Context-based Semantic Caching for LLM Applications

Ramaswami Mohandoss
Data Analytics and AI
Infosys Ltd
Bengaluru, India
https://orcid.org/0000-0001-9653-0791

***Abstract*— Large Language Models (LLM), aided by the popularity of ChatGPT, have provided a paradigm shift to engineering AI Chatbots. LLMs offer many conveniences to the AI engineer, and the most important benefit is their robustness in handling semantic matches of user queries. In other words, an engineer building an AI conversational assistant does not need to train the agent for semantic user query matches explicitly. This benefit comes with a cost, which is felt in two ways. Firstly, LLMs need an expensive infrastructure like GPUs, large RAMs, etc. Secondly, even with all the cutting-edge infrastructure, their response time will not be sub-second anytime soon. So, AI applications that use LLMs are likely to be expensive and suffer from high latency. One way to reduce cost and response time is by introducing a caching solution between the LLMs and the UX layer. Such a caching layer can help minimize calls to LLMs when the queries are similar and repetitive. However, traditional caching methods cannot be used as they are for LLM-based applications. The reason is that queries handled by AI-based conversational assistants are unstructured, i.e., free-flowing user-generated text, and will not always be context-free. In other words, end users tend to query from their point of view. Existing solutions like GPTCache work well for context-free questions, but caching context-sensitive user queries needs an evolved design. In this paper, we shall explore a novel design that, by exploiting the power of context, shall provide effective caching solutions to user-generated queries (both context-free and context-sensitive) that offer improved performance without compromising on the quality of response.**

Keywords*— *LLM, Large Language Models, Semantic cache, Natural Language Processing, Artificial Intelligence, Generative AI, Conversational Assistants, Chatbot.

I. INTRODUCTION

Designing a caching layer for an application that uses structured data is straightforward. The cache is typically an in-memory layer that can hold the request and the response as *<key, value>* pairs. For instance, a Social Media application that needs to refer to the user's information frequently might cache the user attributes when the user logs in. This will help accelerate the application's performance by minimizing database calls. Here, the user's identifier could be the *key*, and the set of user attributes could be held as a document (aka *value*) in the caching layer. What needs to be noted here is that both *key* and *value* are structured data, which the application will not have an ambiguity to deal with.

The situation is quite different when designing a caching layer for an AI application that leverages an LLM. The request is a user-generated query, and the response is AI-generated. The user-generated query is unstructured text and caching that unstructured text as a *key* will not yield the same benefit as in the earlier scenario. There are multiple challenges here. We will see them one by one.

The first challenge is handling the semantics of the user-generated text. The following two questions mean the same: *'How far is NYC from Seattle?'* and *'What's the distance between NYC and Seattle?'*. For the caching layer to be useful, what needs to be matched during retrieval is the intent behind the text. GPTCache has tried addressing this gap through a semantic caching solution. We shall see the details in the next section.

Handling semantics is only part of the problem. Some queries involve spatial data, especially those originating from moving entities like mobile phones. Response to questions like *'Find good restaurants near me'* cannot be cached until they originate from the exact geolocation. Caching queries and responses involving spatial data require a strategy different from semantic caches.

Some queries could be specific to the individual. Responses to questions like *'Suggest training that I should invest in'* must be personalized for the individual, fully considering the individual's context.

In summary, caching user-generated queries and their responses is not a trivial problem and is much more than semantic caching. The caching solution should consider the user's context and perform similarity checks on the user's query before responding.

In this paper, we will present a design of a caching solution for LLM applications that not only addresses the aforementioned challenges but also seamlessly adapts to any user-defined context. This innovative design is called the Context-based semantic caching solution and is built on the foundation of the semantic caching design of GPTCache. We will review the semantic caching solution before moving on to the Context-based design.

II. RELATED WORK

We will begin with a review of the GPTCache design articulated by Fu Bang and De Beng in their article[1]. According to this design, the query text is not held as a key but kept as an embedding in a vector database. Before responding to any user query, the query is first converted to an embedding, and the nearest match (based on cosine distance or other methods) from the semantic cache is checked. If a match is found, the response is retrieved through the cache. When there is no satisfactory match, the LLM is leveraged to generate a response, which will be cached for future questions. This logical flow is shown in the Figure 1.

979-8-3503-5409-6/24/$31.00 ©2024 IEEE
DOI 10.1109/CAI59869.2024.00075

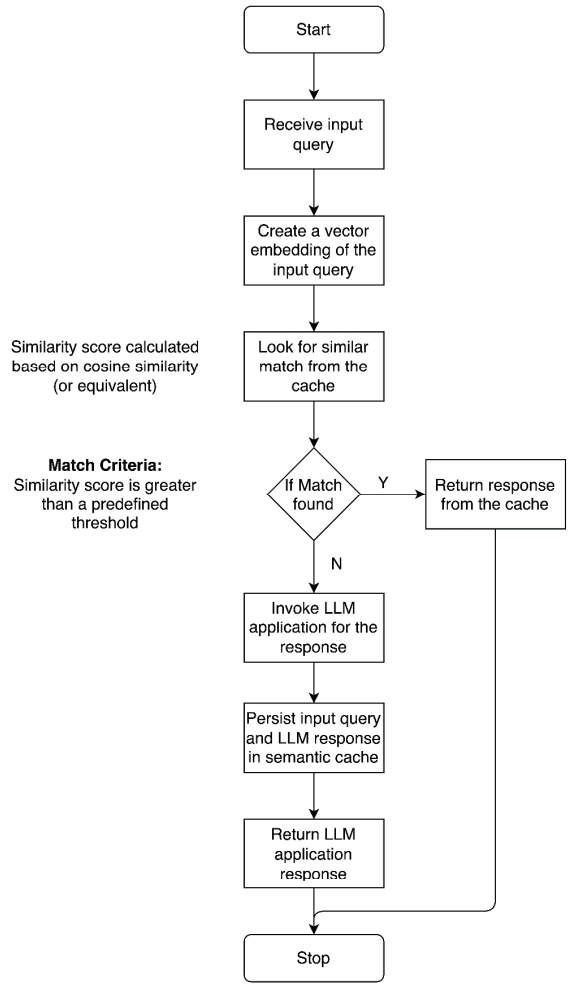

Fig. 1. Flow of semantic caching solution

III. CONTEXT-BASED SEMANTIC CACHING DESIGN

In the introduction section, we saw multiple scenarios, i.e., queries where semantic caching fell short. That's because, in all scenarios, the context of the source of the query was not considered. In other words, these queries typed by the end user are not context-free. An end user is a moving entity, and it is normal for them to query from their point of view (queries like *restaurants near me*), and their coordinates change over time. By coordinates, we are not referring to geolocation alone. Apart from the user's geolocation, many dimensions can pinpoint a user. For instance, dimensions like the country, organization, role, and title can pinpoint the user's coordinates in their respective worlds.

In summary, the user's context can be any set of dimensions, and for a caching solution to be effective, it is inevitable to handle the user's context along with semantic matching. We shall cover the design of the context-based caching solution in four parts. In the first part, we shall review the foundational constructs we will need to capture and manage context. In the second part, we shall go over the data structure of the caching layer. We will then go over the algorithm. Finally, we shall review the design of the context-based semantic caching solution.

A. Context Model

The design of the caching solution we are about to deliberate needs six foundational constructs to capture and manage the context of the moving entity. The moving entity is the end user. We shall define these constructs both formally and informally. To better grasp these constructs, we shall take an example of an AI-based conversational assistant (Employee AI app) that caters to the queries of employees in a company. Employees are the end users or the moving entities from the standpoint of the AI application. Here are six key constructs that we shall make use of in the design:

- Context Dimension (CD): A Context Dimension is an attribute of the moving entity. In our example, the following could be relevant Context dimensions - Employee ID, Employee Base Location, Employee Role, Employee Department, GeoLocation, etc.
- Context Universe (CU): The context universe is a complete set of all context dimensions. We shall formally define a Context Universe as follows:

  $$CU = \{R_i: 1 <= i <= n\}$$

  R_i is a Context dimension belonging to a context universe of size n.
- Context Value (CV): A Context Value is a value that a particular context dimension could hold. A specific context dimension could have any number of distinct values. For instance, as per our earlier example, *"New York, NY"*, and *"Seattle, WA"* could be different Context values of the Context dimension 'Employee Base Location'. Formally, we define a Context Value as follows:

  $$R_i = \{ V_j: 1<=j<=m\}, \text{ where } 1<=i<=n$$

  Here, V_j is a specific context value that the context dimension R_i can hold. The size of the context dimension R_i is m.
- Context instance (CI): The Context Instance is a collection of tuples. Each tuple comprises a context dimension and a specific value that dimension could hold. The size of the context instance cannot exceed the size of the context universe, and a context dimension cannot repeat across tuples within a Context instance. Formally, we define a Context Instance as follows:

  $$CI = \{I_k, 1<=k<=n\}, \text{ where } I_k = <R_i, V_j>$$

  Here is an example of a context instance from our employee chat application:

  {

  " Employee Base Location": "Seattle, WA",

  " Employee Role": "Software Engineer"

  }
- Context Hashkey (CH): A Context Hash Key is a hash value of a specific Context Instance.
- Context Store (CS): The Context Store is a document data store that holds all context values of a moving entity. In our example, the Context Store holds the complete context of the employee.

B. Context Model

In this section, we shall detail the data structure of the Caching layer for an AI application that leverages LLMs. The cache data structure will have five elements at the minimum. This will help perform three kinds of matches: exact match, semantic match, and context-based matches of cached queries. The five elements are detailed below:

- Question_raw: This text field will hold the unstructured question as typed by the end user (or moving entity).
- Answer_raw: This is the final response produced by the LLM application for the input question.
- Question_embedding: This vector field holds the embeddings of the input question. This shall be utilized to find similar questions that semantically match new user queries.
- Context_HashKey_Query: This is a hash value of the context instance derived from the user query. The relevance of this element will be made clear in the next section (Algorithm). This context instance will include every context attribute relevant to the entity referred to in the query. This instance will not include the Geolocation context alone, which will be held separately.
- Context_Location_Query: This field is a Point datatype and will hold the entity's GeoLocation context value if it is referred to in the query.

C. Algorithm

We are now ready to review the algorithm of the context-based semantic caching solution. The algorithm uses the constructs detailed in the earlier section and involves the following core steps while handling the end-user query.

- Capture the entity's identity (through the relevant identifier), current geolocation, and the query text from the request. In our example, the identifier could be any natural key of the employee. A good choice would be the Employee Id.
- Convert the user query into a vector embedding.
- Detect the context dimensions referred to in the query.
- Extract the context values for each context dimension (detected in the earlier step) from the query payload or the context store for the specific entity. Prepare a context instance from the context values (all besides geolocation) along with the context dimensions. Maintaining a context store for all moving entities is a prerequisite.
- Prepare a context hash key from the context instance prepared in the earlier step.
- Perform a lookup on the caching layer based on the following attributes that we just derived:
    - Context hash key: This must be compared for an exact match when the context hash key is not null. In other words, this comparison is optional for queries that are context-free. For instance, the question, *'What is the capital of my country?'* is context-sensitive compared to the query, *'What is the capital of Canada?'*
    - Question_embedding: The query's vector embedding must be matched based on cosine distance or another relevant algorithm. The match must be greater than a preconfigured threshold.
    - Geolocation: The geo-location needs to be close to the geolocation of the entry in the cache. In other words, the Euclidean distance between the two points must be greater than the predefined threshold. Again, this comparison is relevant only when the input query is location-sensitive. Questions like *'restaurants around me'* and *'hotels near me'* are geolocation-sensitive queries.
- If the previous step yielded a match, the response to the input query is returned from the cache. If the lookup fails, the LLM is invoked to respond, and that response will be added to the caching layer (with the relevant keys) for subsequent queries. The flow of the above algorithm is depicted in Figure 2.

IV. Design

We are now ready to walk through the design of the context-based semantic caching solution. The core components of the design include the following:

A. Cognitive Gateway

As the name suggests, this component acts as a gateway to any touchpoint that is a source of a user-generated query. The touch point will likely be a UX widget that manages the conversation with the end user. The cognitive gateway performs the necessary pre-processing, core processing, and post-processing steps of the user request. Pre-processing includes implementing the algorithm's first five steps, detailed in the previous section. Pre-processing includes capturing details from the request payload, embedding generation, context detection, context extraction, and context hash key generation. During the pre-processing step, the gateway will interact with the other components (Context Store, Context detector) as appropriate. The core processing step involves working with the cache manager to perform a lookup. When the cache manager finds a match, the cognitive gateway returns the match as a response. When the cache manager cannot find a match, the cognitive gateway invokes the LLM and moves to the post-processing step. At the post-processing step, the LLM's response is written to the cache through the cache manager and returned to the caller.

B. Context Store

The context store is implemented on a document store or NoSQL database to hold the context values for each moving entity, i.e., the end user. The context store helps serve two functionalities. First, it asynchronously gathers the different entities' context values for the dimensions relevant to the application's universe. It maintains them in a document and indexes the document by the entity's unique identifier. In our example, the context store will hold one document for an employee, and the key will be the Employee Id. The document will contain the specific values for the employee's role, location, etc. When invoked by the cognitive gateway to

extract a particular set of context values for an entity, it shall retrieve and return the available context values from the document.

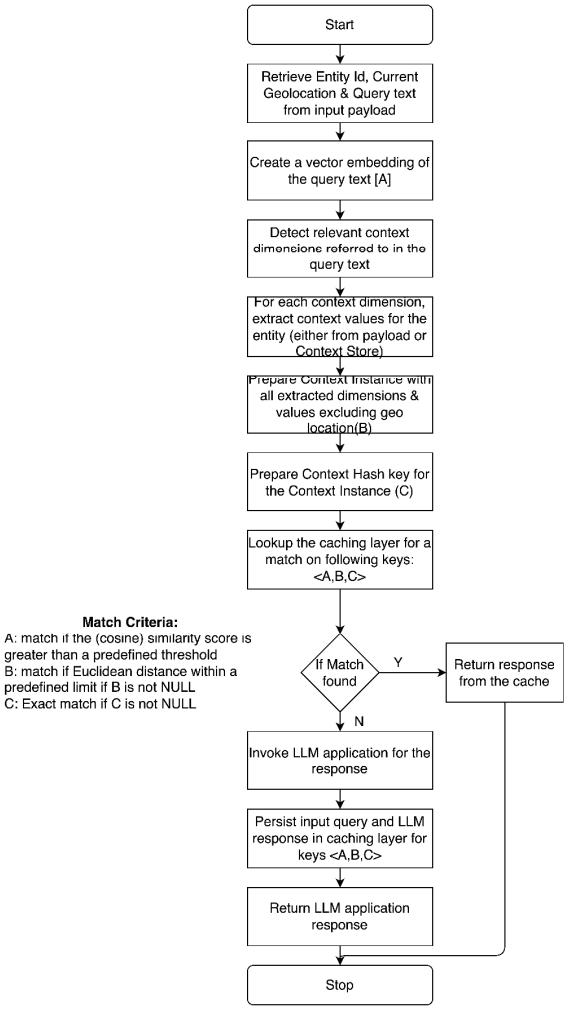

Fig. 2. Flow of a context-based semantic caching solution

C. Context Extractor

The context extractor is a multi-label classifier trained to identify context dimensions. It takes the user query as input and retrieves the context dimensions referenced in the query text. Here are a few examples from our AI application that caters to employee queries.

Example 1:

Query input: Find good restaurants near me

Output: {'Geolocation'}

Example 2:

Query input: Find details of colleagues from my department near me

Output: {'Employee Department', 'Geolocation'}

D. Cache Manager

The cache manager interfaces the cognitive gateway and the caching data layer. Apart from the core functions (cache eviction, time to live), the cache manager supports two key functions that the cognitive gateway will invoke: lookup and write. The cognitive gateway will invoke the lookup function to check if the input query can be retrieved from the cache. The lookup function takes two parameters as input. The first parameter is a *compound key*, a collection of two pairs. The first pair is the Geolocation context dimension and its corresponding value (lat-long). The second pair is the context hash key of all dimensions (excluding Geolocation) referred to in the input query. Remember, the two values can be null if the input query is context-free. The second parameter is the raw input query text. Here are the parameters passed by the cognitive gateway to the cache manager for the two queries mentioned in the above section:

Example 1:

Query input: Find good restaurants near me

Input to cache manager:

{

'Key': {

'Geolocation':<lat long of the entity>,

'Context Hash Key':null

},

'Query' : 'Find good restaurants near me'

}

Example 2:

Query input: Find details of colleagues from my department near me

Input to cache manager:

{

'Key': {

'Geolocation':<lat long of the entity>,

'Context Hash Key':<context hash key of the Employee Department dimension>

},

'Query': 'Find details of colleagues from my department near me'

}

The write function will be invoked when the cognitive gateway must invoke the LLM. The write function takes the *compound key* along with the response from the LLM for the cache manager to persist it in the data layer for subsequent requests.

E. Cache Data Layer

The cache data layer is any database, preferably an in-memory database platform, that supports the datatypes part of the cache data structure articulated in the earlier section. Here is the complete list of datatypes that need to be supported by the platform: text, vector embedding, dictionary, and point datatypes (for geolocation).

F. LLM

LLMs are either Large Language Models (open or closed) or applications that leverage a Large Language Model to process user-generated queries. The complete design is shown in Figure 3.

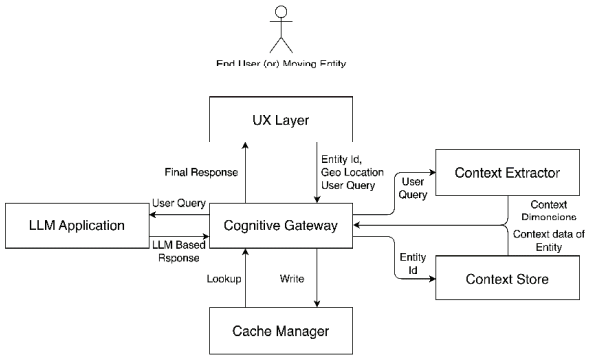

Fig. 3. Context-based semantic caching design for LLM applications

V. Experimental Results

In this section, we will capture the experimental details of the Context-based semantic caching solution. Through the experiments, we will quantify both the benefits and the tradeoffs involved.

- Benefits will be measured through Performance Testing, where we will show the reduction in average response time of an LLM application that has the Context-based semantic caching solution implemented.
- Every performance enhancement will have a cost associated with it and we will discuss this detail i.e., tradeoffs in the later part of this section.

A. Performance Testing

We will Performance Test our design on a dataset with two sets of questions. The first set comprises a set of distinct queries, and the second set is a collection of questions similar to the first set. In other words, every question from the second set is semantically different but means the same as one of the questions in the first set. The second set of questions was programmatically created for this experiment using a Generative Open AI model (GPT 3.5). The two sets are shuffled and shared as input to two instances of the AI application. In the first instance, the AI application invokes the LLM for each question, i.e., no caching is enabled here. The second instance of the AI application leverages the caching solution for a match before formulating the response. For a cache hit to succeed, the similarity score must be over a pre-configured threshold. The similarity score is the cosine similarity between the input string's embedding and the embedding of the cached query. For each instance execution, we will calculate the average response time. A lower response time would indicate two things:

1. The LLM was invoked on fewer occasions.

2. The response time was less when retrieved from the cache.

The outcome of the data experiment on a test size of 10,000 queries is shown in Figure 4. In the enhanced solution, the latency due to an LLM call is replaced with the following additional steps introduced by our caching solution: context extraction, context value lookup, hash key generation, and performing the cache update (and lookup). Even with these many steps, we clearly see an 80% reduction in the average response time on average. This is shown in Figure 4. This indicates that the AI application's response is far better (quicker response) and less expensive (fewer calls to LLM) with the implementation of the context-based semantic caching solution.

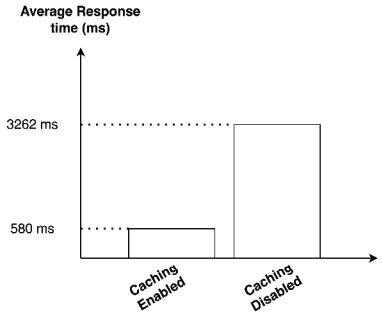

Fig. 4. Average Response time for user queries

B. Cost of Quality

Reducing the average response time doesn't necessarily mean the caching solution works holistically. We must ensure that performance improvement does not come at the cost of quality. To quantify quality, we performed a subtle variation of the same experiment. The change involved how we handled our original data set. The first set of questions in the input data set is now split (60:40) into two sections. The first section (60% subset) is marked for caching, and the second section (40% set) is marked not to be cached. The AI application is now made to process the entire data set. The expected outcome is that, as the AI application processes the first set of distinct questions, it would cache the ones marked for cache, not the remaining ones from the first set. With this, when the AI application starts processing the second set of similar questions, it is expected to either leverage the cache or invoke the LLMs to find the response for questions that did not find a match. The idea is to check for occurrences of false positives or negatives. Three instances of this experiment with varying thresholds for similarity scores were performed on a test size of 10,000 queries. The results of these experiments are captured across three confusion matrices below.

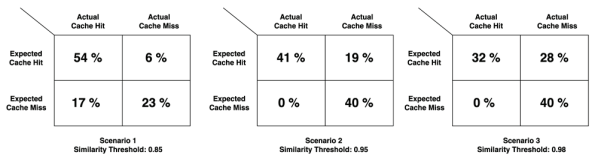

Fig. 5. Confusion Matrices

If the caching solution works to perfection, we will find these values (false positives and false negatives) to be zero in the confusion matrix. As you might see in the confusion matrices, the choice of threshold is crucial to achieving a balance, and our focus should always be to prioritize avoiding false negatives compared to avoiding false positives. As shown in the above figure, setting a similarity threshold of .95 (Scenario 2) helped achieve that balance. In scenario 2, the AI application correctly found a match from the cache for 67% of the scenarios. In 33% of scenarios, it invoked the LLM,

although the cache had a semantically similar record. The good news, however, is the second row of the confusion matrix. While a threshold of .95 led to occurrences of false positives, it did prevent the occurrence of false negatives completely. In other words, the AI application did not incorrectly identify a match in the cache even once.

For the above experiment, we stored the context details of the moving entities in a MongoDB database instance. The number of context dimensions extracted from the queries were between 1 and 10 and the size of the Context Universe was 100.

VI. Potential enhancements

The crux of context-based semantic caching solution is manufacturing a structured counterpart (context) of the unstructured input query and caching that combination (context + query) as a key in the caching layer. There are different ways this can be extended. One possible extension would be storing a modified version of the original query with the context values embedded also in the cache. For example, when someone asks, *'Good Thai restaurants in my hometown'*, we can also cache '*Good restaurants in Milwaukee*', where '*Milwaukee'* is the resolved context value for the context dimension '*hometown*' for the user.

VII. Conclusion

AI applications, i.e., conversational assistants that use LLMs to respond to user queries, can benefit from a caching solution, owing to the high cost and latency they are likely to incur. Caching user queries that involve unstructured text needs a different approach from traditional caching solutions. While introducing a semantic layer between the AI application and the LLMs works to some extent, they break when the user queries are context-sensitive. In many scenarios, end users will query from their point of view, and it is only natural for the user to expect the system to be able to discover the context before the query gets processed. The context based semantic caching solution detailed in this article addresses this white space of caching responses to context-sensitive questions. Besides managing the user-generated text as a vector embedding, the semantic caching layer has additional steps to resolve and persist the end-user context. Even with these steps, we observed the context-based semantic caching solution to reduce the average response time by over 80% without compromising the quality of the responses, as captured in the data experiment. To summarize, we see clear evidence of value created by the context-based semantic caching design when augmented by an LLM-based AI application.

References

[1] Bang Fu; Di Feng. 2023. GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings. In proceedings of the 3rd Workshop for Natural Language Processing Open-Source Software (NLP-OSS). Singapore

[2] Luís Leira; Miguel Luís; Susana Sargento. 2022. Context-based caching in mobile information-centric networks. Elsevier Computer Communications (volume 193 Sep 2022). https://doi.org/10.1016/j.comcom.2022.07.017

[3] Qun Ren; Margaret Dunham; Vijay Kumar. 2003. Semantic Cach-ing and Query Processing. IEEE Transactions on Knowledge and Data Engineering (Feb 2003). Doi.org/ 10.1109/TKDE.2003.1161590

[4] David lee. 2023. Speed Up LLMs Using a Semantic Cache Layer with SingleStoreDB. https://www.singlestore.com/blog/speed-up-llms-usinga-semantic-cache-layer-with-singlestoredb/. Accessed Nov 29 2023

[5] Hui Ding; Aravind Yalamanchi; Ravi Kothuri; Peter Scheuermann. 2006. QACHE: Query Caching in Location-Based Services. 10.1007/3-540- 35589-8_7

[6] Milan Cvitkovic; Badal Singh; Anima Anandkumar. 2018. Open Vocabulary Learning on Source Code with a Graph-Structured Cache. arXiv:1810.08305

[7] Viktor Sanca; Manos Chatzakis; Anastasia Ailamaki. 2023. ContextEnhanced Relational Operators with Vector Embeddings. arXiv:2312.01476v1

[8] Gyeongdo Ham; Seonghak Kim; Suin Lee; Jae-Hyeok Lee; Daeshik Kim. 2023. Cosine Similarity Knowledge Distillation for Individual Class Information Transfer. arXiv:2311.14307v1

[9] Sudip Mittal; Anupam Joshi; Tim Finin. 2017. Thinking, Fast and Slow: Combining Vector Spaces and Knowledge Graphs. arXiv:1708.03310v2

[10] Felipe Almeida and Geraldo Xexéo. 2023. Word embeddings: A survey. arXiv:1901.09069v2

[11] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways. arXiv:2204.02311v5

[12] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive language models can be accurately pruned in one-shot. arXiv:2301.00774.

[13] Dongwon Lee and Wesley W. Chu. 1999. Semantic caching via query matching for web sources. In Proceedings of the Eighth International Conference on Information and Knowledge Management, CIKM'99, page 77–85, New York, NY, USA. Association for Computing Machinery.

[14] Miti Mazmudar, Thomas Humphries, Jiaxiang Liu, Matthew Rafuse, and Xi He. 2022. Cache me if you can: Accuracy-aware inference engine for differentially private data exploration. arXiv:2211.15732

[15] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288