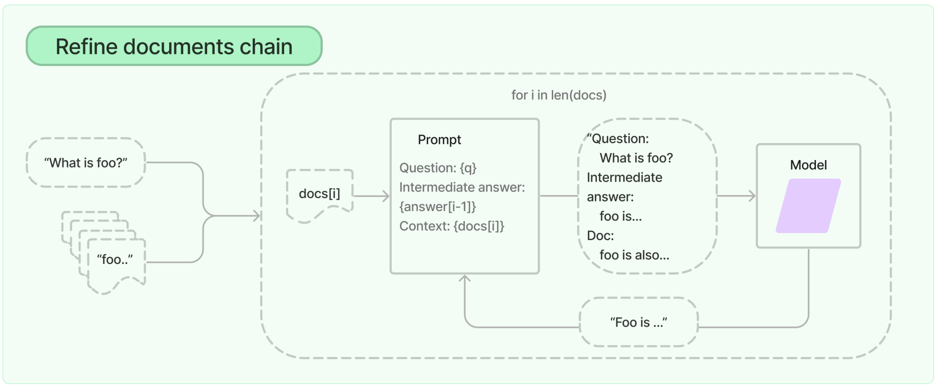


Figure 4: Illustration of the Refine Chain method (LangChain)

Refine Chain

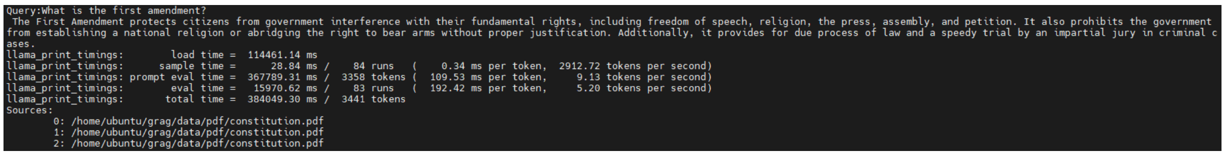
The Refine Documents Chain uses an iterative process to generate a response by analyzing each input document and updating its answer accordingly. It passes all non-document inputs, the current document, and the latest intermediate answer to an LLM chain to obtain a new answer for each document.

This chain is ideal for tasks that involve analyzing more documents than can fit in the model's context, as it only passes one document to the LLM at a time.
However, this also means it makes significantly more LLM calls than other chains, such as stuff chain. It may perform poorly for tasks that require cross-referencing between documents or detailed information from multiple documents.

There may also be dependencies on the order in which the documents are analyzed, thus it might be ideal to provide documents in order of similarity.

3.5.2. Prompting

In additional to model-specific prompting, GRAG provides cookbooks to implement custom generic and few-shot prompts. To use, user needs to download and quantize a model, and ingest their document(s) of choice. As an example, we have ingested the United States Constitution to demonstrate custom prompting. Figure 5 shows an example of a custom prompt output.



```
Query:What is the first amendment?
 The First Amendment protects citizens from government interference with their fundamental rights, including freedom of speech, religion, the press, assembly, and petition. It also prohibits the government
from establishing a national religion or abridging the right to bear arms without proper justification. Additionally, it provides for due process of law and a speedy trial by an impartial jury in criminal c
ases.
llama_print_timings:        load time =  114461.14 ms
llama_print_timings:      sample time =      28.84 ms /    84 runs   (    0.34 ms per token,  2912.72 tokens per second)
llama_print_timings: prompt eval time =  367789.31 ms /  3358 tokens (  109.53 ms per token,     9.13 tokens per second)
llama_print_timings:        eval time =   15970.62 ms /    83 runs   (  192.42 ms per token,     5.20 tokens per second)
llama_print_timings:       total time =  384049.30 ms /  3441 tokens
Sources:
        0: /home/ubuntu/grag/data/pdf/constitution.pdf
        1: /home/ubuntu/grag/data/pdf/constitution.pdf
        2: /home/ubuntu/grag/data/pdf/constitution.pdf
```

Figure 5: Custom Prompt Cookbook Example

"Query: What is the first amendment?" shows the input question.

The generated text is: "The First Amendment protects citizens from government interference with their fundamental rights, including freedom of speech, religion, the press, assembly, and petition. It also prohibits the government from establishing a national religion or abridging the right to bear arms without proper justification. Additionally, it provides for due process of law and a speedy trial by an impartial jury in criminal cases."

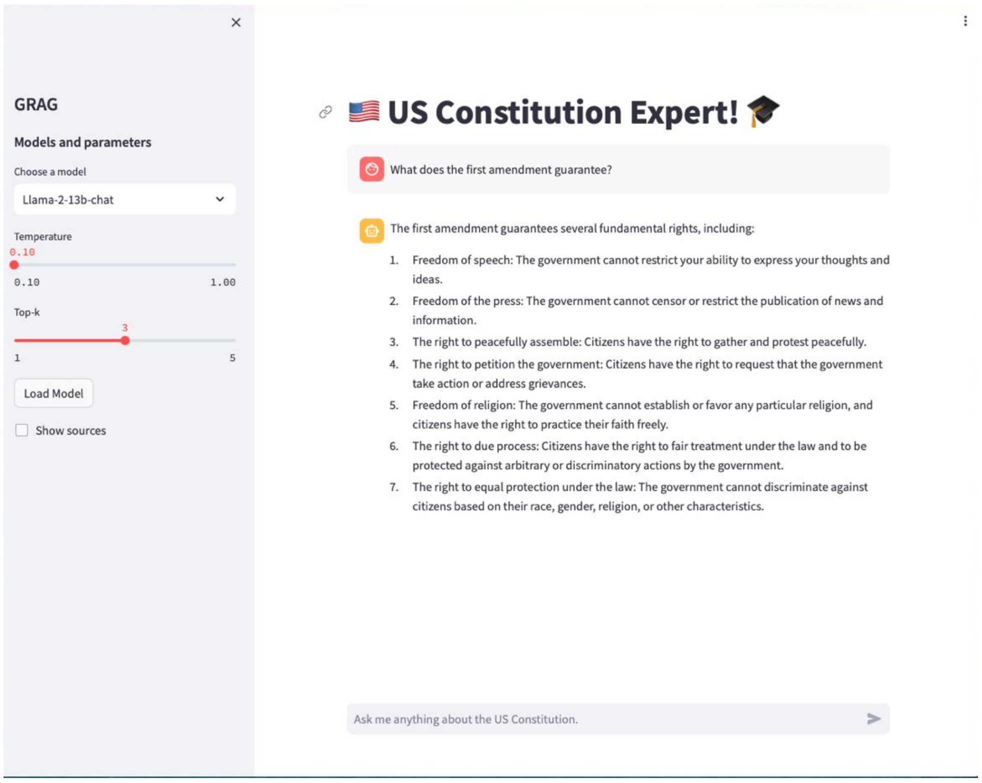
The module output also shows three "Sources" of where the pipeline got the generated information from.

3.5.3. Other Hyperparameters

- **Chunk Sizes** — generally, the smallest chunk size you can get away with.
- **Similarity Score** — e.g., cosine similarity, a measure used to determine how similar two documents or vectors are.
- **Embedding** — a representation of text in a high-dimensional vector space, which allows for capturing the semantic meaning of words or phrases.

3.6. GUI

GRAG provides a RAG GUI cookbook that you can run using Streamlit. To run, you need to download and quantize the LLM models, and ingest the documents you would like to use. As an example, we loaded the United States Constitution as a demo."Temperature" toggle decides how "creative" the generated answer is, and "Top-k" toggle decides how many relevant chunks to retrieve.



Choose a model: Llama-2-13b-chat

Temperature: 0.10 (0.10 – 1.00)

Top-k: 3 (1 – 5)

Show sources: ☐

US Constitution Expert!

What does the first amendment guarantee?

The first amendment guarantees several fundamental rights, including:

1. Freedom of speech: The government cannot restrict your ability to express your thoughts and ideas.
2. Freedom of the press: The government cannot censor or restrict the publication of news and information.
3. The right to peacefully assemble: Citizens have the right to gather and protest peacefully.
4. The right to petition the government: Citizens have the right to request that the government take action or address grievances.
5. Freedom of religion: The government cannot establish or favor any particular religion, and citizens have the right to practice their faith freely.
6. The right to due process: Citizens have the right to fair treatment under the law and to be protected against arbitrary or discriminatory actions by the government.
7. The right to equal protection under the law: The government cannot discriminate against citizens based on their race, gender, religion, or other characteristics.

Figure 6: U.S. Constitution GUI Demo

4. Challenges & Future Work

4.1. Evaluation

When evaluating a RAG pipeline, we assess the performance of both the retrieval and generation processes. For retrieval, we measure how relevant the retrieved information is using precision/recall metrics. For generation, we examine the quality of the generated answer based on the retrieved content.

For generation evaluation, traditional metrics such as ROUGE and BLEU calculate performance based on number of n-gram overlaps between generated text and a reference text. They are particularly limited; they do not correlate well with human evaluation [9]. Recently, LLMs have been proposed as a solution for reference-free end-to-end evaluation of question-answer pipelines like RAG. These models can be employed to predict the likelihood of a generated answer being correct based on context understanding rather than reference texts.

Currently, the most promising framework is Ragas [10]. However, it requires OpenAI to assess the performance. This is rather problematic for our project, since we have limited resources.

4.2. Document Parsing

While we have made improvements in parsing PDF files, the results are not consistent for different PDF layouts. More experimentation is needed to produce a more robust and accurate result. We also plan to implement parsing for more file types, such as HTML.

4.3. Graphs Implementation

Traditionally, RAG implementations uses vector databases for its retrieval process. As RAG uses vector embeddings for its processes, vector database is the optimal choice for ease of retrieval and efficiency in similarity search. However, since RAG simply outputs the closest vector by means of cosine similarity, it leaves room for error if the database does not contain relevant information to the input prompt. This makes RAG overly reliant on the quality of the data and the embedding process.

Graphs present a very promising possibility due to its complex relational network - in theory, this could solve vector DB's limitation and reliance on cosine similarity. There is limited existing literature comparing the performance of vector versus graph structure in a RAG implementation. Our next step is to implement graph database for GRAG, and compare its performance to vector databases.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[2] Unstructured, "unstructured.io," https://unstructured.io/, 2024, accessed: Jan 20, 2024.

[3] J. Singer-Vine, "pdfplumber," https://github.com/jsvine/pdfplumber, 2024, accessed: Jan 20, 2024.

[4] S. Hoffstaetter, "pytesseract: Python bindings for google tesseract ocr," https://github.com/h/pytesseract, 2018, accessed: Jan 20, 2024.

[5] Chroma, "Chroma," https://docs.trychroma.com/, 2023, accessed: Jan 20, 2024.

[6] deeplake, "deeplake," https://docs.deeplake.ai/en/latest/deeplake.html, 2022, accessed: Jan 20, 2024.

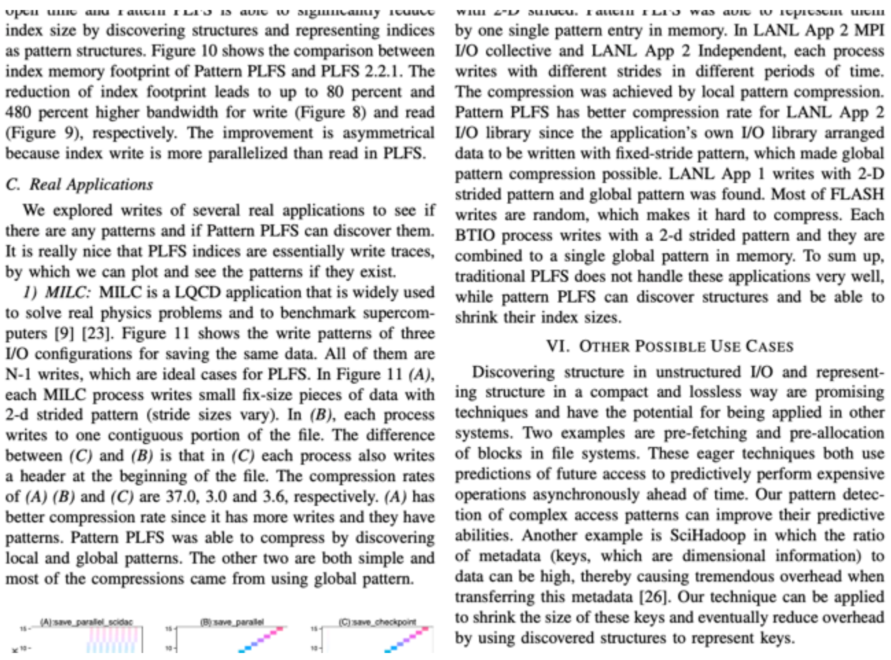
[7] HuggingFace, "Huggingface," https://huggingface.co/, 2024, accessed: Jan 20, 2024.

[8] LangChain, "Llamacpp integration," https://python.langchain.com/v0.1/docs/integrations/llms/llamacpp/, 2023, accessed: Jan 20, 2024.

[9] A. B. Sai, A. K. Mohankumar, and M. M. Khapra, "A survey of evaluation metrics used for nlg systems," 2020.

[10] S. Es, J. James, L. Espinosa-Anke, and S. Schockaert, "Ragas: Automated evaluation of retrieval augmented generation," 2023.

A. Appendix



open time and Pattern PLFS is able to significantly reduce index size by discovering structures and representing indices as pattern structures. Figure 10 shows the comparison between index memory footprint of Pattern PLFS and PLFS 2.2.1. The reduction of index footprint leads to up to 80 percent and 480 percent higher bandwidth for write (Figure 8) and read (Figure 9), respectively. The improvement is asymmetrical because index write is more parallelized than read in PLFS.

C. Real Applications

We explored writes of several real applications to see if there are any patterns and if Pattern PLFS can discover them. It is really nice that PLFS indices are essentially write traces, by which we can plot and see the patterns if they exist.

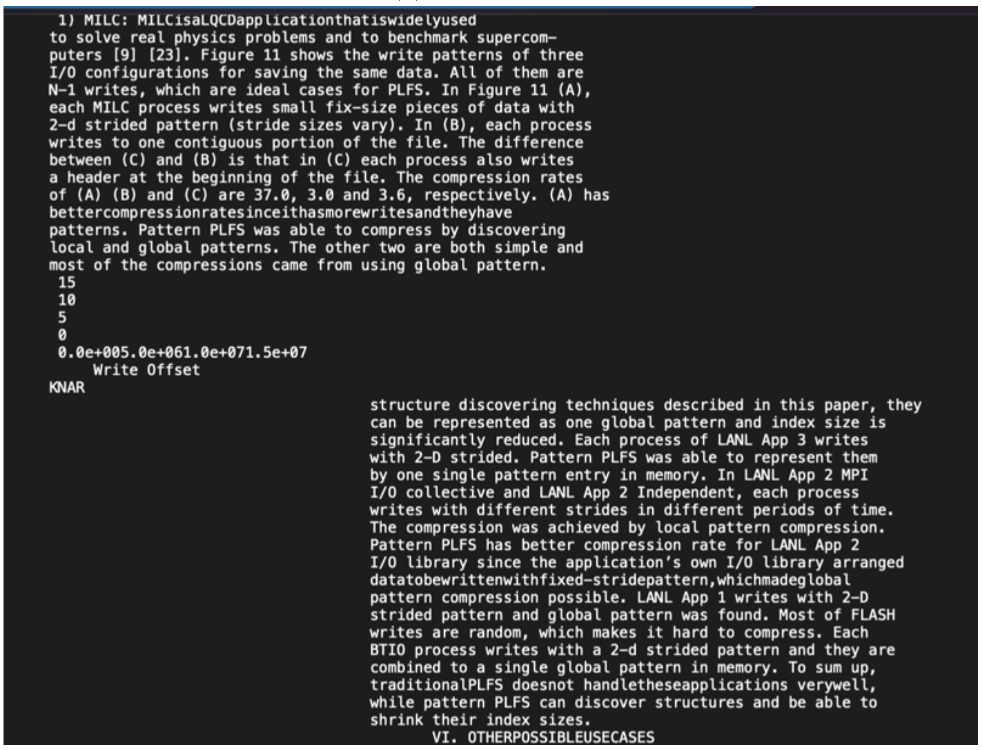
1) MILC: MILC is a LQCD application that is widely used to solve real physics problems and to benchmark supercomputers [9] [23]. Figure 11 shows the write patterns of three I/O configurations for saving the same data. All of them are N-1 writes, which are ideal cases for PLFS. In Figure 11 *(A)*, each MILC process writes small fix-size pieces of data with 2-d strided pattern (stride sizes vary). In *(B)*, each process writes to one contiguous portion of the file. The difference between *(C)* and *(B)* is that in *(C)* each process also writes a header at the beginning of the file. The compression rates of *(A) (B)* and *(C)* are 37.0, 3.0 and 3.6, respectively. *(A)* has better compression rate since it has more writes and they have patterns. Pattern PLFS was able to compress by discovering local and global patterns. The other two are both simple and most of the compressions came from using global pattern.

with 2-D strided. Pattern PLFS was able to represent them by one single pattern entry in memory. In LANL App 2 MPI I/O collective and LANL App 2 Independent, each process writes with different strides in different periods of time. The compression was achieved by local pattern compression. Pattern PLFS has better compression rate for LANL App 2 I/O library since the application's own I/O library arranged data to be written with fixed-stride pattern, which made global pattern compression possible. LANL App 1 writes with 2-D strided pattern and global pattern was found. Most of FLASH writes are random, which makes it hard to compress. Each BTIO process writes with a 2-d strided pattern and they are combined to a single global pattern in memory. To sum up, traditional PLFS does not handle these applications very well, while pattern PLFS can discover structures and be able to shrink their index sizes.

VI. Other Possible Use Cases

Discovering structure in unstructured I/O and representing structure in a compact and lossless way are promising techniques and have the potential for being applied in other systems. Two examples are pre-fetching and pre-allocation of blocks in file systems. These eager techniques both use predictions of future access to predictively perform expensive operations asynchronously ahead of time. Our pattern detection of complex access patterns can improve their predictive abilities. Another example is SciHadoop in which the ratio of metadata (keys, which are dimensional information) to data can be high, thereby causing tremendous overhead when transferring this metadata [26]. Our technique can be applied to shrink the size of these keys and eventually reduce overhead by using discovered structures to represent keys.

(a) Original Text



```
 1) MILC: MILCisaLQCDapplicationthatiswidelyused
to solve real physics problems and to benchmark supercom-
puters [9] [23]. Figure 11 shows the write patterns of three
I/O configurations for saving the same data. All of them are
N-1 writes, which are ideal cases for PLFS. In Figure 11 (A),
each MILC process writes small fix-size pieces of data with
2-d strided pattern (stride sizes vary). In (B), each process
writes to one contiguous portion of the file. The difference
between (C) and (B) is that in (C) each process also writes
a header at the beginning of the file. The compression rates
of (A) (B) and (C) are 37.0, 3.0 and 3.6, respectively. (A) has
bettercompressionratesinceithasmorewritesandtheyhave
patterns. Pattern PLFS was able to compress by discovering
local and global patterns. The other two are both simple and
most of the compressions came from using global pattern.
 15
 10
 5
 0
 0.0e+005.0e+061.0e+071.5e+07
     Write Offset
KNAR
                                    structure discovering techniques described in this paper, they
                                    can be represented as one global pattern and index size is
                                    significantly reduced. Each process of LANL App 3 writes
                                    with 2-D strided. Pattern PLFS was able to represent them
                                    by one single pattern entry in memory. In LANL App 2 MPI
                                    I/O collective and LANL App 2 Independent, each process
                                    writes with different strides in different periods of time.
                                    The compression was achieved by local pattern compression.
                                    Pattern PLFS has better compression rate for LANL App 2
                                    I/O library since the application's own I/O library arranged
                                    datatobewrittenwithfixed-stridepattern,whichmadeglobal
                                    pattern compression possible. LANL App 1 writes with 2-D
                                    strided pattern and global pattern was found. Most of FLASH
                                    writes are random, which makes it hard to compress. Each
                                    BTIO process writes with a 2-d strided pattern and they are
                                    combined to a single global pattern in memory. To sum up,
                                    traditionalPLFS doesnot handletheseapplications verywell,
                                    while pattern PLFS can discover structures and be able to
                                    shrink their index sizes.
                                           VI. OTHERPOSSIBLEUSECASES
```

(b) Parsed Text

Figure 7: Double-column Parsing Example