Leveraging Approximate Caching for Faster Retrieval-Augmented Generation

Shai Bergman
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

Ji Zhang
Huawei Research
Zurich, Switzerland

Abstract

Retrieval-augmented generation (RAG) improves 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, substantially reducing the reliance on expensive vector database lookups. To efficiently scale, Proximity employs a locality-sensitive hashing (LSH) scheme that enables fast cache lookups while preserving retrieval accuracy. We evaluate Proximity using the MMLU and MedRAG question-answering benchmarks. Our experiments demonstrate that Proximity with our LSH scheme and a realistically-skewed MedRAG workload reduces database calls by 77.2% while maintaining database recall and test accuracy. We experiment with different similarity tolerances and cache capacities, and show that the time spent within the Proximity cache remains low and constant (4.8 μs) even as the cache grows substantially in size. Our results demonstrate that approximate caching is a practical and effective strategy for optimizing RAG-based systems.

CCS Concepts

• **Information systems → Middleware for databases**; *Language models*; • **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

ACM Reference Format:
Shai Bergman, Anne-Marie Kermarrec, Diana Petrescu, Rafael Pires, Mathis Randl, Martijn de Vos, and Ji Zhang. 2025. Leveraging Approximate Caching for Faster Retrieval-Augmented Generation. In *26th International Middleware Conference (MIDDLEWARE '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721462.3770776

1 Introduction

Large language models (LLMs) have revolutionized natural language processing by demonstrating strong capabilities in tasks such as text generation, translation, and summarization [27]. Despite their increasing adoption, a fundamental challenge is to ensure the reliability of their generated responses [19, 56]. 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 [21] and finance [44]. 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 [29, 56].

Retrieval-augmented generation (RAG) is a popular approach to improve the reliability of LLM answers [30]. RAG combines the strengths of neural network-based text generation with external information retrieval. This technique first retrieves relevant documents from an external database based on the user query and includes them in the LLM prompt before generating a response. Both user queries and documents are often represented as high-dimensional embedding vectors that capture semantic meanings, and these embeddings are stored in a vector database. Retrieving relevant documents involves finding embeddings in the database that are the closest to the query embedding, a process known as nearest neighbor search (NNS). Thus, RAG enables the LLM to use reliable sources of information without the need to modify the model parameters through retraining or fine-tuning [46].

*Corresponding author

At the same time, the NNS operation that is part of the RAG workflow becomes computationally expensive for large vector databases [48, 57]. Thus, RAG can significantly prolong the inference end-to-end time [23, 40]. To mitigate the latency increase of NNS, we observe that user query patterns to conversational agents or search engines often exhibit spatial and temporal locality, where specific topics may experience heightened interest within a short time span [14] or otherwise exhibit strong bias towards some queries [34]. Similar queries are likely to require and benefit from the same set of retrieved documents in such cases, even if they are not exactly syntactically equal. Building on this observation, we reduce the database load by caching and reusing results from similar past user queries. This approach contrasts with conventional RAG systems [42, 57] that treat each query as independent from the others without exploiting access patterns. However, exact embedding matching is ineffective when queries are phrased differently, since even slight rephrasings of a query typically yield different embedding vectors. To this end, we introduce a novel *approximate caching* mechanism that uses the semantic information in the embeddings of queries, computing similarity in the embedding space and incorporating a similarity threshold to address the exact matching problem. 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 by leveraging previously retrieved results for similar queries, Proximity reduces the computational cost of NNS and minimizes database accesses, effectively lowering the total end-to-end inference latency of the RAG pipeline. Specifically, we store past document queries in an approximate key-value cache, where each key corresponds to the embedding of a previous query, and the associated value is the set of relevant documents retrieved for that query. When a new query is received, the cache checks if there is a cache entry within some similarity threshold τ and if so, returns the entry closest to the incoming query. On a cache hit, the corresponding documents are returned, bypassing the need for a database lookup. In case of a cache miss, the system queries the vector database to retrieve relevant documents for the new query. The cache is then updated with the new query and retrieved documents, and the RAG pipeline proceeds as usual. However, to determine whether previously cached queries are sufficiently close to an incoming query, we need to do a linear scan over all cached entries, which becomes computationally expensive as the size of the cache grows. To address this scalability concern, we leverage a locality-sensitive hashing (LSH) scheme and introduce Proximity-LSH, a variant of our approximate cache.

We implement Proximity and evaluate our system using the Massive Multitask Language Understanding (MMLU) [17], and MedRAG [55] benchmarks, which are commonly used to evaluate RAG frameworks. We provide two versions of the MedRAG benchmarks, one where queries are repeated four times each in slight variations, and one where queries are repeated according to a Zipfian distribution with parameter 0.8. Compared to a RAG pipeline without caching, Proximity brings 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 75% for MedRAG with no or only a marginal

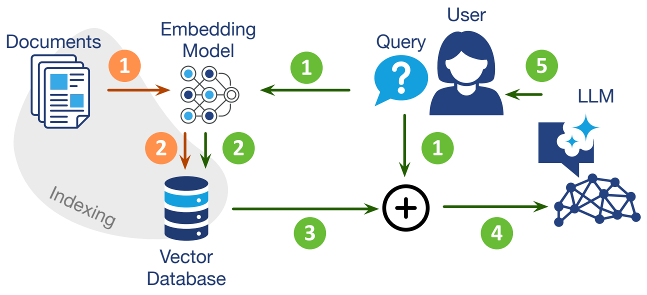

Figure 1: The RAG workflow.

decrease in accuracy. These findings hold across all versions of the experiments, with and without bias towards hot queries. Proximity with our LSH scheme and a realistically skewed MedRAG workload reduces database calls by 77.2% and reduces document retrieval latency by 72.5% while maintaining test accuracy. These results highlight the viability and effectiveness of using approximate caching to improve the speed of RAG-based LLM systems. To the best of our knowledge, Proximity is the first system to apply approximate caching to reduce the document retrieval latency in RAG pipelines during LLMs inference.

Our contributions are as follows:

- We introduce Proximity-FLAT, a novel approximate key-value cache for RAG pipelines that leverages spatial and temporal similarities in user queries to reduce the overhead of document retrieval (Section 3). Proximity-FLAT includes a similarity-based caching strategy that significantly reduces retrieval latency while maintaining high response quality.
- We enhance the scalability of Proximity-FLAT by introducing Proximity-LSH, an efficient and scalable variant that relies on locality-sensitive hashing (LSH) to determine sufficiently similar cache entries (Section 3.2).
- We benchmark both cache variants using two standard datasets, demonstrating substantial improvements in cache hit rates and query latency while maintaining comparable test accuracies (Section 4). These improvements are quantified relative to a baseline RAG pipeline without caching. We also analyze the impact of the cache capacity and similarity tolerance, providing insight into optimizing retrieval performance for workloads with differing characteristics.

2 Background and motivation

We first detail the RAG workflow in Section 2.1 and then outline the process to retrieve the documents relevant to a user query from the vector database in Section 2.2. We then show in Section 2.3 the existence of skew in search engine queries, which provides ground for caching opportunities.

2.1 Retrieval-augmented generation (RAG)

Retrieval-augmented generation (RAG) is a technique that enhances the capabilities of LLMs by integrating information retrieval before the generation process [30]. RAG typically enables higher accuracy in benchmarks with a factual ground truth, such as multiple-choice question answering [9].

Figure 1 shows the RAG workflow that consists of the following eight steps. Before LLM deployment, documents are first converted into high-dimensional embedding vectors using an embedding model ❶ and stored in a vector database ❷. Optionally, documents are divided into smaller chunks before embedding, which improves retrieval accuracy by allowing the system to retrieve the most relevant sections of a document rather than entire documents. These two steps comprise the indexing phase. When the user sends a query to the LLM ❶, this query is first converted to an embedding vector ❷ using the same embedding model as used for the indexing and passed to the retriever. The vector database then searches for embeddings close to the query embedding using some distance metric and returns the relevant documents related to this embedding ❸. These documents and the user query are combined into a single prompt and passed to the LLM ❹. The LLM response is then returned to the user.

2.2 Vector search

The vector search during the RAG workflow 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 [38]. 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 documents 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 [4], performing vector searches at scale poses significant computational challenges. NNS requires comparing query embeddings with millions or billions of stored vectors, which becomes slow and computationally expensive as the database grows [7]. Even with optimized index structures such as as hierarchical navigable small world (HNSW) [33] or quantization-based approaches [26], to maintain low-latency retrieval while ensuring high recall remains difficult.

Several studies show that vector search can account for a significant portion of the end-to-end latency in RAG-based LLM systems [22, 41, 48]. In particular, Shen et al. [48] report that the average Time To First Token (TTFT) increases from 495 ms to 965 ms after deploying RAG, with a significant share of this overhead (71.8%) attributed to the vector database lookup. TTFT captures the latency between sending a query and receiving the first output token from the LLM. The remaining increase comes from a slightly longer LLM pre-fill stage caused by processing the additional retrieved documents. We note that the proportion of TTFT required for vector search depends on factors such as the LLM size, the vector database implementation, and the number of retrieved vectors. Nevertheless, these findings highlight that vector search latency can become a serious bottleneck in RAG systems.

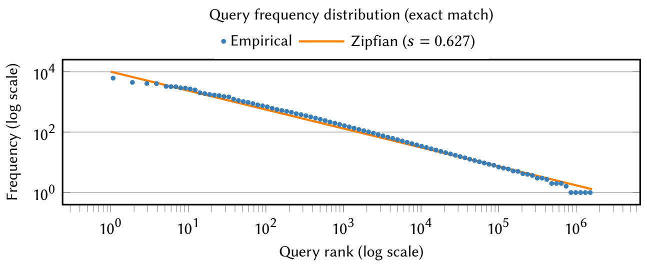

Figure 2: The TripClick empirical query frequencies, along with the expected frequencies from the matching Zipfian distribution.

2.3 Skew in user queries

RAG pipelines often face heavily skewed user query distributions, similar to those found in search engines and conversational agents [15, 18, 50–52]. Specifically, a few popular queries dominate, while most occur rarely or only once. Moreover, LLMs are increasingly being integrated in search engine ecosystems, *e.g.*, ChatGPT Search [37], Google AI Overviews [16] and Microsoft Copilot Search [35]. These systems typically rely on RAG pipelines to ground responses in knowledge sources. In such systems, the documented skew and locality properties of search queries [50, 52] carry over to LLM-based retrieval, making caching particularly appealing.

To quantify this skew, we analyze the TripClick dataset, a large collection of user interactions in a health-focused search engine. TripClick comprises approximately 5.2 million user interactions collected from the Trip Database [34] between 2013 and 2020. The dataset includes around 700 000 unique free-text queries and 1.3 million query-document relevance pairs, making it a valuable resource for studying user search behavior.

Figure 2 shows the exact-match query frequency distribution with the query rank on the horizontal axis and frequency of the query on the vertical axis (both axis in log scale). The empirical distribution of query frequencies closely matches a Zipfian curve with an exponent of $\approx$ 0.627. This power law distribution is not unique to the TripClick dataset. In fact, several studies show that user queries in other domains also often follow a Zipfian power law distribution, with exponent s roughly between s = 0.6 and s = 2.5, resulting in a strong bias towards a few topics of interest [12, 14, 45]. This bias is typical of natural language and brings opportunity for *caching*: the most frequent queries and their semantic variants generate a disproportionate share of retrieval workload and their results may be reused, therefore reducing computational load [3].

Beyond exact query duplicates, many queries differ only slightly in their written form but convey highly similar intent. These include rephrasings, minor spelling variations, synonym substitutions, or changes in the word order (*e.g.*, "best treatment for asthma" vs. "asthma best therapies"). While these variations are not captured by exact matching, they often lie close together in a (latent) embedding space. To analyze this effect, we embed the queries in the TripClick dataset using the MedCPT [25] embedding model and project the resulting vectors (which have a dimension of 768) into two dimensions using a combination of principal component analysis (PCA) [49] as a preprocessing step, followed by t-distributed

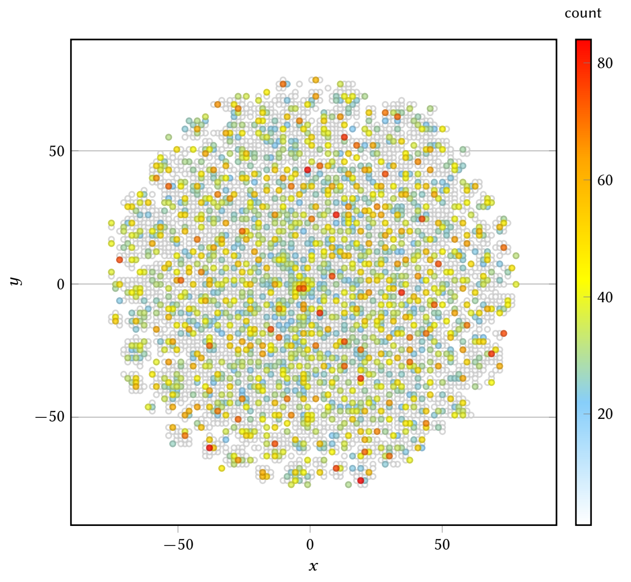

Figure 3: The two-dimensional projection of queries in the TripClick dataset, each one encoded using the MedCPT embedding model.

stochastic neighbor embedding (t-SNE) [53] for visualization purposes. We show the two-dimensional embedding space in Figure 3 as a 100x100 grid. For each point in the grid we count the number of embeddings in it and color it accordingly. This figure highlights that many queries cluster based on semantic content, even when their wording differs.

Thus, we empirically demonstrate using the TripClick dataset that *(i)* user queries frequently repeat exactly, following a power-law distribution; and *(ii)* that syntactically different queries cluster in the embedding space, indicating semantic relationships between distinct queries. These two findings are relevant for RAG systems, which leverage the embedding space. In such systems, each user query is mapped to a high-dimensional vector, and the nearest neighbors in the document index are retrieved based on this vector. When two queries are semantically similar, not only are their embeddings close together, but the resulting retrieved documents often overlap. In other words, different queries may map to the same or similar document sets. *This redundancy means that previously retrieved documents can be cached and reused for future queries with similar embeddings, reducing the number of expensive NNS operations in the vector database.*

3 Design of Proximity

Guided by the above insights, we design a caching mechanism that reduces the need for repeated NNSs by reusing previously retrieved documents for similar queries. More specifically, we leverage *approximate caching* to accelerate RAG document retrieval. Even if the documents 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

Algorithm 1: Cache lookup in Proximity-FLAT

```
Stored state: similarity tolerance $\tau$, cache capacity $c$,
              vector database $\mathcal{D}$, key-value dict $C = \{\}$
1  Procedure LOOKUP($q$):
2      $dists = [(k, \text{DISTANCE}(q, k))$ for $k$ in $C$.keys]
3      $(key, min\_dist) \leftarrow$ min_by_dist($dists$)
4      if $min\_dist \le \tau$ then
5          return $C[key]$
6      $\mathcal{I} \leftarrow \mathcal{D}$.RETRIEVEDOCUMENTINDICES($q$)
7      if $|C| \ge c$ then
8          $C$.EVICTONEENTRY()
9      $C[q] \leftarrow \mathcal{I}$
10     return $\mathcal{I}$
```

reducing retrieval latency. Among others, 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 documents if they are available in the cache. Cache hits thus bypass more expensive vector database searches. However, to determine whether previous cached queries are close to an incoming query, we need to do a linear scan over all cached entries, which becomes computationally expensive as the size of the cache grows. Therefore, the two technical challenges are *(i)* defining an effective set of hyperparameters, such as the similarity threshold that maximizes cache hits without compromising response relevance, as well as the optimal cache size that enables high cache coverage while still being computationally attractive, and *(ii)* ensuring low computational overhead of each query lookup as the number of cached entries grows.

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 retrieveDocumentIndices function that takes as input a query embedding and returns a sorted list of document indices whose embeddings are close to the query embedding. In Section 3.1 we first present the high-level process overview of retrieving vectors with Proximity. Then in Section 3.2 we introduce locality-sensitive hashing (LSH) in the context of approximate caching, one of the key optimizations implemented to reduce the latency of a search in the Proximity cache. Finally, we elaborate in Section 3.3 on the parameterization and components of our caching mechanism.

3.1 Retrieving relevant documents with Proximity-FLAT

We first present the basic version of our cache, named Proximity-FLAT, and later describe an optimized version of our cache in Section 3.2. Algorithm 1 shows the high-level algorithm for retrieving relevant documents with Proximity-FLAT. We also visualize the Proximity cache and workflow in Figure 4 with an example when processing 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,

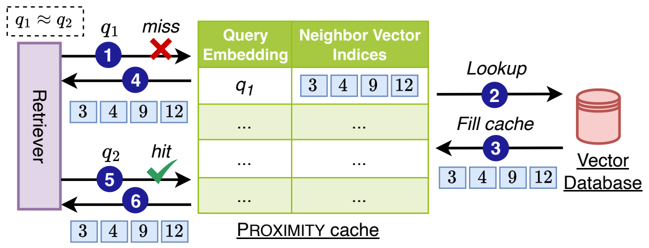

Figure 4: The design and workflow of the Proximity approximate cache when receiving two subsequent, similar query embeddings q_1 and q_2. q_1 results in a cache miss whereas q_2 results in a hit, returning similar document indices as for q_1.

where k is a strictly positive integer constant picked at the start of the experiment. The cache has a fixed capacity of c key entries, which implies that, when full, an eviction policy is applied to make room for new entries (see Section 3.3.2).

When a user initiates a query, it is first converted into an embedding vector by an embedding model. Proximity is agnostic of the specific embedding model used but we assume that the embedding of the user query and the embeddings in the vector database are generated by the same embedding model, since embeddings from different models are not directly comparable. The retriever (left in Figure 4) then forwards this query embedding, denoted as q_1, to the Proximity cache ❶. Proximity first checks whether a similar query has been recently processed by iterating over each key-value pair (k, v) of cache entries (line 2 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 3-5), thus bypassing the vector database. Otherwise, the system proceeds with a standard database query (line 6 in Algorithm 1). This is step ❷ in Figure 4, 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 ❸. Since the number of cache entries might exceed the cache size, the eviction policy will remove a cache entry if necessary (lines 7-8). The cache is now updated with the result obtained from the database (line 9). Finally, the vector indices are returned to the retriever ❹. We adopt the same distance function in Proximity as the underlying vector database to ensure consistency between the caching mechanism and the retrieval process.

When another query embedding q_2 arrives ❺ 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 distance between q_1 and q_2 is below the predefined similarity threshold τ. In that case, the cache returns the previously retrieved document indices associated with the closest matching query ❻, thus bypassing a lookup in the vector database. We name this first approach Proximity-FLAT as it scans the entire cache for every incoming query (line 2) without using any data structure to guide it further. Depending on the specifications of the cache and workload, Proximity-FLAT can reduce retrieval latency and computational overhead, especially in workloads with strong spatial or temporal similarities.

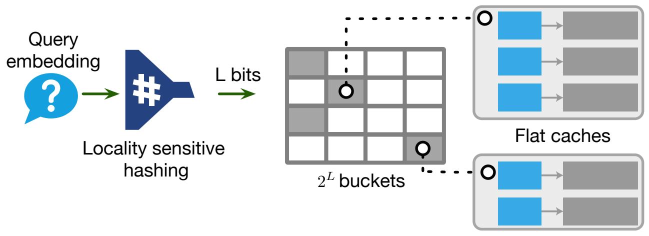

Figure 5: The workflow of Proximity-LSH where each LSH bucket maps to a Proximity-FLAT cache.

3.2 Scalability and Proximity-LSH

While Proximity-FLAT provides caching benefits with minimal complexity, its linear scan over all cached entries is a scalability bottleneck. As the cache size increases, so does the computational cost of each query lookup, eventually increasing the end-to-end inference time. This linear scan ensures that we find the closest match in the cache but also results in a lookup cost that is linearly dependent on c, the cache capacity. For large caches, this cost becomes prohibitive.

To mitigate this scalability bottleneck, we introduce Proximity-LSH, a variant of our approximate cache that supports scalable similarity-based lookups using LSH, to enable the query to be redirected to a small bucket of entries that are the most likely to match it. This bounds the amount of query-key comparisons to be performed to the size of the bucket. More specifically, we rely on random hyperplane LSH [6], a classical method for approximate nearest neighbor (ANN) search in high-dimensional spaces. The key idea is to compare each embedding to a fixed set of L randomly generated hyperplanes passing through the origin. Each hyperplane defines a binary partition of space: a vector lies on one side or the other. By repeating this for L hyperplanes, we obtain an L-bit signature (a binary hash code) that serves as the key for the LSH table. Formally, given an embedding vector $q \in \mathbb{R}^d$, and L random hyperplanes (stored as normal vectors $r_1, r_2, \ldots, r_L$), we compute its hash code $h(q)$ as:

$$h(q) : \{0,1\}^L = (q \cdot r_1 \geq 0, ..., q \cdot r_L \geq 0)$$

In Proximity-LSH, the hash code $h(q)$ determines the bucket to which an incoming query maps. We show the workflow of Proximity-LSH in Figure 5. Each bucket is a fixed-size Proximity-FLAT cache with a capacity of $b = 20$ entries, which we empirically find to strike a good balance between the hit rate and the execution latency per query (see Section 4.3.5). When a query arrives, we compute its LSH hash, identify its corresponding bucket, and perform a linear scan within only that bucket to check for similar embeddings. If no similar entry is found, the query is forwarded to the vector database, and the bucket is updated accordingly. Unlike when using a single flat cache, each bucket in Proximity-LSH operates its own local eviction policy such as first-in, first-out (FIFO) or least-recently used (LRU). In particular, there is no global eviction policy that is shared across Proximity-LSH caches as this would require additional data structures and state storage, resulting in additional compute and memory costs. Note that this makes PROXIMITY-LSH a b-way set-associative cache, where the set corresponding to a given entry is designated by L-bit LSH. We experimentally show in Section 4 that LSH-based partitioning sustains high cache hit rates and accuracy.

Another benefit of using such buckets is the mitigation of false positives, *i.e.*, cases where the similarity tolerance is large enough to consider a query and a cache line as matching, even though the two embeddings are actually semantically very different. This mitigation arises because the tolerance only applies within the bucket to which the query maps, limiting potential false positives to that subset. Due to the LSH property of grouping similar embeddings, cache lines in this bucket are more likely to be relevant. Moreover, if a query is very infrequent, its corresponding bucket may be empty. In such a case, false positives cannot occur, even though other buckets contain entries. This contrasts with PROXIMITY-FLAT, where any cache line can potentially be a false positive depending on the global similarity threshold.

Performance gains compared to PROXIMITY-FLAT. PROXIMITY-LSH yields a significant performance boost when dealing with caches with many entries. In PROXIMITY-FLAT, every query is compared against all c cached embeddings using the chosen distance metric, resulting in a per-query lookup cost of $O(c \cdot d)$, where d is the embedding dimensionality. This linear dependency on the cache capacity becomes noticeable in large caches, especially when c reaches into the thousands, as discussed in Section 4.5.1.

In contrast, PROXIMITY-LSH limits comparisons to a single fixed-size bucket. Each query is hashed using L random hyperplanes, at a cost of $O(L \cdot d)$, and linearly compared only to the b entries (*e.g.*, $b = 20$) in the selected bucket, with a comparison cost of $O(b \cdot d)$ corresponding to the distance computations for each of the b cache lines. Both L and b are (small) constants, making the total cost of a lookup $O(d)$. Most importantly, it is independent of the total cache capacity.

As an example, for $c = 10\,000$ entries, $d = 768$, $b = 20$, and $L = 10$, a PROXIMITY-FLAT lookup performs $10\,000 \times 768 = 7.68$ million operations per query. In contrast, PROXIMITY-LSH performs only $(L + b) \cdot d = 30 \times 768 = 23\,040$ operations, a reduction of over 300× in per-query compute. This performance gap widens with increasing cache size. We show in Section 4 that this speedup does not come at the cost of the quality of the retrieved documents.

3.3 PROXIMITY cache parameters and components

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

3.3.1 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 4). 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 documents 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. For a small enough c, this computational overhead is manageable since the cache size is likely to be small compared to the full vector database.

Effective capacity in PROXIMITY-LSH. Unlike PROXIMITY-FLAT, where the cache capacity c is a flat global limit on the number of cached embeddings, PROXIMITY-LSH distributes this capacity across multiple buckets, each with a fixed maximum size b (*e.g.*, 20 entries). The total cache capacity of PROXIMITY-LSH is thus $c = 2^L \cdot b$, where b is the per-bucket size and L the number of random hyperplanes.

However, because LSH partitions the embedding space based on the distribution of query vectors, in practice not all buckets receive traffic. Some buckets remain unused, especially in workloads with skewed or clustered query distributions. As a result, the actual number of stored entries is often much smaller than the theoretical maximum. This sparsity leads to more efficient space usage: we do not allocate memory for a given bucket until an entry is inserted. PROXIMITY-LSH allocates space only where needed, adapting naturally to the query distribution. Our experiments in Section 4 show that with PROXIMITY-LSH only a fraction of the buckets are actively populated, and the memory footprint scales more gracefully with usage compared to flat caches of the same theoretical size.

3.3.2 Eviction policy. When the cache reaches its maximum capacity, an eviction policy is required to determine which entry should be removed to make space for new ones. The choice of policy can influence the hit rate of the cache, especially in workloads with strong temporal locality or repeated query patterns. In PROXIMITY-FLAT, we support both FIFO and LRU eviction strategies. FIFO evicts the oldest inserted entry regardless of usage frequency. It is straightforward to implement, incurs minimal overhead, and in our experiments, provides comparable accuracy to more sophisticated strategies in many settings. However, FIFO can underperform when there is strong temporal locality, *i.e.*, when recent queries are more likely to be repeated. In such cases, LRU can yield higher hit rates by preferentially retaining recently accessed entries. LRU maintains a usage timestamp or recency ordering, evicting the entry that has gone unused the longest. While slightly more expensive to manage, we find that LRU remains efficient for cache sizes considered in PROXIMITY-FLAT and is particularly effective under bursty query traffic. In PROXIMITY-LSH, eviction is managed separately within each individual bucket.

3.3.3 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 an incoming query. The choice of τ directly impacts recall and test accuracy, and is therefore a key parameter of PROXIMITY. A low value of τ enforces stricter matching, ensuring that retrieved documents are highly relevant but potentially reducing the cache hit rate, thus 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 documents. In our experiments, τ is treated as a global constant, manually set at the start of each evaluation. In general, determining the appropriate tolerance for high-dimensional approximate matching is a challenge that has no standard solution in the literature we reviewed. Most notably, Frieder *et al.* [14] propose reusing the neighbor information to generate a dynamic tolerance per cache line. We found that this still required some arbitrary hand-tuning. In this paper, we leave τ as a hyperparameter to be optimized during deployment. We explore the influence of this parameter on the cache performance in Section 4.

3.3.4 The re-ranking factor ρ. It is common practice for ANN vector databases such as DiskANN [20] and FAISS [10] to retrieve more neighbors than they actually return to the retriever [32]. This over-fetching improves recall at the cost of increased latency. We adopt a similar strategy in Proximity. Specifically, when querying the vector database (line 6 in Algorithm 1), we request a number of neighbors significantly exceeding the count required by the retriever. This does not incur additional latency since the database inherently performs this broader search. Upon a subsequent cache hit (line 5 in Algorithm 1), rather than immediately returning all associated vectors, we re-rank these vectors to prioritize and select only the ones that are most relevant to the current query. We define the re-ranking factor $\rho \geq 1$ as the ratio between the number of vectors retrieved from the database and the number of vectors expected by the RAG pipeline.

4 Evaluation

We implement both Proximity-FLAT and Proximity-LSH, and evaluate our approximate caching approach using two standard benchmarks: Massive Multitask Language Understanding (MMLU) and MedRAG. Specifically, our experiments answer the following three questions:

(i) What is the impact of cache parameters such as cache capacity c, similarity tolerance τ, LSH hashing granularity L, and per-bucket capacity b on the end-to-end test accuracy, k-recall, hit rate, and latency of Proximity-FLAT and Proximity-LSH (Section 4.3)?

(ii) How does the cache occupancy of Proximity-LSH change when varying the LSH hashing granularity L and similarity tolerance τ (Section 4.4)?

(iii) How does the lookup time of Proximity vary with cache occupancy and parameters such as cache capacity c, similarity tolerance τ, and LSH hashing granularity L? Furthermore, how well does Proximity maintain hit rate and recall on large-scale datasets? (Section 4.5)

4.1 Implementation details

We implement Proximity in the Rust programming language and make its implementation available online.[1] We use SIMD CPU instructions for all numerical computations (*e.g.*, the Euclidean distance compuation at line 2 of algorithm 1). Our implementation uses portable-simd[2], an experimental extension to the Rust language that enables ISA-generic explicit SIMD operations. While all LLVM target architectures are technically supported, we only considered modern x64 and ARM64 platforms for unit testing and performance evaluation.

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 Proximity into existing RAG pipelines.

4.2 Experimental setup

To evaluate Proximity, we adopt and modify two existing end-to-end RAG workflows, MMLU and MedRAG, both introduced by previous work on LLM question answering [1, 55]. 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 [10] for efficient ANN search. For the LLM, we use the open-source LLaMA 3.1 Instruct model [11], which is optimized for instruction-following tasks.

4.2.1 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 ANN search. For MedRAG, we use PubMed as the document source, which contains 23.9 million medical publication snippets. The associated vector database is served using FAISS-Flat. For both wiki_dpr and PubMed, we embed each passage as a 768-dimensional vector. We use the MedCPT [25] and the DPR [28] embedding models for the MedRAG and MMLU benchmarks, respectively.

4.2.2 Query workload. We evaluate Proximity by using a subset of the MMLU and PubMedQA question datasets.

MMLU is a comprehensive benchmark designed to evaluate the knowledge and reasoning of LLMs across a wide array of subjects [17]. It encompasses approximately 16 000 multiple-choice questions ranging between 57 topics, *e.g.*, mathematics and history, with question difficulties ranging from elementary to advanced professional levels. 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.

MedRAG is a benchmark for question answering in the biomedical domain, and we specifically focus on the PubMedQA question set that contains 500 questions and answers. Similarly to MMLU, we select at random 200 questions 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. These two datasets help us to showcase the performance of our cache implementation in the case where there is no strong bias in the queries used: every query appears four times in a slightly different format each time. We denote these two datasets as the *uniform* datasets, as they do not display bias in the queries.

MedRAG-Zipf. Finally, to showcase the capabilities of Proximity in the context of strong, real-life-like bias, we create a synthetic third dataset based on PubMedQA by drawing 10 000 queries from

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

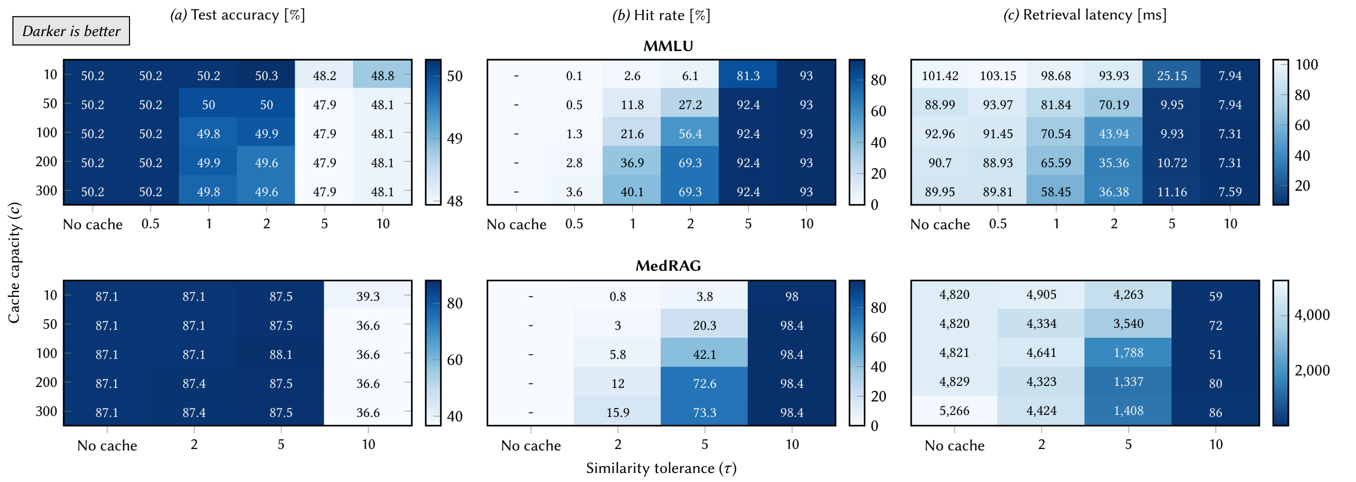

Figure 6: The test accuracy (a), cache hit rate (b), and latency of document retrieval (c) of PROXIMITY-FLAT, for different cache capacities and similarity tolerances, for the MMLU (top) and MEDRAG (bottom) benchmarks.

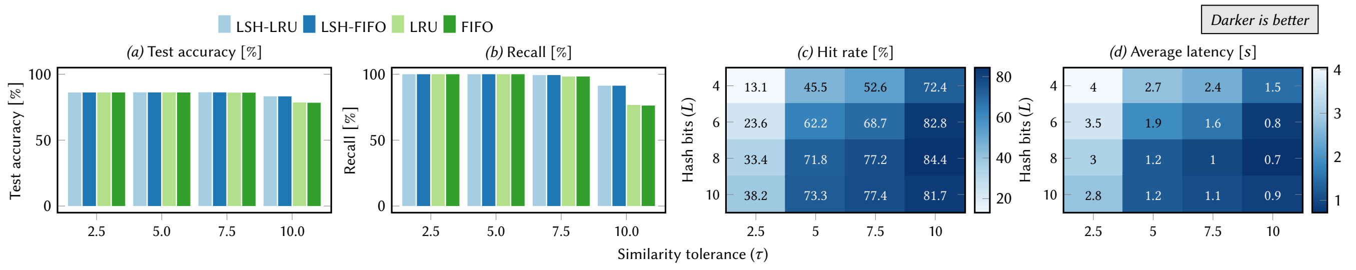

Figure 7: The test accuracy (a), recall (b), hit rate (c) and average latency (d), for different similarity tolerances, eviction policies, with and without LSH, under the MEDRAG-ZIPF benchmark.

the original set of 500 PUBMEDQA questions with repetition, following a Zipf distribution with parameter 0.8, which we estimate to be a reasonable representation of real-world skews, as discussed in Section 2.3. Each time a given query appears, we rephrase it by asking an LLM[5] to rephrase the question in a way that is syntactically different but semantically equivalent to the original. We verify that each generated version of the question is unique across the entire dataset, and that the RAG generative system indeed provides the same answer across all rephrasings when run without PROXIMITY. The most frequently repeated question appears about 700 times in this dataset while the vast majority of original questions appear at most a few times. Every original question appears at least once. This simulates several users asking the same question with various wordings, in which case PROXIMITY should ideally only perform a single retrieval from the vector database, and rely on the cache for further queries. As a worst-case scenario for caching, we do not simulate temporal locality of queries: all queries are statistically independent from each other, even though their distribution is severely biased towards some of the original PUBMEDQA questions. We refer to this third dataset as the MEDRAG-ZIPF dataset.

[5]gpt4o-2024-08-06 by OpenAI

4.2.3 Hardware. We launch our experiments in Docker containers, using 12 cores of an x64 Intel Xeon Gold 6240 CPU and 300 GB of allocated RAM per container.

4.2.4 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 in our query workloads 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 documents, including both cache lookups and vector database queries where necessary. To quantify how well the cache preserves the quality of retrieved context, we also measure the *database k-recall*, defined as the fraction of the top-k documents returned by the cache that are also among the top-k results retrieved from the vector database for the same query. This metric allows us to assess the extent to which cached results align with the true nearest neighbors, serving as a proxy for retrieval quality. While the ultimate goal is to maintain end-to-end accuracy in the RAG pipeline, k-recall provides a more direct and inexpensive way to evaluate the effectiveness of approximate caching, independent of downstream LLM variability. To ensure statistical robustness, we run each experiment five times and with different random seeds. We average all results.

4.3 The impact of cache parameters on test accuracy, recall, hit rate, and latency

We first examine the impact of the cache capacity c and similarity tolerance τ on the three metrics described above. We evaluate these metrics across different cache capacities $c \in \{10, 50, 100, 200, 300\}$ for both uniform benchmarks using the FIFO cache eviction policy. 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. We perform no re-ranking ($\rho = 1$) for MMLU and MedRAG. For the MedRAG-Zipf dataset, we provide the accuracy and database recall for various cache eviction policies, with and without LSH (see Section 3.2). We also report the hit rate and average latency, as in the uniform datasets. We use a re-ranking factor of $\rho = 4$ in experiments on MedRAG-Zipf. Figure 6 shows the results on all uniform datasets and Figure 7 shows the results for MedRAG-Zipf.

4.3.1 *Test accuracy.*

Figure 6(a) shows the end-to-end RAG accuracy for the MMLU and MedRAG benchmarks for different combinations of c and τ and with Proximity-FLAT. The figure 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). Test 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 documents in the LLM prompt, negatively affecting accuracy. We observe similar behavior in MedRAG (bottom 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 the value of τ becomes too high (*e.g.*, $\tau = 10$ for MedRAG).

Figure 7(a) shows the test accuracy when using the MedRAG-Zipf benchmark, for different similarity tolerances, eviction policies, and with and without LSH. For $\tau \in \{2.5, 5, 7.5\}$, we observe comparable accuracies across the evaluated eviction policies and between Proximity-FLAT and Proximity-LSH. For $\tau = 10$, we observe a notable degradation in accuracy for Proximity-FLAT: from 85.7% for LRU with $\tau = 2.5$ to 77.4% for LRU with $\tau = 10$. This same degradation is much less pronounced for Proximity-LSH: from 85.8% for LSH-LRU with $\tau = 2.5$ to 84.1% for LSH-LRU with $\tau = 10$. These results show that Proximity-LSH is robust against higher values of τ.

4.3.2 *Recall.*

Figure 7(b) displays the k-recall for different similarity tolerances, eviction policies, and with and without LSH, and when using the MedRAG-Zipf benchmark. This recall indicates the overlap between document indices returned by the cache and the document indices that the database *would have returned* in the absence of caching. We observe that the k-recall is virtually 100% for tolerances below 7.5, showing that the cache is perfectly transparent in this regime: the retrieved documents by the cache are exactly the same as the ones that the database would have selected. For $\tau = 10$, we observe a degradation in the k-recall, which translates to a similar degradation in the end-to-end accuracy of the model (see Figure 7 (a)). This is consistent across all four cache configurations. We note that this correlation between accuracy and k-recall can be leveraged to empirically fine-tune the tolerance hyperparameter τ at a low cost, as k-recall can be evaluated without LLM inference, which is compute-intensive. Once the optimal tolerance has been found for k-recall, one can extrapolate that it is also optimal for end-to-end LLM accuracy.

4.3.3 *Cache hit rate.*

Figure 6(b) shows the cache hit rate for different values of c and τ for both benchmarks and with Proximity-FLAT. 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% for MMLU and 98.4% for 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%.

Figure 7(c) further illustrates the cache hit rate as a function of the similarity tolerance τ and the number of LSH hash bits L, for Proximity-LSH and with the skewed MedRAG-Zipf workload. For this experiments we use the LRU eviction policy. We observe that hit rates increase with τ, confirming the trends seen Figure 6(b). Importantly, even with small hash sizes in Proximity-LSH (*e.g.*, 4–6 bits, corresponding to 16 and 64 buckets, respectively), hit rates remain high for tolerances above 5.0, validating the effectiveness of LSH in grouping semantically similar queries.

4.3.4 *Query latency.*

Figure 6(c) shows the retrieval latency for different values of c and τ, for Proximity-FLAT. Latency reductions are significant for configurations with high cache hit rates. For $\tau = 0$ (no cache) and $c = 10$, retrieval latency can be as high as 101 ms for MMLU and 4.8 s for 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 further increases retrieval latency [20].

Figure 7(d) shows the average retrieval latency for Proximity-LSH with the LRU eviction policy. Latency drops sharply as the similarity tolerance increases and the cache hit rate improves. This

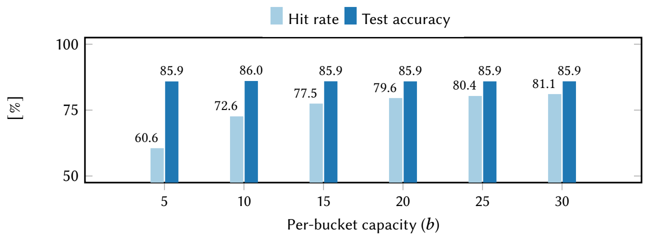

Figure 8: The test accuracy and hit rate for Proximity-LSH as a function of the flat bucket size.

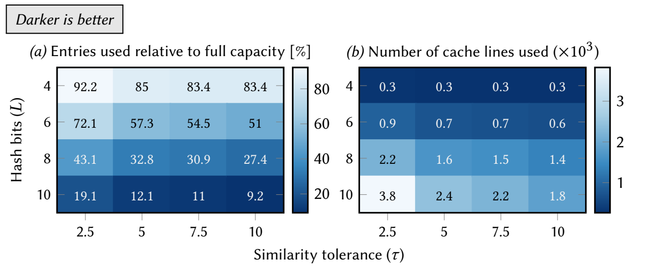

Figure 9: The relative (a) and absolute (b) cache occupancy after completion of the MedRAG-Zipf workload when using the Proximity-LSH cache with a LRU eviction policy, for varying LSH hashing granularities and similarity tolerances.

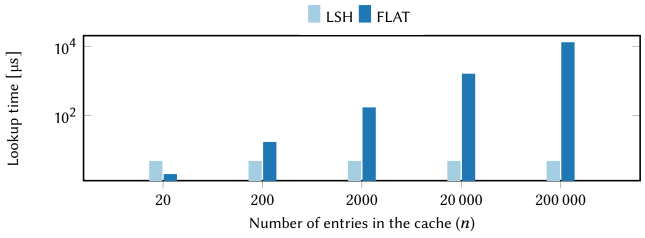

Figure 10: The cache lookup time as the number of cache entries increase, for Proximity-FLAT (requiring a linear scan) and Proximity-LSH. For both cache types we use the LRU eviction policy.

improvement in average latency is again linearly increasing with the hit rate, as cache hits dismiss database calls entirely and cache misses are virtually zero-cost when compared to vector database calls.

4.3.5 Bucket size. We now study the impact of the per-bucket capacity b in Proximity-LSH on the hit rate and test accuracy. Intuitively, larger bucket capacities increase the chance of finding a match among cached entries, thus improving the hit rate. However, they also increase the number of entries to be scanned per lookup, resulting in higher per-query latency.

We evaluate this trade-off with the MedRAG-Zipf dataset, using Proximity-LSH with $L = 8$ hyperplanes, a tolerance parameter $\tau = 7.5$ and the LRU eviction policy. Figure 8 shows the test accuracy and hit rate for different bucket sizes b. Our experimental results show that $b = 20$ offers the best balance between hit rate and lookup cost. Increasing b from 5 to 20 substantially improves the hit rate from 60.6% to 79.6%, while the accuracy remains stable at approximately 85.9%. For $b > 20$, the hit rate quickly reaches a plateau (81.1% at $b = 30$), while accuracy remains unchanged. Thus, larger bucket capacities yield negligible benefits in terms of hit rate or accuracy, yet impose higher scanning overhead. We therefore fix $b = 20$ in our experiments with Proximity-LSH.

4.4 Proximity-LSH cache occupancy

We now determine the cache occupancy of Proximity-LSH, under varying similarity tolerances τ and hashing granularity, expressed as the number of hash bits L, after the MedRAG-Zipf workload has completed. We note that in this experiment, the eviction policy (LRU or FIFO) is not relevant, as they both share the same memory footprint. We show these results in Figure 9. Figure 9(a) shows *relative* usage, computed as the fraction of the theoretical maximum capacity, while Figure 9(b) reports the *absolute* number of vectors stored in the cache. The maximum capacity of Proximity-LSH is $2^L \times 20$, where L is the number of hash bits and 20 is the per-bucket limit.

Across all configurations, we observe a consistent trend: increasing the number of hash bits L significantly reduces relative cache utilization. For instance, with $L = 4$ (*i.e.*, a total of 16 buckets), the cache achieves over 92% utilization with $\tau = 2.5$, filling 295 of the 320 entries. In contrast, for $L = 10$ (1024 buckets), only 19.1% of the cache is used at the same tolerance level, corresponding to 3920 occupied entries out of a maximum of 20 480. This reflects the sparsity inherent to LSH-based caching: many buckets remain unused unless queries are uniformly distributed across the embedding space, which is not the case for our skewed workload.

Increasing the tolerance τ slightly *reduces* both absolute and relative cache occupancy: for all configurations of L, cache occupancy declines as τ increases from 2.5 to 10.0. This is due to the increased hit rate at higher tolerances: more queries are matched to existing entries, thereby reducing the number of insertions required. For example, at $L = 8$, usage decreases from 2209 entries (43.1%) at $\tau = 2.5$ to 1402 entries (27.4%) at $\tau = 10.0$.

We argue that this adaptive sparsity is a desirable property of Proximity-LSH. Unlike Proximity-FLAT that gradually fills up to its maximum size, LSH-based caching only allocates memory where needed. In practical deployments with skewed or clustered query distributions, this leads to significantly lower memory usage while preserving high cache effectiveness. Consequently, Proximity-LSH can be provisioned with a large theoretical capacity for collision avoidance, without incurring the cost of fully allocating all cache slots.

4.5 Proximity scalability

Finally, we evaluate the scalability of Proximity by analyzing the lookup times for different cache occupancies, and when varying cache parameters. We also provide experimental insights on deploying Proximity on larger query sets by analyzing the recall and hit rate on the TripClick dataset.

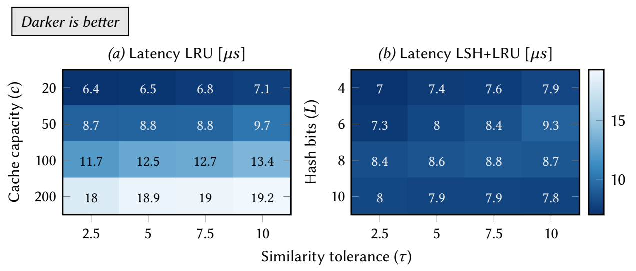

Figure 11: The cache lookup time of MedRAG-Zipf queries sent to Proximity-FLAT (a) and Proximity-LSH (b) for varying cache capacities and similarity tolerances.

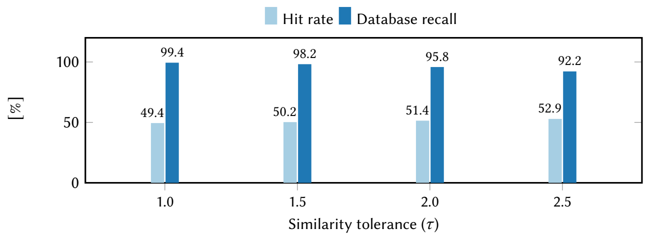

Figure 12: Hit rate and database k-recall of Proximity-LSH on the TripClick dataset and with the LRU eviction policy, while varying the similarity tolerance τ.

4.5.1 Lookup times for increasing cache occupancies. We now evaluate the cache lookup times by Proximity-FLAT and Proximity-LSH by measuring the time spent per query as the cache size increases, under the LRU eviction policy. We vary the number of entries that are stored in the cache when the query arrives, denoted by n and visualize the results in Figure 10. Figure 10 (dark blue bars) shows the per-query latency of the Proximity-FLAT implementation. The lookup time increases nearly linearly with the cache capacity c (this holds regardless of the similarity tolerance τ, as the amount of computation to be done is independent of τ). Specifically, this increase is from 2.0 µs for $n = 20$ to 13.0 ms for $n = 200\,000$. This is consistent with the algorithmic complexity of Proximity-FLAT, which performs a full scan of the cache and computes the distance between the incoming query and every stored key. Even at reasonable cache sizes, this linear scan incurs an overhead that can become a noticeable proportion of the retrieval time budget. In contrast, Figure 10 (light blue bars) displays the scaling capabilities of Proximity-LSH in terms of the amount of vectors that are stored in the cache when the user query arrives at the cache. We use $L = 8$ hashing planes. The lookup time remains constant as n increases: for all evaluated values of n we observe a lookup time of merely 4.8 µs. This highlights the excellent scalability of the Proximity-LSH cache.

4.5.2 Lookup times for varying cache parameters. We run Proximity with the MedRAG-Zipf benchmark and analyze the cache lookup times of queries sent to both Proximity-FLAT and Proximity-LSH ($L = 8$), for varying cache capacities and similarity tolerances (Figure 11). Unlike the measurements in Figure 7(d), which capture the full end-to-end retrieval time including database access, for this experiment we only consider the time spent performing cache lookups. Figure 11(a) shows these results for Proximity-FLAT and highlights that increasing c while fixing τ increases the cache lookup time. This is because there are more cache entries and thus more computational requirements to complete the lookup. A similar trend is visible when increasing τ when keeping c fixed. This is because for increasing values of τ we are left with more candidates after the linear scan completes (line 2 in Algorithm 1) and therefore require more time in determining the best candidate (line 3 in Algorithm 1). Figure 11(b) shows the latency of Proximity-LSH for varying tolerance and LSH hashing granularity. Here, the lookup time remains relatively stable across hashing granularity and similarity tolerances. The time per query depends only on the per-bucket capacity, and not on the total number of cached entries. This confirms our theoretical expectation that LSH-based cache lookups have constant-time complexity with respect to both total cache size and cache tolerances.

4.5.3 Large query dataset. To demonstrate the effectiveness of Proximity on a large-scale query dataset, we use TripClick as the query set (see Section 2.3), PubMed as the document database, and DiskANN as the vector database. TripClick contains approximately 5.2 million user queries and we send queries to the system in the same order as they appear in the original dataset. We evaluate both the cache hit rate and the database recall by comparing the vectors returned by Proximity-LSH ($L = 8$) against those retrieved directly from the database for the same queries. Figure 12 shows how the cache hit rate and database recall behave for different values of the similarity tolerance τ when using the LRU eviction policy. Proximity-LSH maintains a stable cache hit rate around 50% across all similarity tolerance levels τ. For $\tau = 1.0$ and $\tau = 1.5$, the cache effectively reduces the number of retrieval calls by approximately 50% while preserving near-perfect k-recall. However, k-recall decreases as τ increases, *e.g.*, from 99.4% for $\tau = 1.0$ to 92.2% for $\tau = 2.5$. This is because a higher similarity tolerance permits the cache to match a larger number of cache lines. These findings further highlight that Proximity can achieve significant latency reductions while maintaining high retrieval accuracy even when using workloads with a large number of queries and with real-world spatial and temporal locality.

4.6 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 and cache capacity hyperparameters 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 reducing delays associated with sequential processing [57].

Sparse RAG reduces overhead of the LLM encoding stage. RAGServe is a system that dynamically adjusts parameters, such as the number of retrieved documents, for each query [42]. 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 [23]. 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 [31]. 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 [5]. Speculative RAG improves accuracy and reduces latency by using a smaller LLM to generate multiple drafts in parallel from subsets of retrieved documents [54]. 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 long been studied to improve efficiency in information retrieval systems [2]. In particular, similarity-based caching techniques aim to reduce retrieval latency and improve system throughput by exploiting patterns in query semantics and distribution [8, 39]. These approaches typically leverage the observation that similar queries tend to retrieve similar results, enabling cache reuse even in the absence of exact query matches.

Such techniques have found applications across a variety of domains, including approximate image retrieval [13], content distribution networks [36], and recommendation systems [47]. In these contexts, approximate caching is often implemented using distance metrics or clustering methods in embedding spaces to identify and reuse semantically close queries.

However, to the best of our knowledge, these approaches have not been applied in the context of question answering, where approximate nearest neighbor search (NNS) is used not merely to retrieve similar content, but to support factual or task-oriented responses via downstream LLM inference. This makes correctness and latency trade-offs more sensitive and requires a finer-grained control over recall and relevance.

In this work, we extend approximate caching to this domain by introducing Proximity, a system designed specifically for low-latency, large-scale RAG. Our design incorporates key scalability optimizations, including locality-sensitive hashing to reduce lookup complexity and explicit SIMD instructions to accelerate distance computations. These architectural decisions enable our system to operate efficiently under realistic workloads and tight latency budgets, making it, to our knowledge, the first practical application of approximate similarity caching in RAG-based question answering pipelines.

Answer caching. Finally, Regmi et al. [43] propose a semantic caching strategy tailored for LLM applications, where responses to previous user queries are stored and reused when semantically similar queries are detected. In their system, incoming user queries are embedded into a high-dimensional space, and the cache returns a previously generated LLM response if a sufficiently similar query has already been seen. Their approach bypasses both document retrieval and answer generation entirely on cache hits, leading to substantial latency reductions.

While effective in reducing end-to-end response time, this technique assumes that semantically close queries always require identical responses, which may not hold in practice. This is especially the case when subtle distinctions between queries are relevant. In contrast, Proximity focuses on optimizing the RAG document retrieval phase by caching references to retrieved documents rather than final answers. This allows the LLM to regenerate responses tailored to the precise query phrasing, preserving potentially important nuances that may be lost at the embedding level. As a result, our method provides greater flexibility and better preserves fidelity to the user's intent, while still reducing the computational overhead of expensive nearest-neighbor lookups.

6 Conclusion

We introduced Proximity, a novel caching mechanism designed to enhance the efficiency of 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. We also improve the scalability of query lookup times by designing Proximity-LSH, a cache that uses locality-sensitive hashing (LSH) to dramatically speed up the process of determining similar queries in the cache. Our evaluation with the MMLU and MedRAG benchmarks demonstrates that both Proximity-FLAT and Proximity-LSH provide 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 grant number 10001796. We are also grateful to Michał Stawarz for helpful discussions and for his recommendations regarding relevant literature.

References

[1] Vaibhav Adlakha, Parishad BehnamGhader, Xing Han Lu, Nicholas Meade, and Siva Reddy. 2024. Evaluating Correctness and Faithfulness of Instruction-Following Models for Question Answering. *Transactions of the Association for Computational Linguistics* 12 (05 2024). https://doi.org/10.1162/tacl_a_00667

[2] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. 1990. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)* 15, 3 (1990). https://doi.org/10.1145/88636.87848

[3] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. 2007. The impact of caching on search engines. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (Amsterdam, The Netherlands) *(SIGIR '07)*. https://doi.org/10.1145/1277741.1277775

[4] 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. In *International conference on machine learning (ICML '22)*. PMLR. https://proceedings.mlr.press/v162/borgeaud22a/borgeaud22a.pdf

[5] Brian J Chan, Chao-Ting Chen, Jui-Hung Cheng, and Hen-Hsen 25Huang. 2024. Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks. (2024). arXiv:2412.15605

[6] Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '02)*. https://doi.org/10.1145/509907.509965

[7] 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). https://www.microsoft.com/en-us/research/wp-content/uploads/2021/11/SPANN_finalversion1.pdf

[8] Flavio Chierichetti, Ravi Kumar, and Sergei Vassilvitskii. 2009. Similarity caching. In *Proceedings of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Providence, Rhode Island, USA) *(PODS '09)*. https://doi.org/10.1145/1559795.1559815

[9] Ranul Dayarathne, Uvini Ranaweera, and Upeksha Ganegoda. 2024. 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*. Springer. https://doi.org/10.1007/978-981-97-9255-9_26

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

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

[12] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. 2008. A metric cache for similarity search *(LSDS-IR '08)*. https://doi.org/10.1145/1458469.1458473

[13] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. 2012. Similarity caching in large-scale image retrieval. *Information processing & management* 48, 5 (2012). https://doi.org/10.1016/j.ipm.2010.12.006

[14] Ophir Frieder, Ida Mele, Cristina Ioana Muntean, Franco Maria Nardini, Raffaele Perego, and Nicola Tonellotto. 2024. Caching historical embeddings in conversational search. *ACM Transactions on the Web* 18, 4 (2024). https://doi.org/10.1145/3578519

[15] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338. https://doi.org/10.48550/arXiv.2311.04934

[16] Google. 2025. AI Overviews in Search. https://search.google/ways-to-search/ai-overviews Accessed: 2025-09-18.

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

[18] Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. 2024. EPIC: Efficient Position-Independent Caching for Serving Large Language Models. (2024). arXiv:2410.15332

[19] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025). https://doi.org/10.1145/3703155 arXiv:2311.05232

[20] 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). https://papers.nips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[21] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards mitigating LLM hallucination via self reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. https://doi.org/10.18653/v1/2023.findings-emnlp.123 arXiv:2310.06271

[22] Wenqi Jiang, Suvinay Subramanian, Cat Graves, Gustavo Alonso, Amir Yazdanbakhsh, and Vidushi Dadu. 2025. RAGO: Systematic Performance Optimization for Retrieval-Augmented Generation Serving. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 974–989. https://doi.org/10.1145/3695053.3731093

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

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

[25] Qiao Jin, Won Kim, Qingyu Chen, Donald C Comeau, Lana Yeganova, W John Wilbur, and Zhiyong Lu. 2023. Medcpt: Contrastive pre-trained transformers with large-scale pubmed search logs for zero-shot biomedical information retrieval. *Bioinformatics* 39, 11 (2023). https://doi.org/10.1093/bioinformatics/btad651 arXiv:2307.00589

[26] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011). https://doi.org/10.1109/TPAMI.2010.57

[27] 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. (2020). arXiv:2001.08361

[28] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://doi.org/10.18653/v1/2020.emnlp-main.550

[29] Yoonjoo Lee, Kihoon Son, Tae Soo Kim, Jisu Kim, John Joon Young Chung, Eytan Adar, and Juho Kim. 2024. One vs. Many: Comprehending Accurate Information from Multiple Erroneous and Inconsistent AI Generations. In *The 2024 ACM Conference on Fairness, Accountability, and Transparency*. https://doi.org/10.1145/3630106.3662681 arXiv:2405.05581

[30] 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. *Advances in Neural Information Processing Systems* 33 (2020). arXiv:2005.11401 https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[31] Songshuo Lu, Hua Wang, Yutian Rong, Zhi Chen, and Yaohua Tang. 2024. TurboRAG: Accelerating Retrieval-Augmented Generation with Precomputed KV Caches for Chunked Text. (2024). arXiv:2410.07590

[32] Craig Macdonald and Nicola Tonellotto. 2021. On approximate nearest neighbour selection for multi-stage dense retrieval. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3318–3322. https://doi.org/10.1145/3459637.3482156

[33] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018). https://doi.org/10.1109/TPAMI.2018.2889473 arXiv:1603.09320

[34] Emma Meats, Jon Brassey, Carl Heneghan, and Paul Glasziou. 2007. Using the Turning Research Into Practice (TRIP) database: how do clinicians really search? *Journal of the Medical Library Association* 95, 2 (2007). pubmed:17443248 https://pmc.ncbi.nlm.nih.gov/articles/PMC1852632/

[35] Microsoft. 2024. Copilot Search. https://www.microsoft.com/en-us/bing/copilot-search Accessed: 2025-09-18.

[36] Ryo Nakamura and Noriaki Kamiyama. 2024. Analysis of Similarity Caching on General Cache Networks. *IEEE Access* (2024). https://doi.org/10.1109/ACCESS.2024.3489620

[37] OpenAI. 2024. ChatGPT Search. https://openai.com/chatgpt/search Accessed: 2025-09-18.

[38] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33, 5 (2024). https://doi.org/10.1007/s00778-024-00864-x arXiv:2310.14021

[39] Sandeep Pandey, Andrei Broder, Flavio Chierichetti, Vanja Josifovski, Ravi Kumar, and Sergei Vassilvitskii. 2009. Nearest-neighbor caching for content-match applications. In *Proceedings of the 18th International Conference on World Wide Web* (Madrid, Spain) *(WWW '09)*. https://doi.org/10.1145/1526709.1526769

[40] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating Retrieval-Augmented Generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. https://doi.org/10.1145/3669940.3707264

[41] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating Retrieval-Augmented Generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 15–32. https://doi.org/10.1145/3669940.3707264

[42] Siddhant Ray, Rui Pan, Zhuohan Gu, Kuntai Du, Ganesh Ananthanarayanan, Ravi Netravali, and Junchen Jiang. 2024. METIS: Fast Quality-Aware RAG Systems with Configuration Adaptation. (2024). arXiv:2412.10543

[43] Sajal Regmi and Chetan Phakami Pun. 2024. GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching. (2024). arXiv:2411.05276

[44] Sohini Roychowdhury. 2024. Journey of hallucination-minimized generative ai solutions for financial decision makers. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*. 1180–1181. https://doi.org/10.1145/3616855.3635737

[45] Anirudh Sabnis, Tareq Si Salem, Giovanni Neglia, Michele Garetto, Emilio Leonardi, and Ramesh K. Sitaraman. 2023. GRADES: Gradient Descent for Similarity Caching. *IEEE/ACM Transactions on Networking* 31, 1 (2023). https://doi.org/10.1109/TNET.2022.3187044

[46] Tolga Şakar and Hakan Emekci. 2025. Maximizing RAG efficiency: A comparative analysis of RAG methods. *Natural Language Processing* 31, 1 (2025). https://doi.org/10.1017/nlp.2024.53

[47] Pavlos Sermpezis, Theodoros Giannakas, Thrasyvoulos Spyropoulos, and Luigi Vigneri. 2018. Soft cache hits: Improving performance through recommendation and delivery of related content. *IEEE Journal on Selected Areas in Communications* 36, 6 (2018). https://doi.org/10.1109/JSAC.2018.2844983

[48] Michael Shen, Muhammad Umar, Kiwan Maeng, G Edward Suh, and Udit Gupta. 2024. Towards Understanding Systems Trade-offs in Retrieval-Augmented Generation Model Inference. (2024). arXiv:2412.11854

[49] Jonathon Shlens. 2014. A Tutorial on Principal Component Analysis. arXiv:1404.1100

[50] Amanda Spink, Dietmar Wolfram, Major BJ Jansen, and Tefko Saracevic. 2001. Searching the web: The public and their queries. *Journal of the American society for information science and technology* 52, 3 (2001). https://doi.org/10.1002/1097-4571(2000)9999:9999<::AID-ASI1591>3.0.CO;2-R

[51] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2024. Preble: Efficient distributed prompt scheduling for llm serving. (2024). arXiv:2407.00023

[52] Jaime Teevan, Eytan Adar, Rosie Jones, and Michael AS Potts. 2007. Information re-retrieval: Repeat queries in Yahoo's logs. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. https://doi.org/10.1145/1277741.1277770

[53] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008). https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf

[54] Zilong Wang, Zifeng Wang, Long Le, Huaixiu Steven Zheng, Swaroop Mishra, Vincent Perot, Yuwei Zhang, Anush Mattapalli, Ankur Taly, Jingbo Shang, et al. 2025. Speculative RAG: Enhancing retrieval augmented generation through drafting. (2025). arXiv:2407.08223 https://openreview.net/pdf?id=xgQfWbV6Ey

[55] Guangzhi Xiong, Qiao Jin, Zhiyong Lu, and Aidong Zhang. 2024. Benchmarking Retrieval-Augmented Generation for Medicine. In *Findings of the Association for Computational Linguistics ACL 2024*. https://doi.org/10.18653/v1/2024.findings-acl.372

[56] Lexin Zhou, Wout Schellaert, Fernando Martínez-Plumed, Yael Moros-Daval, Cèsar Ferri, and José Hernández-Orallo. 2024. Larger and more instructable language models become less reliable. *Nature* 634, 8032 (2024). https://doi.org/10.1038/s41586-024-07930-y

[57] Yun Zhu, Jia-Chen Gu, Caitlin Sikora, Ho Ko, Yinxiao Liu, Chu-Cheng Lin, Lei Shu, Liangchen Luo, Lei Meng, Bang Liu, et al. 2024. Accelerating Inference of Retrieval-Augmented Generation via Sparse Context Selection. (2024). arXiv:2405.16178