Enhance Graph Retrieval-Augmented Generation by Caching method

Quoc-Huy Trinh
Aalto University
Espoo, Finland
huy.trinh@aalto.fi

Anh Dao
Aalto University
Espoo, Finland
anh.d.dao@aalto.fi

Pablo Rubio
Aalto University
Espoo, Finland
pablo.rubiogarcia@aalto.fi

Abstract

Retrieval-Augmented Generation (RAG) has become a widely adopted technique for improving the reliability of Large Language Models (LLMs) by grounding their outputs in external knowledge. However, traditional RAG systems struggle with highly relational datasets, where semantically connected pieces of information must be retrieved jointly. GraphRAG addresses this limitation by leveraging graph-structured knowledge bases but introduces substantial computational overhead due to repeated graph traversal and query generation. In this work, we enhance the GraphRAG pipeline by integrating a Least Recently Used (LRU) caching mechanism and a memory-efficient caching variant that combines context pruning and on-the-fly compression. These methods reduce redundant computations across similar queries and significantly lower runtime costs. Our results show that memory-efficient LRU caching achieves the highest throughput and lowest latency at scale, outperforming both non-cached and baseline caching approaches. While the improvements in efficiency are substantial, further work is needed to enhance the accuracy of Cypher generation for complex queries. The implementation of this report can be viewed at https://github.com/huyquoctrinh/ScalableSys-Proj2

ACM Reference Format:
Quoc-Huy Trinh, Anh Dao, and Pablo Rubio. 2024. Enhance Graph Retrieval-Augmented Generation by Caching method. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large Language Models (LLMs) have emerged as a highly promising technology due to their wide-ranging applications across finance, healthcare, education, and customer services. Recent advancements—such as GPT [1], Qwen [2], Gemini [7], and LLaMA [4], demonstrate impressive capabilities, particularly in responding to user queries with increasingly human-like reasoning. However, a core challenge remains: ensuring the reliability and correctness of model outputs, especially when dealing with unseen or out-of-distribution data. This limitation poses significant risks in real-world deployment.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

While fine-tuning or post-training can improve model performance on specific domains, these approaches require substantial computational resources, making them expensive and sometimes impractical for large-scale models. To address this issue, Retrieval-Augmented Generation (RAG) [6] has been proposed as an efficient alternative. RAG leverages an embedding model to convert segmented documents into vector representations, enabling the system to dynamically retrieve relevant information during inference and ground the model's responses in a factual context.

Although RAG demonstrates strong potential in LLM applications, it faces limitations in modeling relationships between semantically connected chunks. When different pieces of information are interdependent, treating them as isolated embeddings can lead to incomplete or suboptimal retrieval. To address this issue, Graph RAG [5] was introduced, using graph-structured representations to capture relationships among data segments. While Graph RAG achieves promising results for tasks involving structured or interconnected data, it still suffers from significant computational overhead. As the graph scales, search operations become increasingly expensive, which becomes problematic when graph traversal must be performed for every user query.

To mitigate these challenges, we implement the Least Recently Used (LRU) caching mechanism and introduce a Memory Efficient Caching method for the system. This method stores previously asked questions and their corresponding answers, allowing the system to reuse results when a similar query appears in subsequent interactions. By reducing redundant retrieval and graph search operations, LRU caching improves efficiency and lowers computational cost. Furthermore, to ensure that the LLM consistently produces outputs in the required format, we incorporate a few-shot [3] prompting strategy. This approach guides the model with structured examples, enabling it to follow instructions more reliably and generate well-formatted responses aligned with the task requirements.

In summary, our contribution includes threefold:

- We integrate the LRU caching method into the Graph RAG to reduce the computing resources of the overeall system.
- We implement the few-shot prompting method to let the model solve the Text2Cypher problem.
- We conduct extensive experiments to assess the efficient and the effectiveness of the caching method in the GraphRAG

The rest of the report includes these sections: Section 2 presents the overall of our system architecture, Section 3 describes our experimental setup and results of our system, Section 4 summarizes and explains our contributions. Finally, Section 5 presents the contributions of each member.

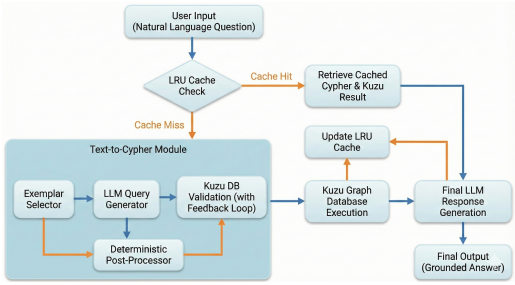

Figure 1: Overall of our system architecture

2 System Architecture

2.1 Overview

Figure 1 presents the overall system architecture. Our system consists of four main components: **(1)** an LRU cache for storing previously processed queries and responses, **(2)** a KUZU database for storing graph embeddings, **(3)** an embedding model to generate embeddings for graph nodes, and **(4)** a Large Language Model (LLM) to produce the final answer.

Given a user query, the system first checks the LRU cache to determine whether a similar query has appeared in previous turns. If a match is found, the cached retrieval results and the previous LLM response are reused to generate the final output. If no match exists, the query is processed through the Graph RAG pipeline, which retrieves the most relevant information from the KUZU database before passing the retrieved context to the LLM for answer generation.

Within the Large Language Model component, we employ a few-shot prompting strategy to ensure that the generated responses follow the desired output format. The few-shot examples guide the model to answer the query according to the provided instructions. This enables the LLM to effectively utilize the retrieved context and produce responses that align with the expected output.

2.2 Graph RAG

Graph Retrieval-Augmented Generation (Graph RAG) extends normal RAG by using a structured graph database as the retrieval source instead of unstructured documents. It exploits relationships between points or entities (nodes) to retrieve a coherent subgraph that represents better the relation between the data being analysed avoiding this way the generation of wrong info. This makes it particularly suitable for highly relational datasets such as the Nobel laureate network explored in this work.

In our system, the knowledge base is represented as a heterogeneous graph stored in the Kuzu database, cointaning nodes such as *Laureates*, *Scholars*, *Prizes*, *Institutions*, as well as edges such as *AFFILIATED WITH* or *AWARDED*). This graph structure enables retrieval that is driven by logical constraints rather than just textual similarity.

The GraphRAG pipeline begins with a Text2Cypher component, where a LLM translates a natural language question into an Cypher query. To ensure that the LLM generates a query aligned with the user's intent, we employ an embedding-based exemplar selection mechanism. Using this, the incoming question is embedded using a sentence-level embedding model, and the most semantically similar few-shot examples are retrieved to guide the LLM during query generation. These embeddings therefore determine which examples best match the semantic space of the user's question, improving the consistency and accuracy of the produced Cypher query. This query is then executed on the Kuzu database, producing the response that the LLM recieves along with the original question and produces a factual and grounded answer. Because the model only operates on the retrieved graph-derived context, the risk of hallucination is substantially reduced compared to the standard RAG.

2.3 Few-shot prompting in Text-to-Cypher

To bridge the gap between natural language and graph queries, we employ a multi-stage strategy designed to handle the complexity of the underlying graph schema and ensure query correctness. This strategy consists of three key phases:

Dynamic Few-shot Prompting. Standard zero-shot or static few-shot prompting often fails to capture the nuances of a complex graph schema. To address this, we adopt a dynamic retrieval approach. The core idea is to provide the LLM with examples that are semantically similar to the current user question. By dynamically selecting the most relevant question-query pairs from a curated index, we condition the model on patterns that are likely to be useful for the specific task at hand. This context-aware prompting significantly improves the model's ability to generate valid Cypher queries that respect the graph's structure.

Iterative Refinement with Database Feedback. Even with good prompting, Large Language Models can produce hallucinations or syntax errors. We mitigate this by treating query generation as an iterative process rather than a single-shot task. Our strategy incorporates a feedback loop where the database itself acts as a verifier. The system attempts to validate the generated query; if the database reports an error, this feedback is passed back to the model. The model then attempts to "repair" the query based on the specific error message, effectively learning from its mistakes in real-time.

Deterministic Post-processing. While LLMs are powerful, they are probabilistic and can struggle with strict formatting requirements or domain-specific conventions. To ensure robustness, we employ a hybrid approach that combines the generative capabilities of the LLM with deterministic code. A post-processing layer acts as a final safeguard, enforcing specific rules such as case-insensitivity for string matching and correcting common schema hallucinations. This ensures that the final output is not only syntactically valid but also optimized for execution and human readability.

2.4 LRU caching

Although the GraphRAG system is powerful, it faces significant computational challenges when serving many concurrent users. Querying large subgraphs, computing node embeddings, and performing multi-hop retrieval can be expensive when repeated across similar user requests. To reduce unnecessary recomputation and improve system responsiveness, we integrate a Least Recently Used (LRU) caching mechanism into our architecture. The LRU cache stores the results of recent GraphRAG operations, such as retrieved neighborhoods, semantic clusters, or ranking outputs, so that repeated queries can be returned immediately. When the cache

reaches its capacity, the least recently accessed entry is evicted, ensuring that storage is reserved for results that are most relevant to ongoing workloads.

Our implementation uses a combination of a hash map and a doubly linked list to achieve $O(1)$ time for both lookup and update operations. The implementation of this idea is used via the Orderdict in python, and included in the cache package.. The hash map provides constant-time access to cached entries, while the doubly linked list maintains the usage order, with the head representing the most recently used item and the tail representing the least recently used. Whenever a cached value is accessed or inserted, its node is moved to the head of the list; when capacity is reached, the tail node is removed. This design allows GraphRAG to reuse previously computed retrieval results efficiently, significantly reducing latency, lowering system load, and enabling more scalable and consistent performance under concurrent user activity.

Moreover, to improve the results of the model, and make the memory efficient, we implement the memory-efficient caching.

Memory Efficient Caching: To further mitigate the memory overhead of caching, we implement a memory-efficient variant of the LRU cache that combines *context pruning* and *on-the-fly compression*. The cache extends a standard LRU-based `DataManager` and intercepts each `set` operation. Before storing a value, the cache prunes oversized context fields by retaining at most a fixed number of context items (e.g., the first N entries, which are typically the most relevant). The resulting object is then serialized using `pickle` and, if its serialized size exceeds a configurable threshold, compressed with `zlib`. Compression is only applied if it yields at least a 10% reduction in size, thereby avoiding unnecessary CPU overhead for objects that do not compress well.

On retrieval, the cache transparently reverses this process by decompressing (when necessary) and deserializing the stored bytes back into the original Python object. The implementation maintains detailed statistics, including the cumulative original and stored sizes, compression ratio, percentage of memory saved, and the number of pruned and compressed entries. In practice, this design achieves substantial memory savings (often 60–90% for large context-heavy entries) while preserving the fast lookup properties of the underlying LRU cache.

3 Implementation

3.1 Text-to-Cypher Implementation

The Text-to-Cypher module is implemented as a pipeline with three key components, leveraging Python libraries such as `scikit-learn` and `DSPy`:

3.1.1 Exemplar Selector. The `ExemplarSelector` class implements our dynamic prompting strategy. It indexes a curated set of question-query pairs using `TfidfVectorizer` from `scikit-learn`. At runtime, it performs the following operations:

- Encodes the user's input question into a TF-IDF vector.
- Computes cosine similarity between the input vector and the pre-computed exemplar vectors.
- Retrieves the top-k (default $k = 3$) matches to be injected into the LLM prompt context.

3.1.2 Query Generator with Refinement. The `QueryGenerator` class uses the `DSPy` framework to orchestrate the generation process. It defines a `RepairCypher` signature that takes the original query, error message, and schema as input. The generation flow is as follows:

(1) **Generation**: An initial query is produced using the retrieved exemplars.
(2) **Validation**: We execute the `EXPLAIN` clause in KuzuDB. This validates the query syntax and schema compliance without running the actual potentially expensive query.
(3) **Repair**: If validation fails, the `RepairCypher` module (implemented as a `dspy.ChainOfThought`) uses the error message to generate a fix. This loop runs for a maximum of 3 iterations.

3.1.3 Post-Processor. The `CypherPostProcessor` class enforces deterministic rules using regular expressions to handle edge cases:

(1) **Case-Insensitivity**: It identifies equality checks on string properties (e.g., `s.name = '...'`) and converts them to case-insensitive containment checks (e.g., `LOWER(s.name) CONTAINS '...'`).
(2) **Expansion**: It detects generic return statements (e.g., `RETURN s`) and expands them to specific properties like `s.knownName` or `p.awardYear` to ensure meaningful output.
(3) **Schema Correction**: It maps common hallucinated properties to their correct schema equivalents, such as correcting `.amount` to `.prizeAmount`.
(4) **Cleanup**: It removes unsupported functions (e.g., APOC calls) and normalizes whitespace for clean execution.

3.2 Experimental setup

In the experiments, the system is implemented with the chain-of-thought method via the dspy framework. For the graph database, we leverage the Kuzu framework as the vector database. All of the implementation is conducted on the PC with I7 146650HX, which includes 24 vCPUs, and 32 GB RAM.

3.3 Dataset

To evaluate the effectiveness of our Text-to-Cypher pipeline, we construct a benchmark dataset inspired by the structure and annotation format of the Neo4j Text2Cypher dataset. The dataset is generated using the GPT-5 model to ensure linguistic diversity and realistic query patterns that resemble real-world information needs. To analyze system performance under varying workload conditions, we additionally create subsets of size 20, 40, and 60 samples. These subsets enable controlled experiments to assess scalability, robustness, and accuracy across different evaluation scenarios.

3.4 Metric

To evaluate the effectiveness of our enhanced Graph RAG pipeline, we report metrics along two complementary dimensions: (1) the quality of the generated Cypher queries in the Text2Cypher task, and (2) the runtime performance of the end-to-end system, with and without LR caching enabled.

System Throughput. To evaluate the runtime efficiency of the overall system, we measure throughput using two metrics: (1) *queries per second* (QPS), which captures how many natural language questions the system can process end-to-end per unit

time, and (2) *tokens per second* (TPS), which reflects the effective decoding speed of the LLM during answer generation. These metrics provide a holistic view of system scalability across both retrieval and generation stages.

Latency. We measure the execution time of each stage of the Graph RAG pipeline: (i) exemplar selection via sentence-level embeddings, (ii) LLM-based query generation, (iii) refinement iterations, (iv) Kuzu query execution, and (v) final answer synthesis by the LLM. This breakdown allows us to attribute total runtime to specific components and evaluate the benefit of caching.

3.5 Experimental Results

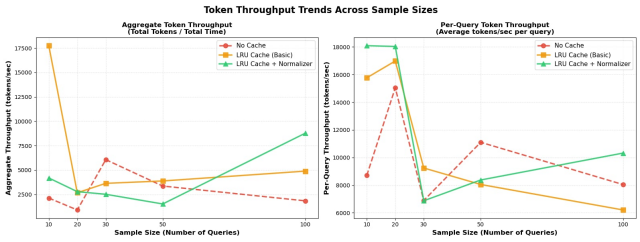

Figure 2: Evaluation of the throughput of the system in the query and in the token level

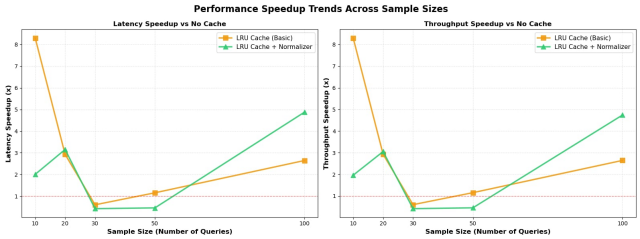

Figure 3: Evaluation of the latentcy of the system in the query and in the token level

Throughput results: Figure 2 describes the throughput of our system across three scenarios: non-caching, caching, and mem-efficient caching method with query normalizer. In general, the LRU Cache + Normalizer method demonstrates the most stable and scalable performance, achieving the highest throughput at larger sample sizes and maintaining strong consistency across experiments. The basic LRU Cache exhibits the fastest performance at small sample sizes but shows significant degradation and instability as the workload increases. In contrast, the baseline with no caching performs the worst overall, with lower throughput and declining efficiency at higher sample sizes. These results indicate that integrating normalization into the LRU caching mechanism provides clear advantages for sustaining high performance under increasing system load.

Latency results: Figure 3 shows the latency benchmark of our system across three scenarios: non-caching, caching, and mem-efficient caching method with query normalizer. Benchmark results show that the LRU cache significantly improves average response time as the sample size increases. At small workloads (sample size = 20), the LRU-enabled system initially exhibits higher latency (0.10s) compared to the non-cached baseline (0.055s), largely due to the overhead of cache initialization and early cache misses. However, as the sample size grows, the LRU cache rapidly becomes more effective: at 40 queries, both methods converge to similar latency, and beyond 60–80 queries the LRU-based system clearly outperforms the baseline, reducing latency to 0.03s compared to 0.04–0.045s without caching. This trend highlights that while caching introduces small overhead at the beginning, it substantially reduces average latency under moderate to heavy workloads, making LRU caching more efficient and scalable for larger query batches.

4 Conclusion

In conclusion, this study integrates few-shot prompting with an LRU-based Memory Efficient Caching strategy to enhance the performance of the Graph RAG system. Across a wide range of sample sizes, our proposed caching method consistently achieves the highest throughput and lowest latency, demonstrating its effectiveness in improving system efficiency under varying workloads. The LRU cache clearly outperforms the non-cached baseline, especially at larger query volumes, highlighting its scalability and practical benefits for real-world deployments. While these results are promising, further work is needed to improve the correctness of generated answers in the complex questions input to ensure that the system is not only efficient but also highly accurate.

5 Team Contribution

This project was done by Quoc-Huy Trinh, Anh-Dao Dinh, and Pablo Rubio. During the implementation process, we set up triple programming once a week in the coffee shop, and we also have asynchronous meetings to discuss the idea and the architecture of the overall system. All of the implementations are committed on GitHub, and Quoc-Huy Trinh will merge all of the commits to construct our final codebase. During implementation, we tested with different approaches for the LRU caching method and dataset samples to observe the performance of the overall system. For the final report, we write it together to ensure emphasize all of the idea and content that we did. Regarding the role and implementation part for each member, we divide the role as follows:

- Quoc-Huy Trinh: Huy led the project, merged all implementations from Huy and team members to construct the final codebase. In the implementation, he led the implementation of the LRU caching method, and all of the optimization parts for the LRU caching method. Beside that he supports all of the team member in setting up the codebase and preparing the approaches to solve the challenge during the implementation. In his work, he faced with the challenge regarding the raising of the memory of the caching, however, he tried some more efficient ways to prevent this problem and succeeeded.
- Anh Dao: Anh led the implementation for the few-shot prompting and create the dataset. During the implementation, he met the challenges in understanding the few-shot prompting problem when he is not familiar with this. However, during the implementation, he understand how can we apply a template and instruct the LLM to follow that.

- Pablo Rubio: Pablo led the part of implementing the benchmark metrics and benchmark pipeline for the overall system. In this work, he has learned a lot about the benchmark of the LLM system, how the RAG system works, and how we can apply prompt engineering into the LLM system.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[5] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. 2024. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309* (2024).

[6] 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), 9459–9474.

[7] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).