GRACE: GRAph CachE System for RAG Applications

Zachary Aristei Rishi Banerjee Apoorva M. Krishnamurthy

Georgia Institute of Technology, College of Computing

Introduction

In GraphRAG systems, there are two bottlenecks, first, fetching the relevant part of the sub-graph, and second, the decoding process by the large language model. To exploit the temporal, spatial, and semantic similarities in typical query workloads, we introduce an intelligent cache in the graph RAG system. We cache the relevant nodes and documents fetched corresponding to each query. We also explore having a semantic cache, ie, we look for semantically similar queries in the cache.

Motivation and Background: In most Q&A applications, the incoming query workload has some patterns that can be exploited, ie, either in a session, a user is likely to look for entities similar to each other or given many independent queries from different users at a time, they might also be looking for similar items, especially in applications like a search engine. In a GraphRAG system, caching the relevant graph entities or neighborhoods prevents expensive and repeated graph querying. Another strength of LLMs is their ability to generate coherent responses for different queries even if the provided context loosely adheres to a template. This implies that semantically similar queries can utilize the same contextual information in the cache. Most open-source GraphRAG systems currently have no caching system behind the scenes.

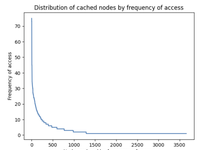

Figure 1. Distribution of Accessed KG Nodes, random BeerQA Queries

Figure 2. Hit Rate for Retrieved Nodes, Random BeerQA Queries

Design and Implementation

To create our data source we leverage LlamaIndex to integrate our vector database with our knowledge graph. We utilize a ChromaDB endpoint for generating and storing our document vector database. For our Knowledge Graph, we hosted a Neo4j database on a Kubernetes cluster on GCP. We constructed our knowledge graphs using REBEL-Large to extract triplets from the documents in our datasets. Our cache was implemented using a local Redis instance. Finally, for our LLM queries, we utilize OpenAI's GPT-3.5 turbo model and its corresponding API. For our experiments, we decided to focus our efforts on two different types of datasets:

- **Stanford's CoQA** dataset to evaluate persistence associated with asking repeated questions on the same context window.
- **Stanford's BeerQA** dataset to evaluate performance on multi-hop question-answering tasks.

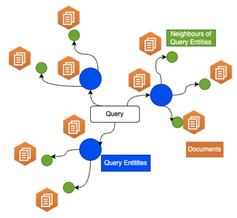

Figure 3. Overview of Retrieval Process

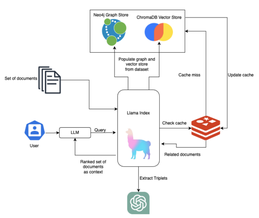

Figure 4. GRACE System Design

Retrieval Process

In order to perform a RAG query, the following steps must be executed:

1. Extract triplets from the query, and match them with the closest entities on the knowledge graph.
2. Retrieve the local neighborhood within *d* hops of each entity.
3. For each node in the neighborhood, extract the top *k* documents embedded nearby. The documents to be extracted get cached with their corresponding nodes.
4. Use a ranking model to return a fixed subset of the returned documents relevent to the query which the LLM then utilizes to generate an answer.

Cache Latencies across Procedure Steps

In order to determine the primary contributions to latency for RAG queries, we compare GRACE's caching architecture with a conventional Graph RAG workflow that omits caching. We find that caching provides significant reductions in latency, especially when the time needed to traverse the knowledge graph for local relevant information is negligible when compared with the cost of retrieving whole documents related to the query in the vector database.

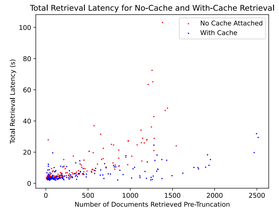

Figure 5. Latency of queries as a function of documents retrieved in local neighborhood, BeerQA

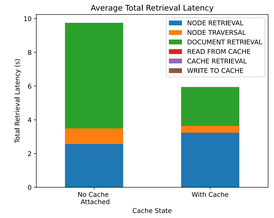

Figure 6. Latency of BeerQA queries on Knowledge Graph Cache v. No Cache

Effects of Traversal Depth on Query Time

For GraphRAG usability in multi-hop question answering use cases, we want to determine to what extent cache performance improves latencies when the knowledge graph traversal depth is increased. We find that there's little correlation between traversal depth and increases in latency, but with respect to the retrieval and ranking times for documents, but we see that document retrieval and ranking latencies do increase markedly as more documents get retrieved with the discovered entities.

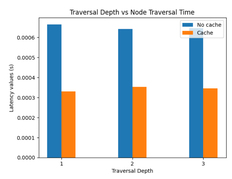

Figure 7. Latency of BeerQA KG Traversal Based on Depth

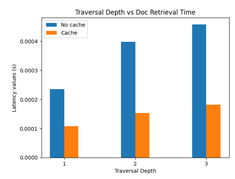

Figure 8. Latency of BeerQA Documents to Retrieve Based on Depth

Semantic Caching as a Pre-Retrieval Step

In order to further decrease average document retrieval speeds, we added semantic caching as a preliminary step to return documents directly for queries similar to those previously asked. Thus, unlike past experiments, this step has possibility to degrade retrieval accuracy, as measured in Figure 10.

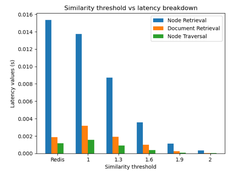

Figure 9. Avg End-to-End Retrieval Latency based on Semantic Cache similarity Threshold

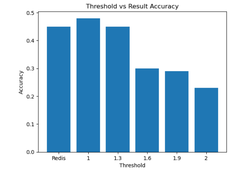

Figure 10. Retrieved Document Accuracy Degradation from Semantic Cache Threshold Increase

Discussion

There is a significant latency improvement on end-to-end retrieval time due to much of node traversal and document retrieval computation being stored in the cache. This shows that in conversational query use-cases, caching shows promising results for quick, accurate contextual retrieval. We can see that this trend also holds as the traversal depth is increased, offsetting the computational load required by the exponential increase in neighbors as traversal depth increases.

While the semantic caching layer shows promise in the ability to reduce the average node retrieval time, which is held mostly constant with or without the document cache, it also holds an issue of needing to be tuned to avoid performance degradation. A system can use a hybrid of both caching systems with the semantic cache being the first line of defense to boost performance even further.

Our experiments have shown promise in applying caching to the retrieval step of RAG systems for two major use cases; building conversational agents and building augmented information retrieval systems. This work can be applied when building and deploying RAG systems in order to reduce the latency required for the retrieval step, and thus provide hope to build LLM systems that are better based on factual information and able to serve requests in reasonable timeframes.

Limitations and Future Work

For the parsing of the Wikipedia data, in order to be parsimonious with the data being chunked due to the computational constraints, we only included the data that is being used in the subset of BeerQA queries that were being tested for latency statistics. In a production-grade system, informational websites like Wikipedia would be crawled and parsed into the knowledge graph index and vector database. We aim to continue this research using data that is more representative of the imbalance in content of workloads inherent to most search engine systems.

For future work, we aim to apply the principles learned from these experiments into the construction of deployed RAG systems for conversational agents and search engines. The first future experiment we would want to test is the handling of a large number of requests in to divide the workload into multiple different instances, where some queries are directly filtered out by a query semantic cache, and the other queries are batched through a process like clustering in order to pass a set of similar nodes to the traversal step and document retrieval in order to take advantage of similar retrievals.

Stanford Graph Learning Workshop 2024, Stanford, CA