Leveraging Approximate Caching for Faster Retrieval-Augmented Generation

Shai Bergman
Huawei Research
Zurich, Switzerland

Zhang Ji
Huawei Research
Zurich, Switzerland

Anne-Marie Kermarrec
EPFL
Lausanne, Switzerland

Diana Petrescu
EPFL
Lausanne, Switzerland

Rafael Pires
EPFL
Lausanne, Switzerland

Mathis Randl*
EPFL
Lausanne, Switzerland

Martijn de Vos
EPFL
Lausanne, Switzerland

Abstract

Retrieval-augmented generation (RAG) enhances the reliability of large language model (LLM) answers by integrating external knowledge. However, RAG increases the end-to-end inference time since looking for relevant documents from large vector databases is computationally expensive. To address this, we introduce Proximity, an approximate key-value cache that optimizes the RAG workflow by leveraging similarities in user queries. Instead of treating each query independently, Proximity reuses previously retrieved documents when similar queries appear, reducing reliance on expensive vector database lookups. We evaluate Proximity on the MMLU and MedRAG benchmarks, demonstrating that it significantly improves retrieval efficiency while maintaining response accuracy. Proximity reduces retrieval latency by up to 59% while maintaining accuracy and lowers the computational burden on the vector database. We also experiment with different similarity thresholds and quantify the trade-off between speed and recall. Our work shows that approximate caching is a viable and effective strategy for optimizing RAG-based systems.

CCS Concepts: • **Information systems → Retrieval models and ranking**; • **Computing methodologies → Natural language generation**.

Keywords: Retrieval-Augmented Generation, Large Language Models, Approximate Caching, Neural Information Retrieval, Vector Databases, Query Optimization, Latency Reduction, Machine Learning Systems

*Corresponding author

ACM Reference Format:
Shai Bergman, Zhang Ji, Anne-Marie Kermarrec, Diana Petrescu, Rafael Pires, Mathis Randl, and Martijn de Vos. 2025. Leveraging Approximate Caching for Faster Retrieval-Augmented Generation. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3721146.3721941

1 Introduction

Large language models (LLMs) have revolutionized natural language processing by demonstrating strong capabilities in tasks such as text generation, translation, and summarization [1]. Despite their increasing adoption, a fundamental challenge is to ensure the reliability of their generated responses [2]. A particular issue is that LLMs are prone to *hallucinations* where they confidently generate false or misleading information, which limits their applicability in high-stake domains such as healthcare [3]. Moreover, their responses can be inconsistent across queries, especially in complex or specialized domains, making it difficult to trust their outputs without extensive verification by domain experts [2, 4].

Retrieval-augmented generation (RAG) is a popular approach to improve the reliability of LLM answers [5]. RAG combines the strengths of neural network-based generation with external information retrieval. This technique first retrieves relevant documents from an external database based on the user prompt and appends them to the user prompt before generating a response. RAG aids the LLM to use vetted, flexible sources of information without the need to modify the model parameters through retraining or fine-tuning [6]. Both user queries and documents are often represented as high-dimensional embedding vectors, capturing semantic meanings, and these embeddings are stored in a vector database. Retrieving relevant documents involves finding embeddings in the database closest to the query embedding, a process known as nearest neighbor search (NNS). NNS, however, becomes computationally expensive for large vector databases [7, 8]. Thus, RAG can significantly prolong the inference end-to-end time [9].

To mitigate the latency increase of NNS, we observe that user query patterns to conversational agents often exhibit spatial and temporal locality, where specific topics may experience heightened interest within a short time span [10]. Similar queries are likely to require and benefit from the same set of retrieved documents in such cases. Leveraging this, we propose reducing the database load by reusing recently retrieved results for similar past prompts. This approach contrasts with conventional RAG systems that treat queries independently without exploiting access patterns. However, exact embedding matching is ineffective when queries are phrased slightly differently, as their embeddings are unlikely to match precisely. We introduce a novel *approximate caching* mechanism incorporating a similarity threshold to address this. Approximate caching allows for some level of tolerance when determining relevant cache entries.

This work introduces Proximity, a novel approximate key-value cache specifically designed for RAG-based LLM systems. By intercepting queries before they reach the vector database and leveraging previously retrieved results for similar queries, Proximity reduces the computational cost of NNS and minimizes database accesses, effectively lowering the overall end-to-end inference latency of the RAG pipeline. Specifically, we store past document queries in an approximate key-value cache, where the key corresponds to the embedding of a previous query, and the value is the set of relevant documents retrieved for that query. When a new query is received, the cache checks if it is sufficiently similar to any of the keys. If so, the corresponding documents are returned, bypassing the need for a database lookup. If not, the database is queried for new results, the cache is updated with the new query, and the RAG pipeline proceeds as usual. The main technical challenge that this work addresses is the design and parametrization of the cache, as there is an inherent trade-off between accuracy and performance.

We implement Proximity and evaluate our system using the Massive Multitask Language Understanding (MMLU) [11] and MedRAG [12] benchmarks, which are commonly used to evaluate RAG frameworks. Our results show significant speed improvements while maintaining retrieval accuracy. Specifically, we find that Proximity reduces the latency of document retrieval by up to 59% for MMLU and 70.8% for MedRAG with no or only a marginal decrease in accuracy. These results highlight the viability and effectiveness of using approximate caching to improve the speed of RAG-based LLM systems.

Our contributions are as follows:

- We introduce Proximity, a novel approximate key-value caching mechanism for RAG pipelines that leverages spatial and temporal similarities in user queries to reduce the overhead of document retrieval (Section 3). Proximity includes a similarity-based caching strategy that significantly lowers retrieval latency while maintaining high response quality.
- We evaluate Proximity using two standard benchmarks, demonstrating substantial improvements in cache hit rates and query latency while maintaining comparable accuracies (Section 4). We compare Proximity against RAG-based solutions without such an approximate caching mechanism. We systematically analyze the impact of the cache capacity and similarity tolerance, providing insights into optimizing retrieval performance for workloads with differing characteristics.

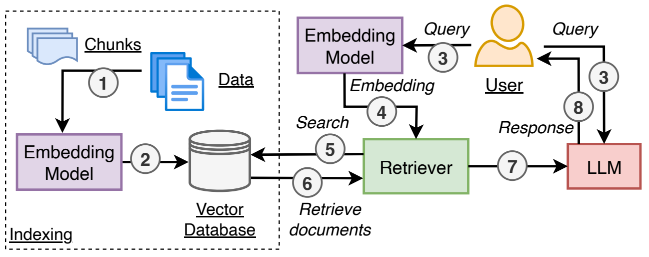

Figure 1. The RAG workflow.

2 Background and preliminaries

We first detail the RAG workflow (Section 2.1) and then outline the process to retrieve the documents relevant to a user query (Section 2.2).

2.1 Retrieval-augmented generation

Retrieval-augmented generation (RAG) is a technique that enhances the capabilities of LLMs by integrating information retrieval before the generation process [5]. RAG typically enables higher accuracy in benchmarks with a factual ground truth, such as multiple-choice question answering [13].

Figure 1 shows the RAG workflow that consists of the following eight steps. Before LLM deployment, raw data (*e.g.*, documents or videos) are first converted into chunks, and each of these chunks is converted into a high-dimensional embedding vector using an embedding model (step 1) and stored in a vector database (2). When the user queries the LLM (3), this query is first converted to an embedding vector using the same embedding model as used for the indexing and passed to the retriever (4). The vector database then searches for embeddings close to the query embedding (*e.g.*, using some distance metric, step 5) and returns the relevant data chunks related to this embedding (6). These data chunks and the user query are combined into a single prompt and passed to the LLM (3 and 7). The LLM response is then returned to the user (8).

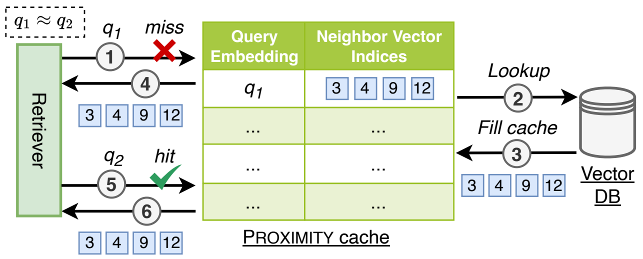

Figure 2. The design and workflow of the PROXIMITY approximate cache when receiving two subsequent, similar query embeddings q_1 and q_2. q_1 results in a cache miss whereas q_2 results in a hit, returning similar document indices as for q_1.

```
Algorithm 1: Proximity Search
State : similarity tolerance $\tau$, cache capacity $c$,
        vector database $\mathcal{D}$, cache state $C = \{\}$

Procedure LOOKUP($q$):
    $d$ = [DISTANCE($q, k$) for $k$ in $C$.keys]
    ($key, min\_dist$) ← $min(d)$
    if $min\_dist \le \tau$ then
        return $C[key]$
    $\mathcal{I}$ ← $\mathcal{D}$.RETRIEVEDOCUMENTINDICES($q$)
    if $|C| \ge c$ then
        // Evict an entry if cache is full
        $C$.EVICTONEENTRY()
    $C[q] \leftarrow \mathcal{I}$
    return $\mathcal{I}$ // Return retrieved indices
```

2.2 RAG vector search

The vector search during the RAG workflow (step 5) obtains relevant embeddings from a vector database based on the embedding vector of the user query. Vector databases are databases that potentially store a vast amount of n-dimensional real-valued vectors and are optimized to solve the nearest neighbor search (NNS), *i.e.*, finding the k elements contained in the database that are the closest to a given query [14]. The similarity metric to be minimized is typically L2, cosine, or inner-product, and is fixed before deployment. This lookup returns a ranked list of indices corresponding to resulting embeddings, and these indices can then be used to obtain the data chunks that will be sent along with the user prompt to the LLM.

Due to the high dimensionality of embeddings and the sheer volume of data in modern vector databases [15], performing vector searches at scale poses significant computational challenges. NNS requires comparing query embeddings with millions or billions of stored vectors, which becomes expensive as the database grows [16]. Even with optimized index structures such as as HNSW [17] or quantization-based approaches [18], maintaining low-latency retrieval while ensuring high recall remains difficult.

3 Design of PROXIMITY

We mitigate the efficiency challenges associated with NNS during RAG by designing a caching mechanism that reduces the need for repeated NNSs by reusing previously retrieved results for similar queries. Our design is motivated by the observation that user queries with conversational LLM agents show spatial and temporal similarities [10]. However, traditional caching mechanisms, such as exact key-value stores, are ineffective in the context of vector search due to the nature of embeddings. User queries rarely produce identical embeddings due to small variations in input phrasing, making exact caches inefficient, as lookups would almost never result in cache hits.

Instead, we leverage *approximate caching* in the context of RAG document retrieval. Even if the data chunks retrieved for a given query are not the most optimal results that would have been obtained from a full database lookup, they can still provide valuable context and relevant information for the LLM, allowing the system to maintain good accuracy while reducing retrieval latency. Our approximate cache is parameterized with a similarity threshold τ. If the distance between two query embeddings q_1 and q_2 is equal to or less than τ, we consider these embeddings alike and return similar data chunks if they are available in the cache. Cache hits thus bypass more expensive vector database searches. The technical challenge lies in defining an effective similarity threshold that maximizes cache hits without compromising response relevance and finding the optimal cache size that enables high cache coverage while still being computationally attractive.

We now present the design of PROXIMITY, an approximate key-value cache designed to accelerate RAG pipelines. PROXIMITY is agnostic of the specific vector database being used but assumes that this database has a LOOKUP function that takes as input a query embedding and returns a sorted list of indices of vectors that are close to the query. In the following, we first present the high-level process overview of retrieving vectors with PROXIMITY (Section 3.1). We then elaborate on the parameters and components of our caching mechanism (Section 3.2).

3.1 Retrieving relevant documents with PROXIMITY

The high-level algorithm is described in Algorithm 1. We also visualize the PROXIMITY cache and workflow in Figure 2 with an example when sending two subsequent similar query embeddings q_1 and q_2. Each key in the cache corresponds to an embedding previously queried, while the associated value is a list of the top-k nearest neighbors retrieved from the database during a previous query. The cache has a fixed capacity of c entries, which implies that, when full, an eviction policy is applied to make room for new entries (see Section 3.2.2).

When a user initiates a query, it is first converted into an embedding vector by an embedding model. The retriever (left in Figure 2) then forwards this query embedding, denoted as q_1, to the Proximity cache (step 1 in Figure 2). Proximity first checks whether a similar query has been recently processed by iterating over each key-value pair (k,v) of cache entries (line 3 in Algorithm 1). If the best match is sufficiently close to the query, *i.e.*, the distance between q and k is lower than some threshold τ, the associated retrieval results are returned immediately (Lines 4-6), thus bypassing the vector database. Otherwise, the system proceeds with a standard database query (Line 7 in Algorithm 1). This is step 2 in Figure 2 where we perform a lookup with q_1 and the vector database. The Proximity cache is now updated with the resulting neighbor vector indices (in blue) from the database lookup (step 3). Since the number of cache entries might exceed the cache size, the eviction policy will remove a cache entry if necessary (Lines 9-10). The cache is now updated with the result obtained from the database (Line 11). Finally, the vector indices are returned to the retriever (step 4 in Figure 2). We adopt the same distance function as the underlying vector database to ensure consistency between the caching mechanism and the retrieval process.

When another query embedding q_2 arrives (step 5 in Figure 2), with a low distance to q_1, Proximity first checks if it is sufficiently similar to any stored query embeddings in the cache. Suppose the similarity score is below the predefined similarity threshold τ. In that case, the cache returns the previously retrieved document indices associated with the closest matching query, thus bypassing a lookup in the vector database (step 6). Depending on the specifications of the cache and workload, our approach can reduce retrieval latency and computational overhead, especially in workloads with strong spatial or temporal similarities.

3.2 Proximity cache parameters and components

We now describe the parameters and components of the Proximity cache and discuss their impact on performance.

3.2.1 Cache capacity c. The cache has a capacity of c entries, which dictates the number of entries it will fill before starting to evict old entries (*i.e.*, the number of rows in Figure 2). This parameter poses a trade-off between the cache hit rate and the time it takes to scan the entire set of keys. A larger cache increases the likelihood of cache hits, allowing the system to reuse previously retrieved data chunks more frequently and reducing the number of expensive vector database lookups. However, increasing the cache size also incurs *(i)* computational overhead as the cache must be searched for similarity matches on each query and *(ii)* memory overhead since additional key-value pairs need to be stored. This computational overhead is manageable since the cache size is small compared to the full vector database. Our current implementation does a linear scan over the keys in the Proximity cache, resulting in a linear compute overhead as c increases. Even with a full linear scan over the cached keys, we found the overhead to be negligible when compared to a database query.

3.2.2 Eviction policy. When the cache reaches its maximum capacity, an eviction policy is required to determine which existing entry should be removed to make space for new ones. While numerous eviction strategies exist, we opted for the FIFO (first-in, first-out) policy. It evicts the oldest entry in the cache, irrespective of how often or recently it has been accessed. FIFO provides a simple and predictable replacement strategy.

3.2.3 Distance tolerance τ. We employ a fuzzy matching strategy based on a predefined similarity threshold τ to determine whether a cached entry can be used for a given query. This threshold defines the maximum allowable distance between a query embedding and a cached query embedding to be considered similar. The choice of τ directly impacts recall and overall RAG accuracy. A low τ value enforces stricter matching, ensuring that retrieved data chunks are highly relevant but potentially reducing the cache hit rate, limiting the benefits of caching. We note that $\tau = 0$ is equivalent to using a cache with exact matching. Conversely, a higher value of τ increases cache utilization by accepting looser matches, improving retrieval speed at the potential cost of including less relevant data chunks. In our experiments, τ is treated as a global constant, manually set at the start of each evaluation. However, one might consider adaptive strategies to dynamically adjust τ based on the characteristics of the data chunks stored or on the patterns of queries sent to the system. Exploring such adaptive mechanisms could further optimize retrieval efficiency.

4 Evaluation

We implement Proximity and evaluate the effectiveness of our approximate cache using two standard benchmarks. Our experiments quantify the effect of the cache capacity c and similarity tolerance τ on the accuracy, the cache hit rate, and the latency required to retrieve documents.

4.1 Implementation details

We implement Proximity in the Rust programming language for efficiency and publish its implementation [1]. The first-in, first-out (FIFO) eviction policy is implemented using a growable ring buffer from the Rust standard collection. For vector

[1] https://github.com/sacs-epfl/proximity.

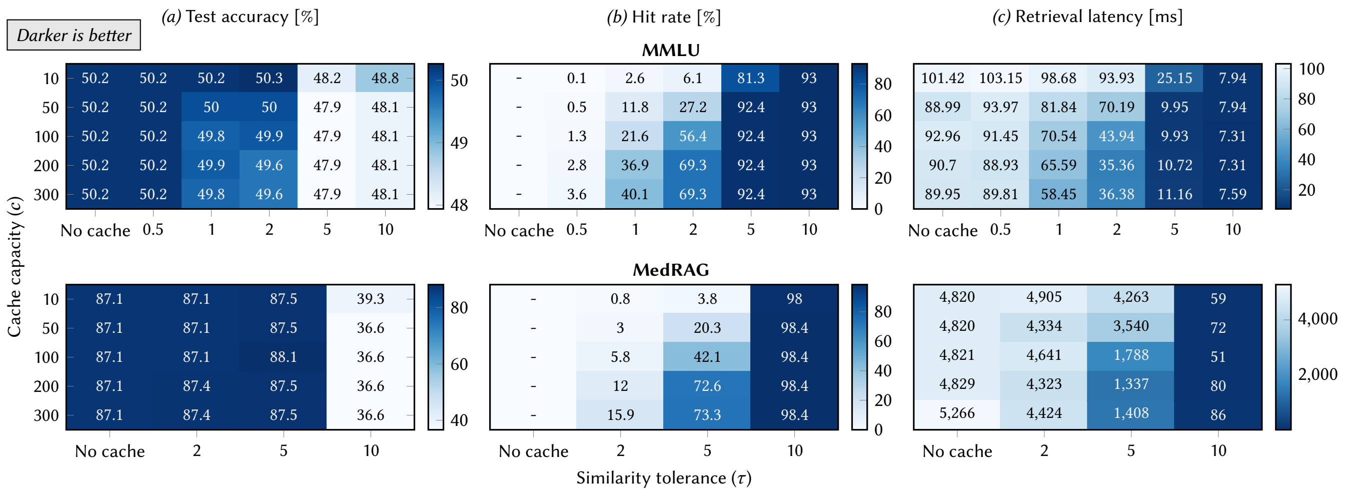

Figure 3. The accuracy (left), cache hit rate (middle), and latency of document retrieval (right), for different cache capacities and similarity tolerances, for the MMLU (top) and MedRAG (bottom) benchmarks.

comparisons, our implementation makes use of PORTABLE-SIMD[2], an experimental extension to the Rust language that enables ISA-generic explicit SIMD operations.

We expose bindings from the Rust cache implementation to the Python machine learning pipeline using PyO3[3] (Rust-side bindings generation) and Maturin[4] (Python-side package management). These bindings streamline the integration of our cache into existing RAG pipelines.

4.2 Experimental setup

To evaluate Proximity, we adopt and modify two existing end-to-end RAG workflows, INSTRUCT-QA[5], as well as MedRAG[6], both from previous work on LLM question answering [12, 19]. In our setup, all vectors are stored in main memory without serialization to disk, which enables low-latency access to them. We leverage the FAISS library [20] for efficient approximate nearest neighbor search. For the LLM, we use the open-source LLaMA 3.1 Instruct model [21], which is optimized for instruction-following tasks.

Document source. For the MMLU benchmark, we use the WIKI_DPR dataset as a document source, which contains 21 million passages collected from Wikipedia. The index used is FAISS-HNSW, a graph-based indexing structure optimized for fast and scalable approximate nearest neighbor (ANN) search. For MedRAG, we use PubMed, which contains 23.9M medical publication snippets, and the vector database is served using FAISS-Flat. For both WIKI_DPR and PubMed, we embed each passage as a 768-dimensional vector.

Queries. We construct and evaluate prompts using a subset of the Massive Multitask Language Understanding (MMLU) and PubMedQA datasets. MMLU is a comprehensive benchmark designed to evaluate the knowledge and reasoning of LLMs across a wide array of subjects [11]. MMLU is frequently used to evaluate the effectiveness of RAG, and we leverage the subset of questions on the topic of econometrics, containing 131 total questions. We specifically pick this subset because it highly benefits from RAG. Similarly, we select at random 200 vectors from PubMedQA to serve as user queries.

To simulate similarity, we generate four variants of each question by adding some small textual prefix to them and we randomize the order of the resulting 524 questions for MMLU and 800 for MedRAG.

Hardware. We launch our experiments in Docker containers, using 12 cores of an Intel Xeon Gold 6240 CPU and 300GB of allocated RAM per container.

Metrics. Our evaluation focuses on three performance metrics: *(i)* The *test accuracy* of the entire RAG system, which is computed as the percentage of multiple-choice questions answered correctly by the LLM; *(ii)* The *cache hit rate*, which is defined as the percentage of queries that find a sufficiently similar match in the cache; *(iii)* The *retrieval latency*, which is the time required to retrieve the relevant data chunks, including both cache lookups and vector database queries where necessary.

To ensure statistical robustness, we run each experiment five times and with different random seeds. We average all results and omit standard deviations as they are negligible.

4.3 Results

Our evaluation examines the impact of the cache capacity c and similarity tolerance τ on the three metrics described

[2]https://github.com/rust-lang/portable-simd
[3]https://docs.rs/pyo3/latest/pyo3/.
[4]https://www.maturin.rs.
[5]https://github.com/sacs-epfl/instruct-qa-proximity.
[6]https://github.com/sacs-epfl/medrag-proximity.

above. We evaluate these metrics across different cache capacities $c \in \{10, 50, 100, 200, 300\}$ for both benchmarks. We experiment with tolerance levels $\tau \in \{0, 0.5, 1, 2, 5, 10\}$ for MMLU and $\tau \in \{0, 2, 5, 10\}$ for MedRAG. Figure 3 shows all results.

4.3.1 Accuracy. Figure 3 (left) indicates that accuracy remains relatively stable across different combinations of c and τ, with values ranging between 47.9% and 50.2% for MMLU (top row). Accuracy is slightly higher for low similarity tolerances $\tau = 0$ (no cache) and $\tau = 0.5$: approximately 50.2%. Increasing τ slightly degrades accuracy, bringing it closer to 48.1%. This is because a higher similarity tolerance increases the likelihood of including irrelevant data chunks in the LLM prompt, negatively affecting accuracy. We observe similar behavior in MedRAG (second row), which shows a more pronounced accuracy drop between $\tau = 5$ (88%) and $\tau = 10$ (37%) for the same reason. Increasing c can lower accuracy, *e.g.*, in MMLU for $\tau = 1.0$, accuracy lowers from 50.2% to 49.8% when increasing c from 10 to 300. Interestingly, the highest observed accuracies of 50.3% (for $\tau = 3, c = 10$ in MMLU) and 88.1% (for $\tau = 5, c = 100$ in MedRAG) are achieved with caching. This serendipitously occurs because the approximately retrieved documents prove more helpful on the MMLU benchmark than the closest neighbors retrieved from the database without caching ($\tau = 0$). In both scenarios, the cache rarely decreases accuracy to the level of the LLM without RAG (48% for MMLU, 57% for MedRAG), except when τ is too high (*e.g.*, $\tau = 10$ for MedRAG).

4.3.2 Cache hit rate. Figure 3 (middle) shows the cache hit rate for different values of c and τ for both benchmarks. Increasing τ increases the hit rate. For $\tau = 0$, there are no cache hits, as queries need to be equal to any previous query. However, for $\tau \geq 5$, hit rates reach 93% (MMLU) and 98.4% (MedRAG), demonstrating that higher tolerances allow the cache to serve most queries without contacting the database. In this scenario, the cache prefers to serve possibly irrelevant data rather than contact the database. Nevertheless, even with such high hit rates, there is only a minor decrease in accuracy for MMLU. Similarly, in MedRAG, despite the larger drop in accuracy, a hit rate of 72.6% ($\tau = 5, c = 200$) sustains an accuracy close to the upper bound. Increasing the cache capacity significantly improves the hit rate, *i.e.*, for MMLU, $\tau = 2$, and when increasing c from 10 to 300, the hit rate increase from 6.1% to 69.3%.

4.3.3 Query latency. Figure 3 (right) shows the retrieval latency for different values of c and τ. Latency reductions are significant for configurations with high cache hit rates. For $\tau = 0$ (no cache) and a cache capacity of 10, retrieval latency can be as high as 101 ms (MMLU) and 4.8 s (MedRAG). Retrieval latency quickly decreases as τ increases, which aligns with the increase in hit rate; more queries are now answered with results from the cache. Furthermore, increasing c also decreases retrieval latency, particularly for higher values of τ. Finally, we remark that the speedup gains by Proximity increase as the latency of vector database lookups increases. While our implementation keeps all vector indices in main memory, other database implementations such as DiskANN (partially) store indices on the disk, which increases retrieval latency when not using Proximity further [22]. Thus, such implementations would highly benefit from the speedups enabled by Proximity.

4.3.4 Experimental conclusion. Our findings demonstrate that Proximity effectively reduces retrieval latency while maintaining competitive accuracy. This makes approximate caching a viable optimization for RAG pipelines in scenarios where queries exhibit spatial and temporal similarity. In practical deployments, however, tuning the tolerance parameter and cache capacity based on workload characteristics will be critical to balancing performance and accuracy.

5 Related work

Improving RAG latency. Various strategies have been proposed to decrease the retrieval latency of RAG. Zhu et al. propose Sparse RAG, an approach that encodes retrieved documents in parallel, thereby eliminating delays associated with sequential processing [8]. Sparse RAG reduces overhead of the LLM encoding stage. RAGServe is a system that dynamically adjusts parameters, such as the number of retrieved text chunks, for each query [23]. The system balances response quality and latency by jointly scheduling queries and adapting configurations based on individual query requirements. PipeRAG integrates pipeline parallelism and flexible retrieval intervals to accelerate RAG systems through concurrent retrieval and generation processes [9]. RAGCache is a multilevel dynamic caching system that organizes intermediate states of retrieved knowledge into a hierarchical structure, caching them across GPU and host memory to reduce overhead [24]. TurboRAG reduces the latency of the prefill phase by caching and reusing LLM key-value caches [25]. Cache-Augmented Generation is a method that preloads all relevant documents into a language model's extended context and precomputes key-value caches, thus bypassing real-time retrieval during inference [26]. Speculative RAG improves accuracy and reduces latency by using a smaller LLM to generate multiple drafts in parallel from subsets of retrieved documents [27]. The above systems optimize different aspects of the RAG workflow and many of them are complementary to Proximity.

Similarity caching. Beyond RAG, caching mechanisms have extensively been explored to improve retrieval efficiency in information retrieval systems [28]. Similarity-based caching techniques increase throughput and reduce retrieval latency by exploiting knowledge of the query distribution and content similarity [29, 30]. This has been leveraged in different domains such as image retrieval [31], distributed content networks [32] and recommendation systems [33]. Finally, Regmi et al. propose a semantic cache where user queries are converted to embeddings and cached LLM responses are served for similar user queries [34]. Proximity, on the other hand, focuses on the RAG document retrieval by caching the retrieved documents.

6 Conclusion

We introduced Proximity, a novel caching mechanism designed to enhance the efficiency of retrieval-augmented generation (RAG) systems. Our approach significantly reduces retrieval latency while maintaining retrieval accuracy by leveraging spatial and temporal similarities in user queries to LLMs. Instead of treating each query as an independent event, Proximity caches results from previous queries and reuses them when similar queries appear. This caching reduces the computational load on the underlying vector database and decreases the end-to-end latency of the overall RAG pipeline. Our evaluation with the MMLU and MedRAG benchmarks demonstrates that Proximity provides substantial performance gains in scenarios where users repeatedly query related topics. We conclude that our approximate caching strategy effectively optimizes RAG pipelines, particularly in workloads with similar query patterns.

Acknowledgments

This work has been funded by the Swiss National Science Foundation, under the project "FRIDAY: Frugal, Privacy-Aware and Practical Decentralized Learning", SNSF proposal No. 10.001.796.

References

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

[2] Lexin Zhou, Wout Schellaert, Fernando Martínez-Plumed, Yael Moros-Daval, Cèsar Ferri, and José Hernández-Orallo. Larger and more instructable language models become less reliable. *Nature*, 634(8032):61–68, 2024.

[3] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. Towards mitigating llm hallucination via self reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1827–1843, 2023.

[4] Yoonjoo Lee, Kihoon Son, Tae Soo Kim, Jisu Kim, John Joon Young Chung, Eytan Adar, and Juho Kim. One vs. many: Comprehending accurate information from multiple erroneous and inconsistent ai generations. In *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, pages 2518–2531, 2024.

[5] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[6] Tolga Şakar and Hakan Emekci. Maximizing rag efficiency: A comparative analysis of rag methods. *Natural Language Processing*, 31(1):1–25, 2025.

[7] Michael Shen, Muhammad Umar, Kiwan Maeng, G Edward Suh, and Udit Gupta. Towards understanding systems trade-offs in retrieval-augmented generation model inference. *arXiv preprint arXiv:2412.11854*, 2024.

[8] Yun Zhu, Jia-Chen Gu, Caitlin Sikora, Ho Ko, Yinxiao Liu, Chu-Cheng Lin, Lei Shu, Liangchen Luo, Lei Meng, Bang Liu, et al. Accelerating inference of retrieval-augmented generation via sparse context selection. *arXiv preprint arXiv:2405.16178*, 2024.

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

[10] Ophir Frieder, Ida Mele, Cristina Ioana Muntean, Franco Maria Nardini, Raffaele Perego, and Nicola Tonellotto. Caching historical embeddings in conversational search. *ACM Transactions on the Web*, 18(4):1–19, 2024.

[11] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[12] Guangzhi Xiong, Qiao Jin, Zhiyong Lu, and Aidong Zhang. Benchmarking retrieval-augmented generation for medicine. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics ACL 2024*, pages 6233–6251, Bangkok, Thailand and virtual meeting, August 2024. Association for Computational Linguistics.

[13] Ranul Dayarathne, Uvini Ranaweera, and Upeksha Ganegoda. Comparing the performance of llms in rag-based question-answering: A case study in computer science literature. In *International Conference on Artificial Intelligence in Education Technology*, pages 387–403. Springer, 2024.

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

[15] 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. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022.

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

[17] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

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

[19] Vaibhav Adlakha, Parishad BehnamGhader, Xing Han Lu, Nicholas Meade, and Siva Reddy. Evaluating Correctness and Faithfulness of Instruction-Following Models for Question Answering. *Transactions of the Association for Computational Linguistics*, 12:681–699, 05 2024.

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

[21] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

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

[23] Siddhant Ray, Rui Pan, Zhuohan Gu, Kuntai Du, Ganesh Ananthanarayanan, Ravi Netravali, and Junchen Jiang. Ragserve: Fast quality-aware rag systems with configuration adaptation. *arXiv preprint arXiv:2412.10543*, 2024.

[24] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457*, 2024.

[25] Songshuo Lu, Hua Wang, Yutian Rong, Zhi Chen, and Yaohua Tang. Turborag: Accelerating retrieval-augmented generation with precomputed kv caches for chunked text. *arXiv preprint arXiv:2410.07590*, 2024.

[26] Brian J Chan, Chao-Ting Chen, Jui-Hung Cheng, and Hen-Hsen Huang. Don't do rag: When cache-augmented generation is all you need for knowledge tasks. *arXiv preprint arXiv:2412.15605*, 2024.

[27] Zilong Wang, Zifeng Wang, Long Le, Huaixiu Steven Zheng, Swaroop Mishra, Vincent Perot, Yuwei Zhang, Anush Mattapalli, Ankur Taly, Jingbo Shang, et al. Speculative rag: Enhancing retrieval augmented generation through drafting. *arXiv preprint arXiv:2407.08223*, 2024.

[28] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, 1990.

[29] Flavio Chierichetti, Ravi Kumar, and Sergei Vassilvitskii. Similarity caching. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 127–136, 2009.

[30] Sandeep Pandey, Andrei Broder, Flavio Chierichetti, Vanja Josifovski, Ravi Kumar, and Sergei Vassilvitskii. Nearest-neighbor caching for content-match applications. In *Proceedings of the 18th international conference on World wide web*, pages 441–450, 2009.

[31] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. Similarity caching in large-scale image retrieval. *Information processing & management*, 48(5):803–818, 2012.

[32] Ryo Nakamura and Noriaki Kamiyama. Analysis of similarity caching on general cache networks. *IEEE Access*, 2024.

[33] Pavlos Sermpezis, Theodoros Giannakas, Thrasyvoulos Spyropoulos, and Luigi Vigneri. Soft cache hits: Improving performance through recommendation and delivery of related content. *IEEE Journal on Selected Areas in Communications*, 36(6):1300–1313, 2018.

[34] Sajal Regmi and Chetan Phakami Pun. Gpt semantic cache: Reducing llm costs and latency via semantic embedding caching. *arXiv preprint arXiv:2411.05276*, 2024.