Hermes: Algorithm-System Co-design for Efficient Retrieval-Augmented Generation At Scale

Michael Shen
Cornell University
New York, New York, USA
mts247@cornell.edu

Muhammad Umar
Cornell University
Ithaca, New York, USA
mu94@cornell.edu

Kiwan Maeng
Pennsylvania State University
University Park, Pennsylvania, USA
kvm6242@psu.edu

G. Edward Suh
NVIDIA, Cornell University
Ithaca, New York, USA
esuh@nvidia.com

Udit Gupta
Cornell University
New York, New York, USA
ugupta@cornell.edu

Abstract

The rapid advancement of Large Language Models (LLMs) as well as the constantly expanding amount of data make keeping the latest models constantly up-to-date a challenge. The high computational cost required to constantly retrain models to handle evolving data has led to the development of Retrieval-Augmented Generation (RAG). RAG presents a promising solution that enables LLMs to access and incorporate real-time information from external datastores, thus minimizing the need for retraining to update the information available to an LLM. However, as the RAG datastores used to augment information expand into the range of trillions of tokens, retrieval overheads become significant, impacting latency, throughput, and energy efficiency. To address this, we propose Hermes, an algorithm-systems co-design framework that addresses the unique bottlenecks of large-scale RAG systems. Hermes mitigates retrieval latency by partitioning and distributing datastores across multiple nodes, while also enhancing throughput and energy efficiency through an intelligent hierarchical search that dynamically directs queries to optimized subsets of the datastore. On open-source RAG datastores and models, we demonstrate Hermes optimizes end-to-end latency and energy by up to 9.33× and 2.10×, without sacrificing retrieval quality for at-scale trillion token retrieval datastores.

CCS Concepts

• **Computer systems organization** → **Architectures**; • **Computing methodologies** → **Natural language processing**; • **Information systems** → **Information retrieval**.

Keywords

Retrieval-Augmented Generation, Machine Learning Systems, Vector Search, Large Language Models, Retrieval and Ranking Models, k-Nearest Neighbor (kNN) Search

ISCA '25, June 21–25, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731076

ACM Reference Format:
Michael Shen, Muhammad Umar, Kiwan Maeng, G. Edward Suh, and Udit Gupta. 2025. Hermes: Algorithm-System Co-design for Efficient Retrieval-Augmented Generation At Scale. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3695053.3731076

1 Introduction

The rapid rise in the use of Large Language Models (LLMs) across everyday, industrial, and academic domains has fueled an explosion of research in the field. Resultingly, over the last few years, state-of-the-art LLMs have grown exponentially to hundreds of billions of parameters, realizing significant accuracy and quality of service improvements [4, 19]. In addition to growing model sizes, the number of consumers and enterprise customers using LLMs continues to grow. The tremendous growth in model capacity and deployment scale poses new challenges for LLM's at-scale. First, the computational demands for retraining LLMs with new and continually evolving data have become increasingly daunting [19]. Second, LLMs have been shown to suffer from hallucinations, producing incoherent or inconsistent outputs [30].

One promising solution to addressing these challenges is Retrieval-Augmented Generation (RAG) [24]. Intuitively, RAG-based LLMs incorporate real-time information from mutable external databases. Encoded input queries from users are used as key vectors to index into and search large vector databases for relevant context via information retrieval or similarity search processes; the relevant context and original input query are then provided as input to LLMs. The use of contextual information from vector databases allows RAG-based LLMs to (1) produce relevant, current output without needing frequent re-training, (2) ground generated outputs to reduce hallucinations (as seen in Figure 1). Relevant contexts are repeatedly retrieved during the generation process to continuously improve the quality of the generated outputs, a technique known as retrieval striding.

While RAG-based LLMs reduce training requirements for modern LLMs and help mitigate hallucinations, they also introduce new challenges to efficient deployment at-scale. Primarily, as the size of knowledge datastores used to augment information continues to grow, the overhead involved in conducting information retrieval or similarity search becomes a major bottleneck. Similarity search on large, TB-scale datastores stress memory systems, resulting

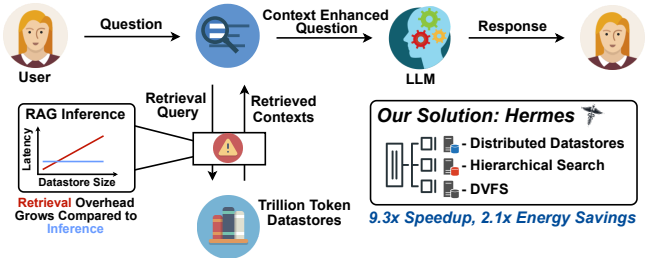

Figure 1: Retrieval-Augmented Generation enables dynamic LLM updates but incurs high latency with at-scale trillion-token datastores. Hermes addresses this with a distributed, hierarchical search framework.

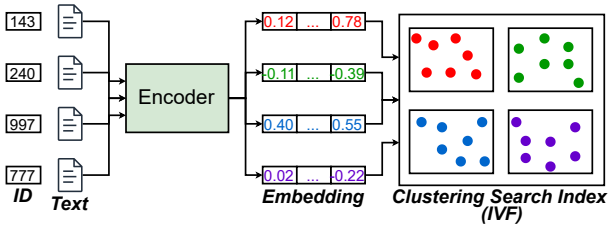

Figure 2: RAG datastores encode document chunks into embeddings and are used in conjunction with the search index for efficient retrieval. Retrieved IDs are used to access corresponding chunks.

in high latency and energy overheads as well as low throughput. Additionally, designing an efficient retrieval index further complicates the process, making seamless integration into real-time RAG applications challenging.

Given the overheads of RAG, systems researchers have begun exploring methods for optimizing RAG-based LLMs [1, 15–17, 49]. For example, researchers have investigated methods to pipeline and overlap retrieval and LLM inference stages [16] and cache frequently accessed documents and LLM prompt computations [17]. While these optimizations are effective for RAG-based LLMs with modestly sized datastores (i.e., only up to a few hundred billion tokens), they overlook the unique challenges of deploying RAG with large datastores (often trillions of tokens in size [3, 41]). For instance, a recently released retrieval datastore dedicated to RAG comprises of 1.4 trillion tokens [41], an order of magnitude larger than studied in prior work. At this scale, we find the retrieval stage of RAG once again introduces significant overheads, not only in terms of latency but also throughput and energy efficiency.

In this paper, we introduce Hermes (as seen in Figure 1), an algorithm-systems co-design framework that addresses the key bottlenecks encountered in large-scale RAG systems. We begin by conducting a detailed characterization of the operational challenges posed by large-scale datastores. Our characterization highlights critical aspects of the retrieval process, with a focus on memory usage, latency, throughput, and energy bottlenecks. We then propose Hermes, which reduces the latency, and memory requirements of the retrieval stage by splitting and distributing the large datastore across several machines. However, naive distribution of the datastore requires aggregation of the retrieval across all machines. Critically, Hermes also improves the throughput and energy efficiency of the system by selectively routing each query to a subset of machines by ranking machines via a fast, limited-scale sampling search, all while retaining the accuracy of retrieval. Using this approach, Hermes achieves improvement in inference time-to-first token (TTFT), end-to-end latency, throughput, and energy. The main contributions of our work are as follows:

- We investigate RAG systems from a large-scale deployment perspective, uncovering significant challenges in trillion-token datastores, where RAG experiences prohibitively long TTFT and overall end-to-end latencies, even with enhancements proposed by prior research.
- We propose Hermes, an algorithm-systems co-design framework that reduces retrieval overhead by partitioning and distributing datastores across multiple CPU nodes, enabling more efficient and methodical search strategies. By employing an intelligent hierarchical search that dynamically directs queries to optimized subsets of the datastore, Hermes significantly enhances throughput and energy efficiency, shifting the critical path for RAG from CPU-based retrieval to GPU-based inference.
- We highlight the performance advantages Hermes offers over previous solutions and demonstrate how our enhancements continue to deliver improvements even with massive trillion-token datastores, where earlier proposed methods have struggled. We demonstrate that Hermes can achieve up to 9.33× speedup in latency, 9.29× throughput, and 2.10× energy efficiency improvements over a monolithic large datastore for context retrieval.

We have open-sourced the Hermes infrastructure, including index construction tools, accuracy evaluation scripts, and online serving optimizations. This release aims to support and accelerate follow-on research on efficient RAG with large-scale datastores. The live GitHub repository is available at: https://github.com/S4AI-CornellTech/Hermes.

2 Background: Retrieval-Augmented Generation

While traditional pretrained language models process queries using only the information stored within their fixed parameters, RAG pipelines differ by incorporating a dynamic, non-parametric datastore. Instead of relying solely on learned knowledge, RAG models query this datastore for relevant external contexts, which are then augmented with the original query before being processed by the model. This retrieval step allows RAG models to generate more informed and contextually relevant responses. This non-parametric datastore is incorporated into the LLM through 2 distinct stages:

- **Offline Search Index Creation**: where the dataset is processed for relevant information, transformed into a format more suitable for the search paradigm (i.e. vector embeddings for dense searches in our case), and organized into a search index to enable fast and efficient retrieval of relevant chunks during inference.
- **Online Inference**: where the model interacts with the search index in real-time to retrieve and integrate relevant data chunks, augmenting the query with the external knowledge from the datastore to improve accuracy.

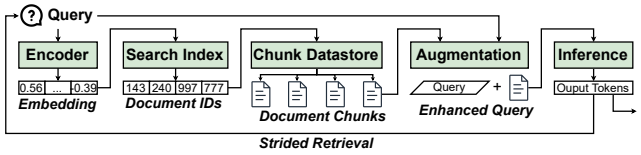

Figure 3: Online RAG inference encodes the query to retrieve document IDs, maps them to document text chunks, and appends them to the query. Every s tokens, the query is updated with generated output, repeating until completion.

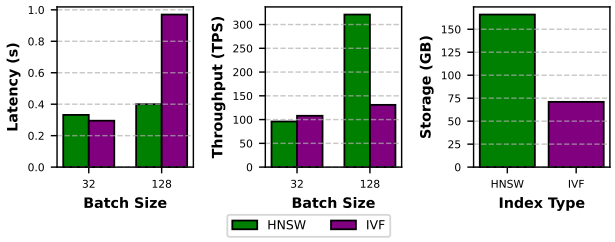

Figure 4: Comparison of systems metrics for HNSW and IVF 10B token indices built on 100M document subsets of Common Crawl [36]. HNSW indices achieve more than 2.4x latency (0.40s vs 0.97s) and throughput (321 QPS vs 131 QPS) improvements at a batch size of 128, but have a 2.3x higher memory requirement (166GB vs 71GB)

2.1 Index Creation

For efficient information retrieval during online inference, non-parametric datastores must be thoughtfully structured to reduce the search space without compromising accuracy. This goal is typically achieved by creating streamlined yet powerful vector search indices. This is done by first partitioning the datastore documents into manageable chunks (groups of tokens with specified lengths to better facilitate retrieval) and then encoding them. Afterwords, a framework such as FAISS [6] is typically used to organize the embeddings for efficient similarity searches. The full index creation workflow can be seen in Figure 2.

While prior research has explored *sparse* text-based search indices, *dense* vector-based search indices have gained prominence among researchers due to their ability to more effectively identify semantic similarity between a query and document vectors [40]. Sparse retrieval relies on traditional term-based methods and exact matching, which make them better suited for handling rare terms that cannot be adequately represented through embeddings. Sparse retrieval is limited in application, however, due to their inability to effectively capture contextual relationships between words the way embeddings can. Although research has explored combining sparse and dense retrieval methods [40], we focus our efforts on studying and refining dense vector indices, as they are more effective for RAG applications.

The task of retrieving document chunks relevant to a query in dense vector indices is a vector nearest-neighbor search problem. While we can naively use a brute-force search to return relevant chunks, it is more efficient to use an approximate nearest-neighbor (ANN) algorithm to reduce the search space of the vector search. A wide variety of such algorithms exist, that span graphs, trees, hashing, and clustering-based algorithms [34]. In recent works

Table 1: Comparison of different IVF quantization schemes. We choose SQ8 as the point that best optimizes memory without adversely hurting recall.

	Recall	Vector Size (Bytes)
Flat	0.958	3072
SQ8	**0.942**	**768**
SQ4	0.748	384
PQ256	0.585	256
OPQ256	0.596	256
PQ384	0.748	384
OPQ384	0.742	384

[16, 17, 31], however, it has been demonstrated that in the high-dimensional LLM embedding regime, two types of vector indices exhibit good performance in terms of accuracy and throughput: Hierarchical Navigable Small World (HNSW) [29] and Inverted File (IVF)[50]. HNSW is a proximity-graph-based index, whereas IVF is a clustering-based index. Although HNSW can deliver significantly higher throughput with a similar recall as the IVF index, the storage demands for HNSW indices (due to bidirectional links that need to be stored to connect nodes of the graph structure) are excessively large. At scale, these memory requirements surpass the capacity of most state-of-the-art CPUs, rendering HNSW impractical for the systems we aim to target. We provide an analysis of the metrics achievable by a small 10-billion-token HNSW and IVF index in Figure 4.

Inverted File: IVF clusters similar data together, enabling focused searches within these clusters rather than the entire dataset. The clustering is usually performed using a standard algorithm such as K-Means using Lloyd's algorithm [28]. At construction time, the `nlist` parameter controls how many clusters the dataset is divided into. Typically, `nlist` is $\propto \sqrt{N}$ where N represents the total number of vectors in the datastore. At search time, the `nProbe` parameter, controls how many unique clusters are searched within the IVF index. This parameter can be adjusted to balance latency and accuracy trade-offs. IVF can be combined with quantization schemes such as simple scalar quantization (SQ) and Product Quantization [13] to further decrease the index's latency and memory footprint, though this comes at some cost to the accuracy. We provide an analysis of how a variety of state-of-the-art quantization techniques affect the recall and vector size of our IVF indices in Table 1. In this work, we focus on IVF with the SQ8 quantization to develop our efficient retrieval scheme, as this approach optimizes memory footprint without hurting recall. Indices like HNSW incur too much memory overhead, making them impractical for at-scale deployment, while quantization methods other than SQ8 offer minimal benefits relative to their impact on recall.

2.2 Retrieval Enhanced Inference

In a RAG-enhanced LLM, the query is encoded to produce a vector for searching the index. This encoded query retrieves the IDs of the k closest neighbors, and a lookup then fetches the corresponding document text chunks. The retrieved document chunks can be re-ranked for relevance, using either similarity scores or advanced neural methods, and then integrated into inference by prepending them to the query [39] or cross-attending to their embeddings [3]. This enhanced query is then used to generate output tokens.

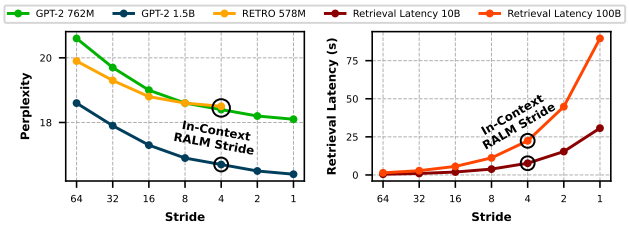

Figure 5: Prior works [16, 39] show that increasing retrieval stride allows models with half the parameters to match the accuracy of larger models, but at the cost of exponentially higher retrieval times. Circles indicate stride lengths identified as optimal for output accuracy in prior work [39].

Figure 3 presents a taxonomy of the end-to-end inference flow in RAG systems, illustrating how the non-parametric datastore is integrated with the LLM.

In state-of-the-art RAG systems, multiple retrieval iterations occur per query to refresh the documents used for generating every s tokens [39], a process known as retrieval striding. This approach aims to improve the accuracy of the final generated tokens by continuously updating the document set, allowing for more relevant information to be incorporated as the context changes over time from stride to stride.

Recent work from industry and academia, as shown in Figure 5, has demonstrated that retrieving new context after generating a number of tokens can improve overall LLM perplexity and quality of output [3, 16, 39]. As seen in Figure 5 increasing the retrieval stride frequency allows smaller inference models to achieve similar perplexity to models that have 2× the number of parameters, showing that a significant proportion of model output generation reliability can be placed on retrieval frequency. Unfortunately, retrieving new context every 4 tokens, which was suggested as an ideal point for optimizing accuracy in prior works [39], comes at a significant cost; for instance, for a 100-billion token scale datastore retrieving context every 4 versus 64 tokens increases the end-to-end latency by 12.12× (from 32.0s to 388.5s). For the purposes of our work, we choose a more conservative stride length of 16 to balance accuracy and run-time costs as a baseline; our final evaluation studies the impact of our design on various stride lengths.

3 Understanding System Bottlenecks in RAG

In this section, we look at the bottlenecks and trade-offs that exist with current RAG systems. We focus on performance and efficiency bottlenecks that arise when scaling to large datastore sizes found in publicly available datasets (e.g., Massive-DS [41]). We also show how, even with existing enhancements proposed by prior works, retrieval continues to throttle the performance of RAG-based LLMs.

TAKEAWAY 1: The retrieval phase and selection of retrieval stride length of RAG introduce overheads that scale linearly with datastore size. This leads to significant TTFT and end-to-end latency overheads in large-scale datastores. Figure 6 illustrates the TTFT and end-to-end latency of RAG-based LLMs as we scale the retrieval datastore size from 10B to 100B tokens. We set batch size to 32 across the entire pipeline. The inference model implements Gemma2-9B [43] with 512 input tokens and 256 output tokens running on an NVIDIA A6000 Ada GPU with

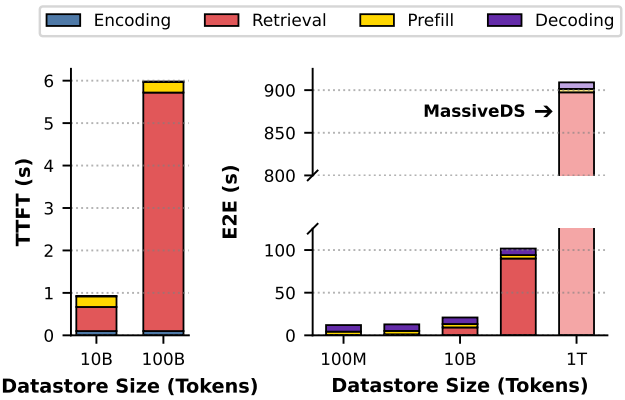

Figure 6: TTFT latency for 100B-token indices is much higher than for 10B, with end-to-end latency growing exponentially at a stride of 16 when generating 256 output tokens, reaching several minutes for 1T-token datastores at a batch size of 32. Extrapolated 1T-token latencies are shown in lighter color.

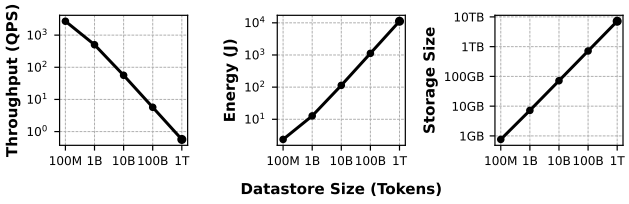

Figure 7: Throughput, Energy, and Memory Footprint scaling trends for an IVF retrieval index with 8-bit scalar quantization for different datastore sizes.

a retrieval stride length of 16 tokens. The retrieval stages are run using 32 cores on a server-class Intel Xeon Gold with 2.3GHz frequency. The retrieval encoding model is a modern, commonly used embedding model, BAAI general embedding (bge-large-en) [46].

In terms of TTFT, Figure 6(left) shows retrieval accounts for ≈61.21% and ≈93.98% of the latency for 10 billion and 100 billion token-sized datastores, respectively. Scaling the datastore from 10 billion to 100 billion tokens increases the retrieval latency from 5.62s to 56.1s (11.1×). As we further scale the datastore, we see roughly linear growth in latency with datastore size. Publicly available datastores like MassiveDS implement trillion-token scale datastores; here, retrieval latencies would continue to grow, taking several seconds to complete, precluding efficient at-scale deployment. Figure 6(right) shows the impact of retrieving relevant context from large datastores on end-to-end RAG-based LLM latency. While the end-to-end LLM latency is about 12.0s for 100M token datastores, the latency increases to 101.8s and 909.1s for 100-billion and trillion-token datastores, found in MassiveDS [41]. The impact of retrieval on end-to-end latency is due to not only large datastores but also repeated retrievals as new tokens are generated in the LLM decode phase.

TAKEAWAY 2: In addition to TTFT and end-to-end latency, RAG systems with large-scale datastores encounter significant challenges in terms of throughput, energy efficiency, and memory capacity demands. Figure 7 illustrates the impact of scaling retrieval datastore size on throughput (left), energy cost

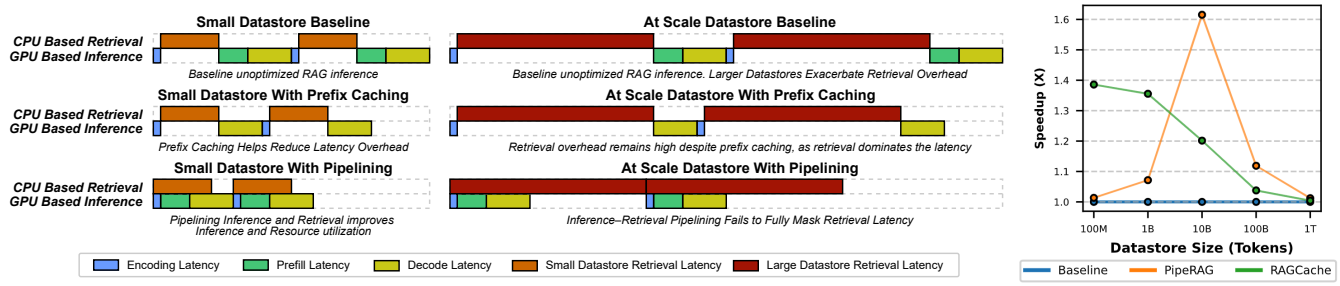

Figure 8: Prior enhancements improvements are not realized as well with larger datastores. Enhancements like prefix caching and pipelining are less effective with larger datastores due to longer retrieval times, causing GPU underutilization. In smaller datastores, pipelining overlaps better with retrieval latency, and prefix caching significantly reduces prefill stage latency. PipeRAG performs best when retrieval and inference latencies are similar, while RAGCache excels with small datastores, where prefill optimization has the greatest impact.

per retrieval (center), and index storage size (right). This analysis assumes an IVF index with 8-bit scalar quantization which achieves a recall of 0.94 (see Table 1). We choose this IVF configuration based on an hyperparameter sweep considering (1) retrieval algorithm, (2) product versus scalar quantization, and (3) recall. Throughout and energy are measured on a server-class Intel Xeon Gold CPU with 32 cores and Intel RAPL [21] to monitor power. As expected, increasing the datastore in terms of tokens has a direct, linear impact on index storage size; datastore sizes of 1 trillion tokens require nearly 10 TB of memory capacity using even memory-efficient IVF index types. Similar to memory capacity, increasing datastore size directly degrades retrieval throughput and energy efficiency. Figure 7 left and center show increasing datastore size by 10× has a nearly commensurate impact on throughput and energy. In fact, with 100 billion token-scale datastores, a single CPU achieves a throughput of only 5.69 QPS at the cost of nearly 1124 Joules per query. In contrast, a single NVIDIA A6000 Ada GPU, with a TDP comparable to the CPU, delivers a throughput of 132 QPS while only consuming 2.2 Joules per query during the prefill phase, and 67 QPS with the same energy consumption of 2.2 Joules per retrieval stride during the decode phase. These results are measured using the Gemma2-9B model with 512 input tokens and 256 output tokens with 16 token retrieval strides. These memory capacity, throughput, and energy efficiency limitations underscore the critical need for new strategies to efficiently handle trillion-token datastores at scale.

TAKEAWAY 3: While prior work has successfully enhanced the performance of RAG systems for smaller search indices, it has largely overlooked the immense retrieval overhead at the trillion-token scale, which significantly constrains the potential performance improvements these systems can achieve at scale. Recent research accelerating RAG systems has introduced two key dimensions of optimization:

- **PipeRAG: Pipelining retrieval and LLM inference** [16]: Pipelining allows systems to overlap retrieval on CPU and LLM inference on GPU, maximizing hardware utilization. Pipelining the stages relies on hiding the retrieval search with LLM inference of previously fetched documents; effectively queries use potentially stale documents when re-retrieving new documents after generating new tokens, as defined by the stride length.
- **RAGCache: Caching prefill computation for documents** [17]: Caching exploits the observation that subsequent retrievals for new documents may exhibit significant overlap in retrieved documents. The prefill stage can be shared across these documents, eliminating duplicated computation by caching key-value (KV) tensors. In our system, we assume we can achieve an ideal 100% hit rate in the key-value tensor cache, effectively minimizing the overhead associated with additional prefills during subsequent retrieval strides and maximizing the potential performance gains that can achieved through this work.

Figure 8 illustrates the impact of pipelining and caching on end-to-end RAG system latency as we vary the datastore size. While both pipelining and caching optimize end-to-end RAG system latency; TTFT latency and the energy cost of retrieval on large datastores are not optimized. Figure 8(left) shows an example baseline RAG system, both caching and pipelining provide good performance improvements. In particular, pipelining almost fully overlaps retrieval and LLM inference, saving up to 1.62× end-to-end latency. However, for large-scale datastores, the opportunity to overlap is reduced. Figure 8(center) shows retrieval latency dwarfs prefill and decoding (for a given, fixed stride length), here the performance benefits of both pipelining and caching are reduced. PipeRAG adjusts key retrieval parameters such as *nprobe*, which defines the retrieval search depth, and stride length to balance retrieval and LLM stages; however, in large datastores, sacrificing *nprobe* and stride length comes at the cost of output quality and model perplexity. Figure 8(right) shows as we increase datastore size to a modest 100B Tokens, the performance benefit from caching also monotonically decreases, as retrieval accounts for a larger fraction of execution time; pipelining provides benefit until retrieval and LLM stages can be perfectly overlapped; after this, the speedup diminishes.

4 Hermes Design Overview

In this section, we describe Hermes, an extensible system that co-designs retrieval algorithms and hardware for efficient RAG at scale. Hermes is designed to primarily optimize (1) time-to-first-token (TTFT) latency, (2) end-to-end (E2E) latency, and (3) energy of end-to-end RAG-based LLMs. Figure 9 illustrates the overall architecture of Hermes. Central to Hermes is a split-index design that distributes the typically singular large search index across multiple

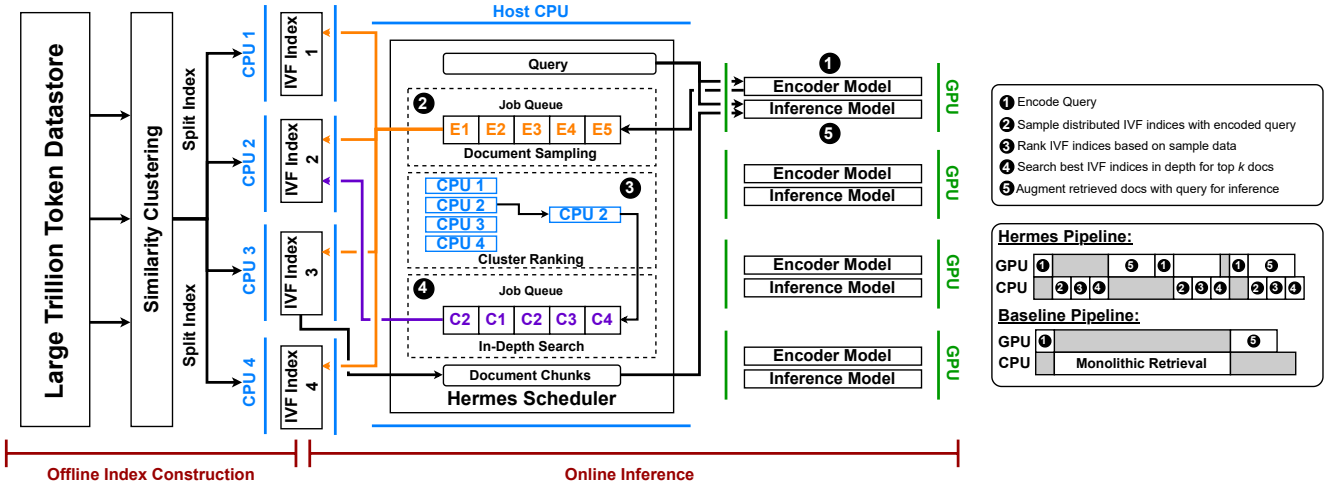

Figure 9: Hermes introduces three key enhancements to RAG pipelines: (1) partitioning datastores into separate IVF indices to enable more efficient parallel retrieval, and (2) implementing hierarchical searching through document sampling to navigate the distributed search space more effectively. Together, these innovations significantly reduce retrieval latency and energy consumption compared to traditional RAG pipelines that rely on a single, monolithic search index for the entire datastore.

Table 2: Hermes framework configurable parameters.

Configuration Aspect	Tuning Options
Latency & Accuracy	Sample Search Depth Deep Search Depth Number of Clusters to Search Number of Documents to Retrieve
Node Scaling	Number of Search Indices
Memory Efficiency	Size of Search Indices

nodes. Distributing the large non-parametric datastore across nodes and performing context retrieval in parallel on the smaller clusters directly reduces latency. However, naively splitting the datastores and concurrently searching *all* clusters is wasteful, levying high energy costs and limiting overall retrieval throughput. To address these challenges, Hermes begins by distributing indices into clusters based on their similarity (Section 4.1). It then employs a hierarchical search algorithm that first *samples* each cluster. Based on the sampling results, the algorithm identifies the clusters with the highest probability of producing relevant context and focuses on these for a more in-depth search. Despite reducing the number of in-depth searches, Hermes achieves equivalent retrieval quality, measured as NDCG (defined in Section 5) (Section 4.2). Hermes tunes the number of clusters created and searched, the partitioning of indices in clusters, and the trade-off between sampling and in-depth search to co-optimize end-to-end RAG LLM latency and energy, as well as retrieval throughput. Hermes's flexible design allows it to be easily adapted for various RAG deployment scenarios. Table 2 lists the tunable parameters that enable these customizations.

4.1 Distributed Retrieval Indices

As shown in Section 3, IVF search latency scales with the size of the datastore; IVF search through smaller indices exhibits significantly lower latency while improving energy efficiency and throughput. Hermes exploits this scaling trend to split monolithic datastores into smaller search indices that can be concurrently searched.

Unfortunately, naively dividing datastores into sub-clusters results in energy costs that are greater compared to that of searching a baseline monolithic index, as all clusters must still be searched to maintain accuracy comparable to searching the monolithic datastore. Hermes splits the search indices so that similar documents end up within the same cluster. This allows Hermes to only search a select number of "relevant" clusters to achieve equivalent accuracy as the monolithic search index. In order to find document similarity, we apply K-means clustering to the dataset to create a set of unique document clusters and then construct a separate IVF index for each resulting cluster (as illustrated Figure 10 (left)).

A key parameter in Hermes' index splitting procedure is determining the number of clusters to split the datastore into. To guide the number of clusters to split the non-parameteric datastore into, we rely on the fact that retrieval latency can be overlapped with subsequent LLM prompt computation and token generation phases, as shown by prior work (see Figure 8) [16]. Figure 10(right) shows the retrieval latency as we vary datastore size in comparison to latency of an NVIDIA A6000 Ada GPU performance inference using a Gemma 2-9B model. As an example, we find splitting an example 100B token datastore into 10 10B token clusters sufficiently hides the retrieval latency for our system.

However, not all clusters are equally sized. Due to the inherent randomness in the K-means initial centroid choice and despite the iterative improvement of these centroids during K-means training, some clusters still comprise more documents than others which leads to an imbalance in search latency across queries and clusters searched within a query. It is thus desirable to minimize (if not eliminate) the imbalance in the cluster sizes. The imbalance can be calculated in multiple ways (e.g. variance or entropy of the cluster sizes). We simply choose the ratio of the largest to the smallest cluster size as a proxy for imbalance. Seeding K-means with different initial random centroids leads to varying imbalances

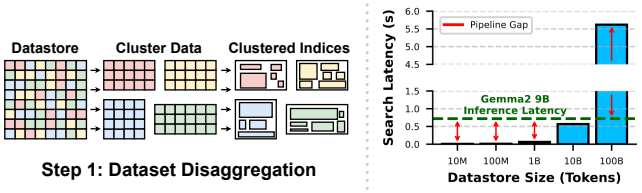

Figure 10: Clustering related data to each other allows us to create multiple smaller indices that can be searched in parallel. In this figure, we simplify the datastore clustering situation with each square representing a document of the dataset and color signifying similar documents.

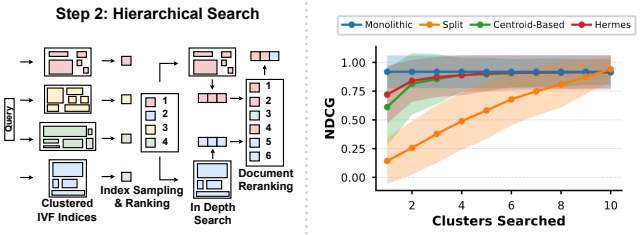

Figure 11: Hermes' hierarchical search begins with a document sampling step with a low nProbe value across all constructed indices. Based on the similarity scores of the retrieved documents, the top-performing indices are selected for a more detailed in-depth search. In this phase, multiple documents are retrieved from the selected indices. The retrieved documents are reranked, with the top-ranked ones used for inference.

in the resulting final clusters after training. We propose to quickly iterate across different seeds in multiple K-means runs to choose the lowest empirical imbalance for a given dataset. Given the size of the datastores, multiple K-means runs can be very costly. Hermes mitigates this cost by observing the imbalance on a small subset of documents. On an example 100M datastore, clustering based on even 1-2% tracks the clustering scheme very well for the larger datastore. This reduces the cost of a K-means run down to a few seconds for each seed during the offline index construction phase. On the example 100M datastore, we obtain the seed that minimizes imbalance the best (i.e. the gap between the largest to the smallest cluster, of 2×). We further address the inefficiency associated with accessing these uneven-sized clusters in Section 4.2.

4.2 Hierarchical Search via Document Sampling

Hermes' document clustering scheme reduces the search space for documents by allowing queries to focus on a subset of clusters while preserving accuracy. Since not all clusters will be equally relevant to a given query, Hermes saves energy and improves throughput by searching index clusters where the highest concentration of relevant data is likely to be located for a given query. Building on this insight, we propose a hierarchical search strategy that samples clusters to find the best indices for in-depth searches, enabling a more efficient retrieval without compromising retrieval quality.

Figure 11(left) shows the multi-step hierarchical search process implemented by Hermes. First, the encoded query embedding is used to perform *document sampling*, a coarse-grained search into all clusters based on the K-means clustering partitioning. A single document is retrieved from each of the clusters. Using the sampled documents, we rank the clusters for relevance based on the document's similarity to the original query. Using the most relevant clusters determined from our sampling, we conduct an *in-depth search* into those clusters for a larger set of retrieved documents. The final set of retrieved documents is then ranked to get the top k documents that are utilized for inference in the RAG pipeline. Central to Hermes' hierarchical search is using IVF's *nProbe* run-time parameter that defines the *search effort*; queries with larger nProbe values take longer to process but yield more relevant documents.

Intuitively, for the initial sampling, we use a low nProbe to rapidly sample a single document from each cluster. Instead of relying solely on centroid values, we retrieve documents directly from the clusters based on their relevance to the query. This method enhances retrieval accuracy by ensuring that documents are more closely aligned with the query and are not based on generalizations of all of the documents in the cluster, improving our ability to identify which clusters contain the most pertinent information.

The right portion of Figure 11 shows the accuracy implications of the centroid based searching method. We sweep the number of clusters in which an in-depth search is conducted and the impact on document quality as measured by NDCG. We compare Hermes to the split search (naively splitting the datastore into equal sizes search indices), to search that leverages the cluster centroids only, and a monolithic search, on a 100M datastore from a subset of the Common Crawl dataset [36]. We find that Hermes reaches iso-accuracy by searching only a small number of clusters in depth, whereas naive splitting requires searching nearly 10 clusters to achieve comparable accuracy. The figure also demonstrates the advantage of document sampling which gives better accuracy than the centroid-only search for a given number of in-depth clusters searched. We posit that searching 3 clusters in-depth in Hermes is a good design point that balances the in-depth search cost and accuracy.

Design Space Exploration: We also do a design space exploration case study (as seen in Figure 12) to determine what nProbe value to use for our sampling search and in-depth search, to optimize accuracy and latency in our setup. On the left, we vary the nProbe parameter for sampling and the number of clusters searched in the subsequent in-depth search (while using a fixed high nProbe for the in-depth search). As we increase the nProbe parameter we observe improving NDCG at the expense of latency. On the right, we assume a fixed nProbe of 8 for sampling, and vary the nProbe parameter for the in-depth search. Similarly, we find larger nProbe values yield higher NDCG at the expense of latency, however the latency overhead is more pronounced than the overhead seen for smaller search values. Through this analysis, we identify an optimal configuration with a small nProbe value of 8 and a large nProbe value of 128, which maximizes the end-to-end accuracy while not significantly impacting the latency.

Load Balancing Optimization: By intelligently splitting the data and searching only a subset of the nodes, Hermes improves throughput and energy efficiency. We further optimize Hermes' energy efficiency by leveraging Dynamic Voltage and Frequency Scaling (DVFS). As shown in Figure 13, when distributed search

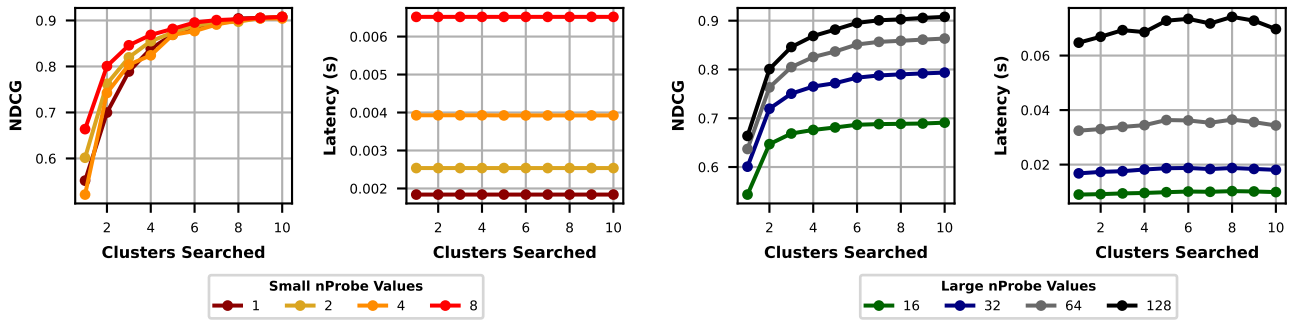

Figure 12: Design space exploration for small and large nProbe sweeps. In small nProbe sweeps, we vary the nProbe values during the fast search phase before transitioning to a fixed large nProbe of 128. In large nProbe sweeps, we hold the small nProbe at 8 and vary the large nProbe values. This analysis identifies an optimal configuration with a small nProbe of 8 and a large nProbe of 128, achieving a balance between efficient latency and reasonable accuracy.

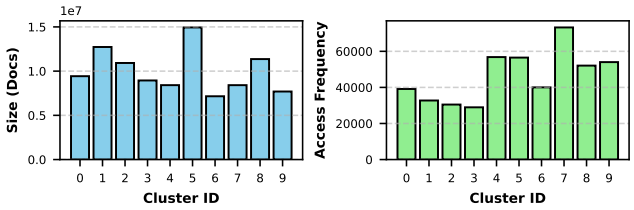

Figure 13: Cluster size and access frequency imbalance. Frequency analysis done analyzing queries from the Natural Questions Dataset [22].

indices are created using K-means clustering, the resulting clusters exhibit substantial variation in size. Some of the largest clusters are nearly twice as large as the smallest ones. Additionally, the frequency with which these clusters are accessed varies as well, with certain clusters being accessed more than twice as often as others. This imbalance leads to an uneven distribution of time spent searching through different clusters. Some clusters sit idle while waiting for a batch of queries to finish in the more intensely searched clusters. Intuitively, some clusters can be slowed down to save energy while incurring no latency or throughput cost on the system. Thus, instead of using the maximum CPU frequency to process all clusters uniformly, we propose to use DVFS dynamically at a query batch level, and to lower the frequency of the nodes that are allocated a lighter load for the in-depth search. This approach allows us to further reduce the energy consumption of Hermes by 10.1–14.5% as we vary the number of clusters searched in-depth.

5 Experimental Setup

We design a representative RAG-based LLM pipeline to evaluate performance trade-offs, leveraging state-of-the-art open-source models and retrieval indices.

Datasets and models. We use a BGE Large encoder model to encode queries at run-time [46]. For inference, we use several open-source models, including: Phi 1.5 1.3B [27], GEMMA 2 9B [43], and OPT 30B [47]. Our baseline assumes an input sequence length of 512 tokens and a generated output of 256 tokens, based on an average short length of queries and responses as seen in production systems [35]. We set our retrieval stride length to 16 tokens. Given our models and hardware systems, we set a default batch size of 128 for all stages.

Retrieval Indices. For our retrieval indices that contain less than 10B tokens we use a subset of Common Crawl [36]. We generate a synthetic set of embeddings for our retrieval indices, containing more than 10B tokens up to 100B tokens. To study the impact of Hermes' hierarchical search on recall, we construct indices from the 10B token subset of Common Crawl, ranging from 5GB to 11GB each and totaling 73GB in memory. Additionally, we construct synthetic indices up to 100B tokens that total up to 580GB of memory. End-to-end performance, power, and energy are evaluated using a combination of indices built using the 10B token Common Crawl dataset as well as synthetically created datasets.

For retriever performance and accuracy analysis, we leverage queries from the TriviaQA-test dataset [18] and Natural Questions dataset [22]. We leverage Normalized Discount Cumulative Gain (NDCG), with documents from an exhaustive brute-force search as our ground truth, to evaluate the accuracy of our designs. NDCG quantifies the quality of ranked results by comparing the relevance of retrieved items to the ideal ranking of the ground truth. It accounts for both the order of the results and their relevance.

During retrieval, we retrieve the 5 nearest document chunks. After each retrieval, we prepend the nearest chunk from the 5 (obtained via re-ranking using inner-product distance with the query vector) to the input query to form the prompt for generation.

Hardware/Software. For retriever experiments, we use 32 cores of an Intel(R) Xeon(R) Gold 6448Y, and the FAISS [6] library for ANN search. For Transformers, we leverage an NVIDIA A6000 Ada GPU, using the HuggingFace [45] library with vLLM [23], under FP16 precision. We use the Intel RAPL interface to measure power [21] for CPU-based retrieval and pynvml (nvidia-smi) to measure the power of our GPU based inference.

Multi-Node Analysis. To evaluate Hermes' design optimization on larger datastores across many nodes, we created a tool that leverages real hardware measurements on commodity hardware platforms and aggregates the measured numbers to estimate multi-node behavior. Figure 15 shows how the latency, power, and energy of individual index clusters and inference stages are measured on real Intel and ARM server-class CPUs and NVIDIA GPUs. We measure latency, power, and energy across different batch sizes,

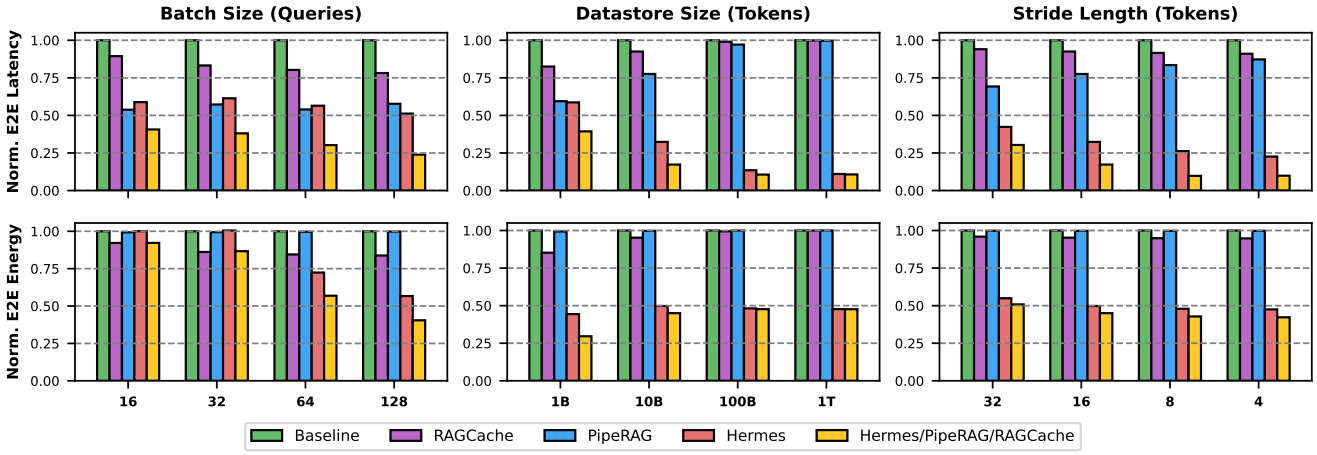

Figure 14: Comparison of Hermes with other prior acceleration techniques for at-scale retrieval indices. We find Hermes, under a wide variety of retrieval serving systems, can achieve improvements in latency and energy savings relative to the monolithic baseline. We use an IVF SQ8 index for our retriever paired with Gemini-2 9B models and BGE-Large for our inference and encoder models. For consistency, we standardize our batch size at 128 with a datastore size of 10 billion tokens and a stride length of 16 unless directly specified otherwise in the plot.

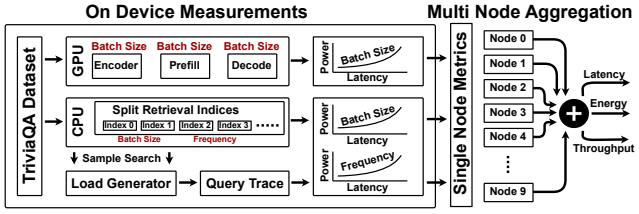

Figure 15: Our multi-node analysis tool collects measurements across the hardware platforms and uses that data to model end-to-end latency, energy, and throughput.

retrieval strides, and sequence lengths, constructing a comprehensive lookup table. Pairing these per-node on-device measurements with a trace of the top clusters accessed during the deep search based on TriviaQA [18], we aggregate the measurements to model the end-to-end latency, power, and energy of a Hermes-based RAG system.

6 Evaluation

To evaluate the performance of Hermes, we conducted a comprehensive evaluation of an end-to-end RAG pipeline (as seen in Figure 14. This pipeline leverages BGE Large as the encoder and Gemini 9B as the LLM. Throughout our study, we use IVF indices with 8-bit scalar quantization (SQ8, instead of FP32), configured with an nProbe value of 128 and `nlist` set to $\sqrt{n}$ where n is the total number of documents in our datastore, to ensure optimal accuracy and performance balance. For our in-depth search in Hermes, we identify 3 clusters as an ideal number of clusters for balancing performance with retrieval accuracy. We use a 512-token sequence as the length of our enhanced input that is passed into the LLM and generate 256 tokens of output with a retrieval stride length of 16 tokens. We compare Hermes against a baseline (unoptimized) RAG pipeline, PipeRAG [16], and RAGCache [17]. Our results demonstrate that while Hermes can deliver impressive performance as a standalone solution, we can further optimize performance when integrating Hermes with existing approaches like RAGCache [17] and PipeRAG [16], achieving added levels of efficiency and effectiveness. Due to the vastly different orders of magnitude of data we see from scaling some of the comparison metrics, we normalize the latency and energy values so that we may better illustrate the speed-ups across methods.

TAKEAWAY 1: Hermes allows us to achieve significant latency speedups and energy improvements through a diverse set of retrieval serving configurations, excelling in more computationally demanding environments. Figure 14 shows the impact Hermes has on the performance of RAG systems under various retrieval-oriented serving constraints. We explore the enhancements Hermes can achieve under different batch sizes (Figure 14(left)), different datastore sizes (Figure 14(center)), and varying stride lengths (Figure 14(right)). Through these metrics, we aim to show that under different retrieval serving configurations (retrieval load, length, and frequency), Hermes is still capable of showing latency improvements ranging from 2.45× – 10.25× and energy improvements ranging from 1.08× – 3.37×. To accurately model the latency and energy trends in Hermes, we use our multi-node analysis tool (as described in Section 5).

In Hermes, as well as the baseline, different batch sizes in the retriever stage are handled by FAISS, which schedules one thread per query and greedily processes the queries in a batch i.e. work stealing. Higher batch sizes allow better overlap of queries with each other, and fewer idle cores as a batch is completed. For all the batch sizes in our study, we obtain significant speed-ups relative to the baseline. We get on average 6.91× latency improvements by distributing a monolithic index over 10 machines in our experiments.

Distributed searching, while faster, can potentially lead to higher energy consumption than searching on a single machine over a monolithic index. However, since Hermes searches a subset of machines per query, we achieve overall better energy efficiency than

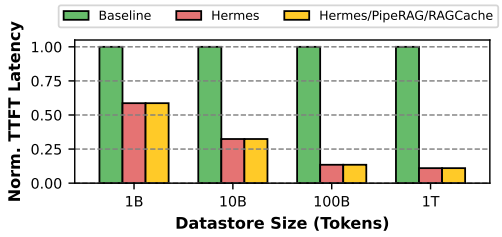

Figure 16: Hermes addresses the prohibitively expensive TTFT Latency with a 9.1× improvement in latency for trillion token datastores, an improvement that has not been addressed by prior works

naively searching all clustered indices in parallel, across all batch sizes. This underscores why Hermes is better than a naively distributed index that searches all machines; Hermes reduces latency as well as energy consumed by queries.

Examining trillion token datastores, Hermes achieves a 9.33× speed-up to latency while using 2.10× less energy. The most significant performance gains come from Hermes' distributed splitting strategy, especially for larger datastores. Since datastore search index times scale linearly with size, while datastore growth has been exponential, scaling down the search space results in multiplicative performance benefits. Although Hermes offers latency and energy efficiency improvements for smaller systems (e.g., 1 billion-token datastores), these gains are less pronounced because, in such cases, the critical performance bottleneck shifts to the GPU rather than the retrieval stage, limiting the overall impact of our enhancements.

We observe a similar trend in increasing the frequency of retrieval stride as with datastore size. Because Hermes excels in reducing the retrieval overhead through the distributed index searching strategy, increasing the frequency of retrieval cascades the enhancements from retrieval throughout every stride of the generation process. This gives us cumulative improvements in performance (reaching up to 10.12× improvements in latency at a stride length of 4 tokens). A similar trend holds true for energy, where at a retrieval stride length of 4 tokens, we see a 2.37× saving in energy.

TAKEAWAY 2: Hermes addresses the high latency costs associated with TTFT generation. In production environments, minimizing TTFT is crucial for delivering a positive user experience. Variations and imbalances in the TTFT can adversely affect the quality of service that production systems strive to maintain. Previous approaches like PipeRAG [16] and RAGCache [17] have primarily addressed inefficiencies in RAG from an inference standpoint. While these methods offer performance enhancements, they are limited by their reliance on information generated in prior inference strides, which constrains their ability to reduce TTFT. Additionally, as discussed in Section 3, a substantial portion of TTFT latency originates from the retrieval process itself and not from inference. By focusing on optimizing the retrieval process itself, Hermes achieves performance improvements within the retrieval stage that allow it to achieve 9.1× improvements in latency during TTFT at the trillion token scale.

TAKEAWAY 3: The adaptability of the Hermes Framework allows seamless integration across diverse inference model architectures and hardware. Hermes demonstrates significant performance improvements even when applied to models with

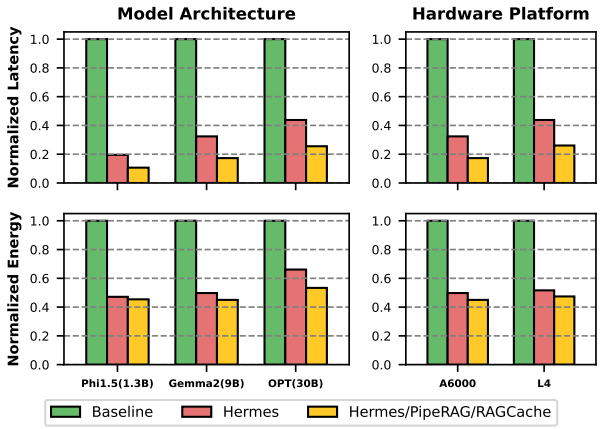

Figure 17: Hermes performance comparison under different inference serving configurations. Unless stated otherwise, all tests use the Gemma 2 model running on an A6000 Ada GPU. However, the OPT model requires two A6000 Ada GPUs to fit within the available memory, while the Gemma 2 model requires 2 L4 GPUs due to its memory requirements.

varying sizes, architectures, and inference times. Changes in the inference model, such as increased size or complexity, can impact latency and the efficiency of optimizations like pipelining, which benefit from the reduced retrieval times that Hermes offers. However, despite these changes in inference model size, Hermes continues to provide performance improvements of up to 3.92× with energy savings of 1.87× with larger models such as OPT (as illustrated in Figure 17). Hermes' flexible index splitting strategy allows the framework to easily be fine-tuned and its indices strategically split in different ways (larger or smaller distributed splits) to optimize performance for specific LLMs. This adaptability ensures that Hermes can be tailored to meet the unique requirements of different models and serving environments to best optimize RAG pipelines.

Since RAG systems at scale are predominantly bottlenecked by retrieval times and Hermes' focus is on optimizing retrieval, it consistently delivers performance improvements across diverse inference serving systems. However, as inference model latencies grow significantly, the performance benefits provided by Hermes become less pronounced. We can see how performance changes from 9.38× speedup with the smaller Phi 1.5 model (1.3 billion parameters) to only 3.92× speedup with the OPT model (30 billion parameters) while simultaneously requiring more energy (going from a 2.20× energy savings to only 1.87× energy savings)

Moreover, Hermes maintains its performance advantages across different LLM serving hardware platforms such as inference class L4 GPUs with a lower TDP compared to A6000 Ada's. This adaptability underscores the versatility of Hermes as a scalable solution suitable for a wide range of computational settings.

In our experiments shown in Figure 17, tests with the OPT model and L4 GPUs require two GPUs. The OPT model is too large to fit on a single GPU, and the L4 GPUs lack sufficient memory to support a Gemma 9B model on one GPU.

When considering the resource scaling for retrieval and inference, it is important to account for both the batch size and the model size being used. Depending on these configurations, adding

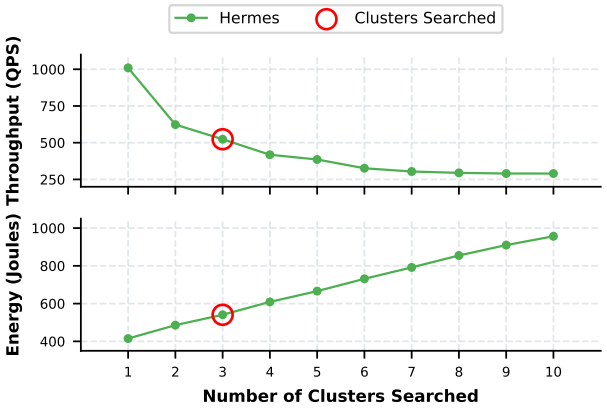

Figure 18: Hermes greatly improves the retrieval throughput and energy efficiency relative to a naively distributed retrieval scheme. Searching 3 clusters allows us to have 1.81× higher throughput and 1.77× energy savings compared to searching through all 10 clusters.

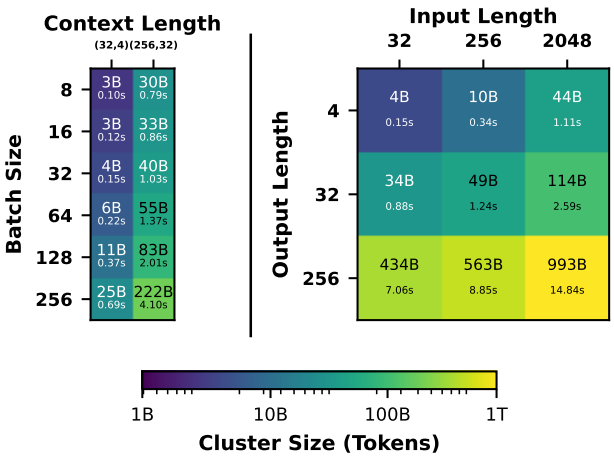

Figure 19: Analysis of how the optimal Hermes cluster size should be determined for effectively overlapping retrieval latency across various inference serving scenarios. In the left plot (32, 4) and (256, 32) indicate how many input and output tokens made up the context length respectively.

more resources can lead to higher energy consumption without proportional performance gains. In our setup, retrieval operates in batches of 128, while inference models leverage tensor parallelism across multiple GPUs as needed to accommodate the model in memory.

Our findings reveal that increasing the number of CPU cores beyond the batch size provides minimal performance gains without significantly impacting energy efficiency. However, for GPUs, adding additional devices (particularly with smaller models like Gemma2 9B) yields diminishing returns in performance. Tensor parallelism with smaller models results in substantial increases in energy usage while performance improvements remain minimal. This likely points to why even with L4 GPUs we are only able to achieve a 2.11× saving in energy compared to the 3.84× savings achievable with the A6000 ADA GPUs despite the fact L4 GPUs require much less power to operate. The communication overhead from distributing the model across GPUs, combined with the fact that A6000 Ada GPUs offer higher peak performance within a lower power budget compared to L4 GPUs (91 TFLOPS at 300 watts vs. 31 TFLOPS at 140 watts), explains why inference-scale L4s achieve less energy savings compared to general-purpose A6000 Adas.

This pattern is observable as well with larger models that require tensor parallelism across multiple GPUs to function. Depending on the inference model, latency performance benefits of adding multiple GPUs to inference setups can come at the cost of high trade-offs between energy consumption and performance scalability.

TAKEAWAY 4: Hermes improves the retrieval throughput and energy efficiency relative to a naively distributed retrieval scheme. Compared to a distributed retrieval system that unintelligently splits the datastore into N nodes and has to search and aggregate results from all machines for each query, Hermes intelligently splits the data so it is only required to search a subset of the nodes to reach the same accuracy. Hence, the reduced computation leads to significant throughput and energy improvements. We use our multi-node analysis tool to model Hermes and the naive split search. Figure 18 shows the system throughput and energy consumed per cluster searched with Hermes. The batch size evaluated is 128, over queries from the Natural Questions Dataset [22]. While the throughput of the naively distributed system is 290 queries/second (QPS), Hermes can improve this by 1.81× to reach comparable accuracy, by reducing the number of clusters searched to 3 (see Figure 11). Similarly, the energy consumed per batch is reduced to 1.77× that of using all 10 clusters. These benefits demonstrate the key advantages of Hermes over a naively distributed system, namely, throughput enhancements and improved energy efficiency.

TAKEAWAY 5: Hermes' design parameters enable it to be configured to effectively scale to a wide range of deployment scenarios, hiding retrieval latency under inference and matching retrieval and inference throughput. Hermes provides a broad range of tunable hyperparameters that can be optimized for various serving scenarios (see Table 2). These parameters enable Hermes to dynamically adapt to different RAG applications, handling diverse software and hardware deployment scenarios.

Inference. RAG-based LLMs can be used in various applications such as conversation-based tasks and coding-based tasks [42]. Recent work [35] shows that coding-based tasks require significantly fewer output tokens (with most containing less than 250 tokens) while conversation-based tasks, on average, generate greater than 250 tokens. These runtime parameters directly impact TTFT latency and end-to-end inference latency. To optimize across these diverse inference application scenarios, Hermes is designed to be adaptable to different use cases. Figure 19 shows inference latency across input and output sequence lengths, which can represent different tasks. Across these scenarios, Figure 19 shows how Hermes can be configured in terms of cluster sizes to effectively overlap retrieval latency with subsequent inference stages. For example, with a fixed output sequence of 32 tokens, as input sequence length increases from 32 to 2048 tokens, cluster sizes can increase from 34B to 114B, reducing the number of parallel clusters and retrieval nodes needed.

Hardware Architecture Because retrieval can be run on a diverse set of hardware platforms, Hermes provides many systems hyperparameters that can be adjusted for additional throughput

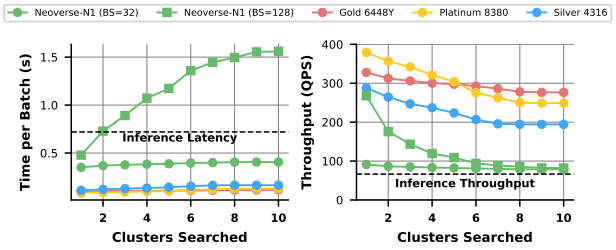

Figure 20: Different generations and architectures of processors impact retrieval latency compared to inference latency. By optimizing batch sizes, we can equalize throughput across various hardware platforms.

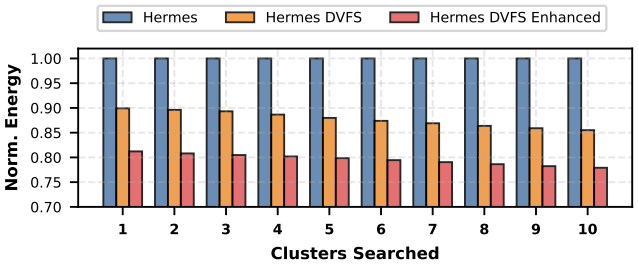

Figure 21: By slowing down the latency of Hermes retrieval to match the latency of inference, we can achieve additional energy savings. We see an average energy savings of 12.24% with baseline DVFS and 20.44% with the enhanced DVFS.

gains and energy savings based on the characteristics of an underlying platform. Figure 20 shows how different generations of CPU hardware with different microarchitectures can impact the latency and throughput of our Hermes search compared to inference, with the latest Intel generation (Platinum 8380) achieving the best throughputs (249 – 379 QPS) and latencies (0.084 – 0.13s). Although the ARM processor yields lower throughput and less favorable latency scaling, its higher core count allows us to run Hermes searches at a larger batch size to achieve comparably high levels of throughput (when only a few clusters are searched).

Additionally, as the Hermes search on Intel CPUs outperforms inference in both throughput and latency, we can deliberately slow it down to enable more aggressive DVFS. Because our inference and retrieval latencies are pipelined, a faster retrieval does not offer an added benefit, and slowing searches does not hurt performance. Figure 21 shows that by allowing the latency of individual searches to match the inference latency, instead of limiting DVFS based on the cluster that takes the longest to search through, the energy efficiency of DVFS further improves energy, ranging from an additional 18.8 to 22.1% savings compared to the 10.1 - 14.5% savings achieved when slowing down the frequency to the latency of only the slowest cluster. We see energy savings of 19.6% searching 3 clusters with this enhanced form of DVFS (the configuration used in our evaluation).

7 Related Work

RAG Acceleration: System-level techniques have been proposed to improve RAG performance, via pipelining RAG stages [16], caching document states [17], and speculative retrieval [49]. However, these techniques lose their efficacy as the datastore is scaled to very large sizes, a scenario where Hermes shines. While promising research on custom accelerators for RAG exists [11, 14, 15, 37, 38], we chose to focus on optimizations that can target currently deployable production systems at scale. Although advancements in accelerator technologies, such as near-memory processing proposed in [15], have demonstrated substantial performance gains for RAG, large-scale deployment of these solutions in datacenters is not yet a practical solution for high-volume serving.

Scaling Retrieval: Strategies have been proposed to scale the retrieval stage to larger datastores. Distributed retrieval indices that use horizontal scaling for larger datastores exist in commercial vector databases [7, 44] as well as literature [36, 48]. In naive distributed systems, the datastore is sharded, and a query search is broadcast to all nodes, and the results are aggregated. Such systems do reduce the latency and memory per machine, but cannot achieve throughput as high as Hermes, which does a subset of distributed nodes for in-depth search. Furthermore, to tackle the large memory footprint, reducing the precision (i.e. quantization of vectors [13]) has been proposed – however, this leads to a loss in accuracy. Moreover, DRAM/Disk hybrid indexes (i.e. storing partial or reduced-precision vectors in memory and full-precision on fast SSD disks) have been proposed in [5, 12], but this leads to only memory reduction, not latency. Hermes delivers improvement in memory footprint, latency, and throughput while maintaining accuracy.

Hierarchical K-means: Prior works explore hierarchical K-means trees for efficient searching [8, 32, 33]. Hermes differs from these works in that it distributes nodes and uses document sampling instead of just comparing top-level cluster centroids.

ANN Search & Databases: Recent studies in approximate nearest neighbor search include ANN Benchmarks [2] and Locality Sensitive Hashing (LSH) [9], which focus on efficient, high-dimensional querying. Unlike conventional hierarchical search methods found in database literature [10, 26], Hermes employs a flexible and dynamic approach tailored for RAG applications, offering a more scalable, efficient, and accurate solution than static hierarchical search frameworks.

IVF Optimizations: Within IVF, prior works [25, 48] use input/intermediate results to learn to predict search extent and terminate search early. SPANN [5] does query-time pruning of clusters based on the distance with the best centroid. These proposals do improve latency & throughput, but need to be used in conjunction with our distributed system to really scale to large datasets.

8 Conclusion

In this paper, we introduce Hermes, an algorithm-system co-design framework designed to address the challenges of scaling information retrieval in large-scale trillion-token datastores for RAG systems. By distributing retrieval search indices across multiple CPUs and employing a hierarchical search strategy that intelligently targets specific, organized search clusters, Hermes significantly enhances performance and energy efficiency in large-scale RAG systems without compromising accuracy. Our evaluation demonstrates that Hermes achieves up to a 9.33× speedup and 2.10× energy savings for trillion token datastores when integrated with previous state-of-the-art RAG acceleration methods [16, 17]. Furthermore, Hermes delivers up to 9.29× improvements in throughput compared

to more naive searching strategies, showcasing its potential to simply but effectively enhance the efficiency of large-scale retrieval systems for RAG.

Acknowledgments

We thank the anonymous reviewers and the artifact evaluation committee for their time and insightful feedback, which helped improve this work. This research was supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-2139899, as well as by NSF Awards No. 2118709 and CFF-2326608. Computational resources were generously provided by Chameleon Cloud [20] and Google.

References

[1] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511* (2023).

[2] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[3] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. *ICML* (2022).

[4] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[5] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[6] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The FAISS library. *arXiv preprint 2401.08281* (2024).

[7] Elastic. [n. d.]. Elasticsearch. https://www.elastic.co/elasticsearch/vector-database.

[8] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[9] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[10] Panagiotis G Ipeirotis and Luis Gravano. 2002. Distributed search over the hidden web: Hierarchical database sampling and selection. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 394–405.

[11] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. {CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 585–600.

[12] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[13] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[14] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. 2023. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[15] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. 2023. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. *arXiv preprint arXiv:2310.09949* (2023).

[16] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. 2024. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *arXiv preprint arXiv:2403.05676* (2024).

[17] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *arXiv preprint arXiv:2404.12457* (2024).

[18] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. 2017. TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. *ACL* (2017).

[19] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[20] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[21] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.

[22] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural Questions: a Benchmark for Question Answering Research. *Transactions of the Association of Computational Linguistics* (2019).

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*.

[24] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS* (2020).

[25] Conglong Li, Minjia Zhang, David G Andersen, and Yuxiong He. 2020. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2539–2554.

[26] Chung-Sheng Li, Philip S. Yu, and Vittorio Castelli. 1996. Hierarchyscan: A hierarchical similarity search algorithm for databases of long sequences. In *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE, 546–553.

[27] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks Are All You Need II: **phi-1.5** technical report. *arXiv preprint arXiv:2309.05463* (2023).

[28] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[29] Yu A Malkov and DA Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *TPAMI* (2020).

[30] Gary Marcus. 2020. The next decade in AI: four steps towards robust artificial intelligence. *arXiv preprint arXiv:2002.06177* (2020).

[31] Sewon Min, Suchin Gururangan, Eric Wallace, Hannaneh Hajishirzi, Noah A Smith, and Luke Zettlemoyer. 2023. SILO language models: Isolating legal risk in a nonparametric datastore. *arXiv preprint 2308.04430* (2023).

[32] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence* 36, 11 (2014), 2227–2240.

[33] David Nister and Henrik Stewenius. 2006. Scalable recognition with a vocabulary tree. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 2. Ieee, 2161–2168.

[34] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33, 5 (2024), 1591–1615.

[35] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[36] Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Dmytro Okhonko, Samuel Broscheit, Gautier Izacard, Patrick Lewis, Barlas Oğuz, Edouard Grave, Wen-tau Yih, et al. 2021. The web is your oyster-knowledge-intensive NLP against a very large web corpus. *arXiv preprint arXiv:2112.09924* (2021).

[37] Ruiyang Qin, Zheyu Yan, Dewen Zeng, Zhenge Jia, Dancheng Liu, Jianbo Liu, Zhi Zheng, Ningyuan Cao, Kai Ni, Jinjun Xiong, et al. 2024. Robust Implementation of Retrieval-Augmented Generation on Edge-based Computing-in-Memory Architectures. *arXiv preprint arXiv:2405.04700* (2024).

[38] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2024. Accelerating Retrieval-Augmented Generation. *arXiv preprint arXiv:2412.15246* (2024).

[39] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-Context Retrieval-Augmented Language Models. *TACL* (2023).

[40] Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. 2024. Blended RAG: Improving RAG (Retriever-Augmented Generation) Accuracy with Semantic Search and Hybrid Query-Based Retrievers. *arXiv preprint arXiv:2404.07220* (2024).
[41] Rulin Shao, Jacqueline He, Akari Asai, Weijia Shi, Tim Dettmers, Sewon Min, Luke Zettlemoyer, and Pang Wei Koh. 2024. Scaling Retrieval-Based Language Models with a Trillion-Token Datastore. *arXiv preprint arXiv:2407.12854* (2024).
[42] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. Dynamollm: Designing llm inference clusters for performance and energy efficiency. *arXiv preprint arXiv:2408.00741* (2024).
[43] Gemma Team. 2024. Gemma. (2024). https://doi.org/10.34740/KAGGLE/M/3301
[44] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.
[45] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2021. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv preprint 1910.03771* (2021).
[46] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding. arXiv:2309.07597 [cs.CL]
[47] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]
[48] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 995–1011.
[49] Zhihao Zhang, Alan Zhu, Lijie Yang, Yihua Xu, Lanting Li, Phitchaya Mangpo Phothilimthana, and Zhihao Jia. 2024. Accelerating retrieval-augmented language model serving with speculation. *arXiv preprint arXiv:2401.14021* (2024).
[50] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *Comput. Surveys* (2006).

A Artifact Appendix

A.1 Abstract

This artifact includes all the necessary data, source code, and scripts to reproduce the figures within the background, characterization, design, and evaluation sections of this study. First, we outline the steps for constructing retrieval indices with various configurations. Next, we provide profiling scripts used to measure the latency, energy consumption, and accuracy of these retrieval indices, along with several open-source inference models taken from Huggingface. Finally, our multi node analysis tool aggregates the system metrics to deliver comprehensive end-to-end analytics on throughput, latency, and energy for our Hermes Retrieval approach. Additional evaluation scripts analyze the accuracy of Hermes compared to other searching strategies.

A.2 Artifact check-list

- **Program:** Python and Shell
- **Model:** Multi-node latency and energy analysis tool provided; retrieval index construction scripts included; Hugging Face models publicly available.
- **Data set:** SPHERE (Encoded Common Crawl Subset), TriviaQA, Natural Questions
- **Run-time environment:** Anaconda on Ubuntu 24.04 with CUDA 12
- **Hardware:** Server class Intel CPU and Nvidia GPU
- **Run-time state:** Anaconda Environment with installed required packages and sudo access are required. An alternative docker image is also available at *michaeltshen/hermes-env:latest* for easy setup
- **Execution:** Sole User
- **Metrics:** Normalized Discount Cumulative Gain (NDCG), Recall, Throughput (QPS), Latency (s), Power (W), Energy (J)
- **Output:** Experiments produce log files (csv) and matplotlib plots. Expected output can be found within the paper.
- **Experiments:**
  - Hermes Accuracy Comparison to Other Search Strategies
  - Index nProbe Design Space Exploration
  - Cluster Size and Access Frequency Analysis
  - End-to-End Hermes Latency and Energy Comparison
  - TTFT Hermes Latency Analysis
  - Hermes Cluster Access Energy Throughput Trend
  - Hermes Different Hardware Latency Throughput Analysis
  - Hermes DVFS Analysis
- **How much disk space required (approximately)?:**
  - Docker Environment: ≈ 32GB
  - Creating ISCA Plots: ≈ 32GB
  - 100K Indices: ≈ 32GB
  - 899M Indices: ≈ 6-7TB
- **How much time is needed to prepare workflow (approximately)?:**
  - Setting up Environment: ≈ 15–30 Min
- **How much time is needed to complete experiments (approximately)?:**
  - Creating ISCA Plots: ≈ 1–2 Hours
  - 100K Indices: ≈ 1–2 Days
  - 899M Indices: ≈ Several Weeks (Depending on Hardware)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** GitHub (github.com/S4AI-CornellTech/Hermes) and Zenodo (doi.org/10.5281/zenodo.15258027).

A.3 Description

A.3.1 How to access. Code for this paper can be accessed from the live public GitHub repository at github.com/S4AI-CornellTech/Hermes. The artifact is also available on Zenodo at doi.org/10.5281/zenodo.15079515.

A.3.2 Hardware dependencies. To reproduce the results presented in this paper, we recommend using state-of-the-art server-class hardware. For CPUs we recommend a machine with at least 32 cores and 512 GB of memory (e.g., Intel Xeon Platinum 8380) and high-performance NVIDIA server-grade GPUs (e.g., A6000 Ada). In the absence of such hardware, reproduction is still feasible using any multi-core CPU with ample memory and a compatible NVIDIA GPU.

A.3.3 Software dependencies. For LLM inference, we use PyTorch on NVIDIA GPUs with CUDA version 12. Our models are implemented using the Hugging Face `transformers` library, along with the `datasets` library. We also use `vLLM` to deploy HuggingFace models efficiently and to achieve state-of-the-art inference performance. For retrieval, we leverage the `FAISS` library for efficient similarity search. To profile power consumption, we use a combination of `pyRAPL`, `pynvml`, `perf`, and `rapl_read`.

A.3.4 Datasets. We conduct our experiments using the SPHERE dataset, a pre-encoded subset of Common Crawl. SPHERE is available in three sizes: 100K, 100M, and 899M document subsets. These can be accessed on Hugging Face at `mohdumar/SPHERE_100K`,

Table 3: Index Construction Customization

Configuration	Explanation
Index Size	Specifies the size of the monolithic index to build, or the dataset size to use for clustered or split indices (100K, 100M, 899M for non-synthetic indices).
Number of Indices	Applicable only for split or clustered indices. Specifies how many separate indices the dataset should be divided into.

`mohdumar/SPHERE_100M`, and `mohdumar/SPHERE_899M`. To evaluate our retrieval indices, we additionally use the TriviaQA dataset. An encoded version of the dataset queries is provided at `triviaqa/triviaqa_encodings.npy`.

A.3.5 Models.

- Multi Node Latency Aggregation: Provided at `modeling/latency_sim.py`
- Multi Node Energy Aggregation: Provided at `modeling/dvfs_sim.py`
- FAISS Search Indices: Construction Scripts Provided at `index/`
- Huggingface Models: Publicly Available on the Huggingface model hub.

A.4 Installation

To set up Hermes, you can install it natively by cloning the public GitHub repository and configuring the provided Anaconda environment. Alternatively, a pre-built Docker image (approximately 30 GB) is available for installation.

A.5 Experiment workflow

This workflow describes how to set up the Hermes experimental environment natively. Steps 1 - 6 can be skipped by pulling the pre-built docker image instead.

(1) Set up Conda environment and clone GitHub repository.
(2) Download the encoded TriviaQA dataset.
(3) Install required dependencies in the Conda environment, including `faiss`, `transformers`, `vllm`, `datasets`, `pynvml`, `pyRAPL`, and other required dependencies.
(4) Resolve any `torchvision` dependencies, if applicable.
(5) Use `sudo` to enable access to RAPL energy metrics.
(6) Build `rapl-read` from the `uarch-configure` repository.
(7) Construct Monolithic, Split, Flat, and Hermes-clustered search indices.
(8) Profile search latency and power consumption across different configurations.
(9) Measure latency and power usage of state-of-the-art encoder and inference models across different configurations.
(10) Generate cluster access traces from the encoded TriviaQA dataset.
(11) Perform multi-node aggregation for latency and energy profiling.
(12) Evaluate the accuracy of each search index.
(13) Generate and visualize plots for analysis.

A.6 Evaluation and expected results

By following the workflow outlined in Section A.5, the results presented in Figures 6, 7, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, and 21 can be reproduced. While we do not provide scripts for Figures 7 and 10 (since they require constructing and profiling multiple indices of varying sizes, which would substantially increase the complexity of the workflow), these figures primarily serve as background and are not central to the paper's core contributions. To maintain a more streamlined workflow, we chose not to include them. Nonetheless, all figures can be regenerated using the approaches described below.

- **Fig 6**: Build Indices of the specified size in A.5 Step 7 and collect profiled data from Steps 8 and 9
- **Fig 7**: Build Indices of the specified size in A.5 Step 7 and collect profiled data from Step 8
- **Fig 10:** Build Indices of the specified size in A.5 Step 7 and collect profiled data from Steps 8 and 9
- **Fig 11**: Build Indices in A.5 Step 7 and then run the accuracy evaluation from Step 12.
- **Fig 12**: Build Indices in A.5 Step 7 and collect profiled data from Step 8. Then run the accuracy evaluation from Step 12.
- **Fig 13**: Build Indices in A.5 Step 7 and collect trace data from Step 10.
- **Fig 14**: Build Indices of the specified size in A.5 Step 7 and collect profiled data from Step 8 and 9. Then use the multi-node aggregation tools from step 11.
- **Fig 16**: Build Indices of the specified size in A.5 Step 7 and collect profiled data from Step 8 and 9. Then use the multi-node aggregation tools from step 11.
- **Fig 17**: Build Indices in A.5 Step 7 and collect profiled data of specific models on specific hardware from Step 8 and 9. Then use the multi-node aggregation tools from step 11.
- **Fig 18**: Build Indices in A.5 Step 7 and collect profiled data from Step 8. Then use the multi-node aggregation tools from step 11.
- **Fig 19**: Build Indices of various sizes in A.5 Step 7 and collect profiled data of different model configs in Step 8 and Step 9.
- **Fig 20**: Build Indices in A.5 Step 7 and collect profiled data of different model configs in Step 8 on several different hardware platforms. Then use the multi-node aggregation tools from step 11.
- **Fig 21**: Build Indices in A.5 Step 7 and collect profiled data of different model configs in Step 8. Then use the multi-node aggregation tools from step 11.

A.7 Experiment customization

The source code for index construction and profiling is designed to be highly customizable, allowing users to specify the types of indices to build and explore a wide range of Hermes-based RAG configurations via command-line arguments to the provided scripts.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

Table 4: Index Profiling Customization

Configuration	Explanation
nProbe	How many centroids to search in the search index
Sample and Deep nProbe	Applicable only for clustered indices in the Hermes search. Specifies how many centroids to search during the sampling and deep search.
Batch Size	How many queries are in each index search
Queries	Option to change the dataset if you have access to another encoded dataset
Retrieved Docs	How many documents to retrieve per query for each search
Number of Threads	How many threads to use per search

Table 5: Model Profiling Customization

Configuration	Explanation
Model Name	The target LLM model. A full list of supported models is available in the VLLM documentation: https://docs.vllm.ai/en/latest/models/supported_models.html
Number of GPUs	Number of GPUs to use for inference.
Batch Size	Number of queries processed per inference batch.
Input Length	Input token sequence length per query.
Output Length	Number of tokens to generate per query.