InpoCache: Indexed Prompt Caching for Efficient LLM Query Serving

Muhammad Ghifary Komara Putra - 13523066
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: m.ghifary.k.p@gmail.com , 13523066@std.stei.itb.ac.id

***Abstract*—Large Language Models (LLMs) are the center of many modern AI applications, such as chatbots and virtual assistants. However, their use often incurs high latency and computational costs due to redundant API calls for semantically similar queries. To address this inefficiency, we propose InpoCache, an indexed prompt caching system designed to reduce the lookup latency of LLM responses using lightweight indexing schemes, including binary search trees (BSTs) by query length, top-n k-means clustering based on embeddings, and K-D trees. Experimental results across multiple datasets demonstrate that the best-performing method of InpoCache, that is the K-D-tree based indexing, consistently records the lowest average latency while maintaining high accuracy with no false positive results, that is, 93.83 ms with 97.33% accuracy on a dataset with 1503 entries. Furthermore, compared to unindexed sequential search, all proposed indexing schemes significantly reduce lookup time while maintaining a high accuracy, no lower than 90%.**

Keywords—binary search tree, indexing, k-d tree, k-means clustering, prompt caching, sequential search

I. Introduction

Large Language Models (LLMs) have become an important part of modern AI applications due to their ability to comprehend and generate human-like text. They are used in several systems such as chatbots, virtual assistants, and customer support systems to provide relevant responses given some user queries. Despite the advancements of LLMs, a significant challenge arises from the need to make individual API calls to the LLM for each user query. This process can be costly and time-consuming [1]. Beyond that, it can be redundant for cases where the LLM deals with large volumes of similar and repetitive semantically similar questions. Picture a programming assistant chatbot. A user might ask similar questions to the chatbot over time, e.g. “How to push to a different branch to git?” or “How to center a div inside another div?”.

To address the inefficiency, various methods have been proposed to cache LLM responses, such as developing more reliable semantic embedding methods and improving the efficiency of decoding via speculative sampling [1, 2, 3]. The aforementioned work focuses more on embedding and token generation methods. On the other hand, InpoCache will complement those works by focusing more on reducing latency in the cache lookup process. This is done by designing an indexed prompt caching system based on the query and its embedding vector.

We propose InpoCache, an indexed prompt caching system that stores historical LLM queries, its embedding vectors, and corresponding responses in several simple and lightweight indexing schemes rather than naively inserting it into the cache. This means that we can perform cache lookup in a more time-efficient manner while still maintaining the accuracy of the resulting cached responses. This paper will explore three main indexing schemes, that are indexing by a binary search tree (BST) based on query length, indexed by top-n k-means clusters, and indexed by a K-D Tree based on the embedding vector of the query. This approach hopes to achieve better average lookup latency compared to sequential search on an unindexed cache or, in the worst case, achieve results no worse than that baseline.

II. Theoretical Framework

A. Indexing

Indexing in the context of Database Management System (DBMS) refers to the process of creating a copy of specific columns or fields from a database and organizing them into a separate structure, making the process of searching and retrieving data in databases quicker and more efficient. Several types of such structures include tree-like data structures and hash-like data structures. Indexing is used on tables with a high volume of data and frequent access patterns, making it a worth trade-off between lookup latency and overhead computation of inserting new entries into the database [4].

B. Prompt Caching

Prompt caching is an optimization technique used in Large Language Model (LLM) applications to temporarily store frequently used information between API calls and the model provider. Prompt caching reduces the cost and latency of LLMs, making it more efficient in handling repetitive content. This is achieved by evaluating similarity between user input queries and the information stored in the cache and reusing similar information to reduce the need of process identical queries multiple times [5]. A more detailed process on prompt caching mechanism can be presented as a flowchart in Fig. 1 below.

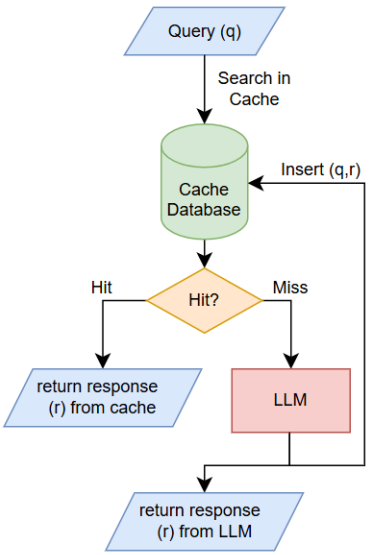

Fig 1. LLM Prompt Caching Flowchart

Other than reducing cost and latency, prompt caching also provides several other benefits, including scalability, resource and energy efficiency, security and privacy, and enhanced user experience. Several real-world LLM applications that often utilize prompt caching include conversational agents, coding assistants, and large document processing [5].

C. *Sequential Search*

Sequential search is a basic brute force searching algorithm that works by comparing every element in a dataset (array, database entries, etc.) with the value to be searched. This searching algorithm provides a simple and direct way of searching for an element inside a given dataset, making it often used as a performance benchmark for more efficient searching algorithms. With its O(n) time complexity, it is also still an efficient searching algorithm for smaller datasets [6].

D. *Binary Search Tree*

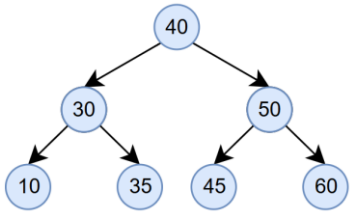

Fig 2. Balanced Binary Search Tree

Binary search tree (BST), also known as ordered binary tree, is a tree-based data structure in which each node has no more than two child nodes, where each child is either a leaf or the root of another binary search tree. The left subtree contains nodes with values less than the parent node while the right subtree contains nodes with values greater than the parent node. A more improved version of BST is the balanced binary search tree, where all the nodes are evenly distributed throughout the tree [7].

A balanced binary search tree has a unique property, that searching for an element inside the tree can be done in a binary-search-like fashion, making it an efficient searching method with O(log n) time complexity, far better than sequential search on a large dataset. This is done by only searching the promising child node while skipping the other, thus halving the number of elements to be searched on each iteration. Notice that this is essentially a decrease-and-conquer algorithm, where the problem is reduced to several smaller subproblems (decrease) and only some of them are processed to obtain the solution (conquer) [8].

E. *K-Means Clustering*

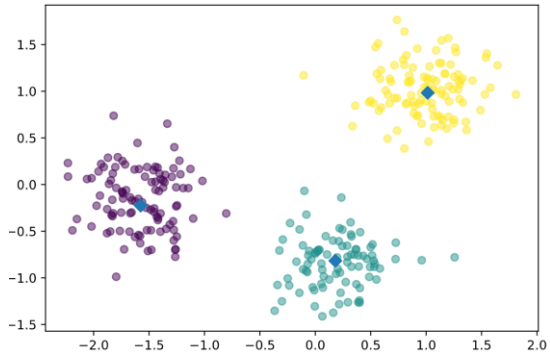

Fig 3. K-Means Clustering Visualization with circles as data points, different colored circles representing different clusters, and blue diamonds representing the clusters' centroid
Source: https://medium.com/@jwbtmf

K-means clustering is an unsupervised learning algorithm used for data clustering, which groups unlabeled data into clusters. It is a centroid-based clustering algorithm that partitions a dataset into similar groups based on the distance between the centroids, that is, the center of the cluster that can be obtained using the mean or median of all the points inside a cluster or other metrics relevant to the characteristics of the data. For further explanation, k-means clustering works by selecting the number of clusters the dataset will be divided into (k), creating the initial centroids based on some sampling methods, and assigns each data point to its closest centroid based on a distance metric (e.g. euclidean distance or cosine similarity) and updating the centroids on each iteration [9].

F. *K-D Tree*

K-dimensional tree, also known as K-D tree, is a space-partitioning data structure for organizing data points in a k-dimensional space. It functions similarly to a binary search tree with each node representing data in a multidimensional space. The data structure was developed by Jon Bentley in 1975 as a method to store spatial data with accomplishing three main criteria: nearest neighbor search, range queries, and fast lookup [10]. Insertion in K-D tree, using two-dimensional space (x, y) for simplification, works as follows:

1. Traverse the K-D tree from the root node at depth 0.
2. If we are on an even-depth node, compare the x-value of the current root node and the inserted value. If the x-value of the inserted value is higher, continue the

lookup to the right subtree. Otherwise, continue the lookup to the left subtree.

3. If we are on an odd-depth node, compare the y-value of the current root node and the inserted value. If the y-value of the inserted value is higher, continue the lookup to the right subtree. Otherwise, continue the lookup to the left subtree.
4. Repeat step 2 and 3 until there's no more subtree to evaluate. The inserted value becomes a new leaf node inside the K-D tree

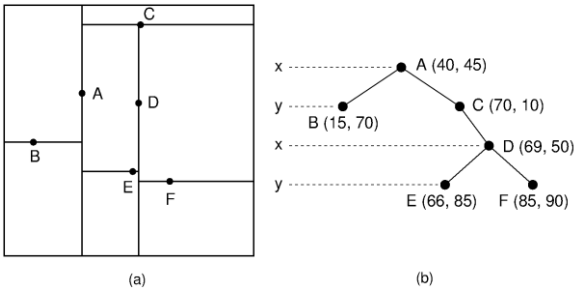

Fig 4. K-D Tree Visualization

Source: opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/KDtree.html

Searching in K-D tree also works in a similar way to a binary search tree, that is by traversing the tree and comparing x-value on even-depth nodes or the y-value on odd-depth nodes. This provides a decrease-and-conquer approach of searching, much more efficient than sequential search, with time complexity ranging from O(log n) to O(n) depending on the resulting K-D tree [10].

G. *Confusion Matrix*

A confusion matrix, also known as an error matrix, is a method for comparing the classification predicted by a system with the actual classification results. It is a metric to evaluate the performance of a classification model. In a confusion matrix, results are divided into four categories: true positive (a positive case that is correctly predicted), true negative (a negative case that is correctly predicted), false positive (a negative case that is incorrectly predicted as positive), and false negative (a positive case that is incorrectly predicted as negative) [11]. This provides a metric that shows how a classification model behaves when making predictions. It doesn't just indicate the number of errors, but also the types of errors the model makes.

III. INPOCACHE

This section describes how InpoCache works, focusing mainly the indexing schemes and lookup algorithms that are being developed and are being explored. These schemes and algorithms include no indexing with sequential best-first search, query-length-based indexing with adjusted binary best-first search on balanced BST, query-embedding k-means clusters indexing with top-n clusters sequential search, and K-D-tree-based-indexing with K-D tree search.

A. *No Indexing with Sequential Best-First Search*

Caching with no indexing and sequential best-first search acts as a benchmark for other caching schemes. A better caching scheme should, in theory, perform faster cache lookup on average than this method. This method returns the first entry from the cache that is semantically similar to the given query that exceeds a cosine similarity threshold. Notice that this method will not always return the best query from the cache since there's no guarantee that the first found entry is the best one in the entire cache. It can also be represented in the pseudocode shown in Algorithm 1.

Algorithm 1. Sequential Best First Search

```
Input : embedding_model, cache_data, query, threshold
Output: best-first query from cache (if exists)

query_embedding <- embedding_model.encode(query)
For each entry in cache_data do
  sim <- cosine_similarity(query_embedding,
                           entry.embedding)
  if sim >= threshold then
    return entry.question
return Null
```

B. *Query-Length-Based Indexing with Adjusted Binary Best-First Search on Balanced BST*

Caching with query-length-based indexing and adjusted binary best-first search on balanced BST works by storing the cache based on the length of the query and constructing a balanced BST of query lengths based on the cache dataset. Cache lookup is performed by traversing the BST in a binary search manner. If there is no cache hit by the end of the binary search, the algorithm backtracks to the parent node and continues the searching on the other subtree. This ensures that the search method achieves accuracy close to sequential search while maintaining a lower average latency. The search algorithm is also be represented in the pseudocode shown in Algorithm 2.

Algorithm 2. Adjusted Binary Best-First Search on Balanced BST

```
Input : embedding_model, sorted_cache_data, query, tree,
threshold
Output: best-first query from cache (if exists)

query_embedding <- embedding_model.encode(query)
query_length <- length(query)
stack <- [tree]

while stack is not empty do
 node <- stack.pop()
 if node is Null then
   continue

 left <- node.start_index
 right <- node.start_index + node.count
 if query_length = node.value then
   visited_equal_node <- true
   for each entry in sorted_cache_data[left:right]do
```

```
      sim <- cosine_similarity(query_embedding,
                               entry.embedding)

     if sim >= threshold then
       return entry

  if query_length > node.value then
    stack.push(node.left)
    stack.push(node.right)
  else
    stack.push(node.right)
    stack.push(node.left)

  visited_equal_node <- (node.value = query_length)
  if not visited_equal_node and query_length !=
                                node.value then
    for each entry in sorted_cache_data[left:right]do
      sim <- cosine_similarity(query_embedding,
                               entry.embedding)
      if sim >= threshold then
        return entry.question

return Null
```

C. *Query-Embedding K-Means Clusters Indexing with Top-n Clusters Sequential Search*

Caching with query-embedding k-means clusters indexing with top-n clusters sequential search works by storing the k-means clusters and centroids for the cache dataset and doing the cache lookup using sequential search only within clusters whose centroids are semantically similar to the given query. Though it retains the O(n) time complexity, on a small to medium sized dataset, this method aims to perform better than non-indexed sequential search. The search algorithm is also represented in the pseudocode shown in Algorithm 3.

Algorithm 3. Top-n Clusters Sequential Search

```
Input  : embedding_model, cache_data, query, centroids,
cluster_map, threshold, top_n_clusters
Output : best-first query from top-N clusters (if
exists)

 1:  query_embedding <- embedding_model.encode(query)
 2:
 3:  centroid_sims <- empty list
 4:  for each centroid in centroids do
 5:      sim <- cosine_similarity(query_embedding,
                                  centroid)
 6:      append sim to centroid_sims
 7:
 8:  ranked_clusters <- indices of centroids sorted by
                        similarity descending
 9:  top_clusters <- first top_n_clusters elements of
                  ranked_clusters
10:
11:  for each cluster_id in top_clusters do
12:    for each row_index in cluster_map[cluster_id] do
13:      row <- cache_data[row_index]
14:      sim <- cosine_similarity(query_embedding,
                                  row.embedding)
15:      if sim >= threshold then
16:        return row.question
17:
18: return Null
```

D. *K-D-Tree-Based-Indexing with K-D Tree Search*

Caching with K-D-tree-based-indexing with K-D tree search works by generating a K-D tree of the cache dataset and performing the cache lookup based on said tree. This provides a time complexity ranging from O(log n) to O(n), depending on the resulting K-D tree. The searching algorithm is also represented as the pseudocode shown in Algorithm 4.

Algorithm 4. K-D Tree Search

```
Input  : embedding_model, cache_data, query, kdtree,
threshold
Output : nearest query from cache (if within distance
threshold)

 1:  query_embedding <- embedding_model.encode([query])
 2:  distances, indices <- kdtree.query(query_embedding)
 3:
 4:  best_distance <- distances[0][0]
 5:  best_index <- indices[0][0]
 6:
 7:  if best_distance > threshold then
 8:    return Null
 9:
10:  best_row <- cache_data[best_index]
11:  return best_row.question
```

IV. EXPERIMENTS

This section describes the experiments conducted to test the performance of different indexing schemes and search algorithms in InpoCache. We first introduce the datasets used to simulate the cache and the preprocessing appliedto those datasets in Section A. Tthen, in Section B, we present the testing scheme, including the performance metrics that are being used and how the testing dataset was developed. Furthermore, the results of the experiments are shown and discussed in Chapter V.

A. *Cache Dataset Collection and Preparation*

This experiment uses two main datasets, that are a chatbot question-answer dataset [12] and a scientific question-answering dataset [13], commonly used for developing chatbot and information retrieval systems, respectively. The question-and-answer pairs from these datasets are stored in a new CSV file, simulating the cache. To optimize the caching system, additional information is added to each entry of the cache, that is the length of the question, the embedding vector of the question, and the k-means cluster id of the question's embedding vector. This paper utilizes embedding system from SBERT due to its reliability and open-source nature.

Futthermore, to evaluate InpoCache's performance on datasets of different sizes, the final simulated cache includes the chatbot dataset (503 entries), scientific question-answering dataset (1000 entries) and combined dataset (1503 entries). The resulting dataset can be accessed via the Github repository listed in Appendix A.

B. Testing Scheme

There are two main evaluation metrics used to assess the performance of InpoCache. The first metric is latency or average lookup time. An acceptable indexing scheme should perform no worse on average than an unindexed sequential search. The second metric is accuracy, which refers to whether the caching system returns the correct cache hit or miss given a query. Specifically, accuracy is represented using a confusion matrix which informs the percentage of true positives, true negatives, false positives, and false negatives. In this context, a false positive means an incorrect result due to returning an unrelated cache entry, while a false negative means a relevant entry was not returned. This distinction is important because, in prompt caching systems, we may tolerate a missed hit (false negative) more than an incorrect hit (false positive). Therefore, a higher accuracy rate and lower false positive and false negative rates define the quality of InpoCache's caching system.

To ensure the caching system performs well on a broad range of query types, the testing dataset is divided into three main categories: exact matc, semantically matc, and random datasets. The exact match dataset is taken from samples of the cache dataset, using entries from evenly spaced indices to generate a fair benchmark for the unindexed sequential search method. The semantic match dataset contains queries with semantically similar meaning to those in the exact dataset. Finally, the random dataset consists of random unrelated queries to evaluate whether the cache incorrectly returns a result. Both the semantic and random match datasets are generated synthetically. The resulting datasets can be accessed via the Github repository listed in Appendix A.

The testing is conducted by measuring the time taken and the accuracy of each cache dataset with its corresponding exact match, semantic match, and random test datasets. For each query from the test dataset, the searching is performed five times to account for performance variability across iterations, resulting in an average latency that accurately reflects it actual performance. The testing uses a 0.9 cosine similarity threshold and 0.5 euclidean distance threshold for the K-D Tree.

The implementation is done in Python and Jupyter Notebook due to their extensive support for data processing libraries. The source code for the indexing and algorithm implementation, as well as the testing scheme and results, can be accessed via the Github repository listed in Appendix A.

V. Results and Analysis

A. Results

Based on the datasets and testing scheme in Chapter IV, the results of the experiments are shown on Table 1. TP, TN, FP, and FN are notations for true positive, true negative, false positive, and false negative respectively. Furthermore, method A, B, C, and D represents indexing schemes and search algorithms in the same order as presented in Chapter III.

Table 1. Experiment Results

Dataset	Method	Latency (ms)				Accuracy (%)				
		Average	Min	Max	Std.	TP+TN	TP	TN	FP	FN
Chatbot	A	163.28	63.78	274.05	56.48	98.18	36.36	61.82	1.82	0
	B	157.85	52.22	273.70	76.52	100.00	28.18	61.82	0	0
	C	118.19	59.13	172.49	22.69	96.36	34.55	61.82	1.82	1.82
	D	107.78	52.25	173.42	34.61	100.00	38.18	61.82	0	0
Scientific Question Answering	A	233.76	87.58	320.35	55.10	93.33	50.67	42.67	0	6.67
	B	196.38	60.10	342.83	108.25	93.33	50.67	42.67	0	6.67
	C	124.00	66.43	108.25	26.47	92.00	49.33	42.67	0	8.00
	D	103.88	49.96	26.47	26.45	96.00	53.33	42.67	0	4.00
Combined	A	299.21	99.19	398.92	69.57	91.33	38.67	52.67	1.33	7.33
	B	271.98	89.39	422.50	114.05	92.67	40.00	52.67	0	7.33
	C	132.92	78.82	196.36	26.27	90.00	37.33	52.67	1.33	8.67
	D	93.83	41.51	166.46	33.79	97.33	44.67	52.67	0	2.67

B. Latency Analysis

Based on the results shown in Table 1, across all datasets, method D, which is K-D-tree-based-indexing with K-D tree search, consistently achieves the lowest average latency. For instance, in the Chatbot dataset, it records an average latency of 107.78 ms, outperforming all other methods. This trend holds in both the Scientific Question Answering dataset and the Combined dataset, showcasing the scalability of this method. Furthermore, method D produces more consistent results across datasets of different sizes, unlike other methods which show an increased average latency on larger datasets.

As a side note, method C, which is query-embedding k-means clusters indexing with top-n clusters sequential search, demonstrates the lowest standard deviation of latency across datasets, indicating more stable and consistent performance, albeit with a slightly higer average latency than method D.

It is also important to note that every proposed indexing scheme and search algorithm for InpoCache, namely method B through D, performs better than the benchmark (method A) in terms of latency across all datasets, making them all acceptable and effective caching systems.

C. Accuracy Analysis

Based on confusion matrix accuracy shown in Table 1, across all datasets, method D, which is K-D-tree-based-indexing with K-D tree search, achieves the highest or near-highest accuracy scores. it reaches 100% accuracy alongside method B on the Chatbot dataset, 96% accuracy on the Scientific Question Answering Dataset with no false positives, and 97.33% accuracy on the Combined dataset, also with no false positives.

It is also important to notice that all proposed indexing scheme and search maintain highly acceptable accuracy across all datasets, with the lowest accuracy recorded at 90%. Furthermore, each method tends to return false negatives rather than false positives, which aligns with the intended behavior of the caching system. Additionally, the few false positives returned during the cache lookup are still considered acceptable, as the retrieved query remain semantically similar to the intended one, as illustrated in Appendix B.

VI. Conclusion

In this paper, we introduce InpoCache, an efficient prompt caching system for Large Language Models that leverages lightweight indexing techniques to improve cache lookup performance. Unlike prior work that focuses on embedding generation and token decoding efficiency, InpoCache addresses the bottleneck of cache retrieval latency.

Through experiments on several datasets. We found that all three proposed indexing methods, which are query-length-based binary search, embedding-based top-n k-means clustering, and K-D-tree-based indexing, outperforms the sequential search baseline in terms of latency. Among said methods, the K-D-tree-based indexing consistently demonstrated the lowest average latency across datasets while achieving high accuracy and zero false positives in most cases.

Moreover, the observed false negatives were more common than false positives, which is a favorable property for caching systems that prioritize precision. Even in cases of false positives, the retrieved responses remained semantically relevant.

In summary, InpoCache provides a practical and effective solution for accelerating LLM-based applications. Future work may focus ondeveloping a ready-to-deploy pipeline for InpoCache, exploring more sophisticated indexing methods such as vector databases, or designing an entirely new indexing schemes or embedding techniques.

VII. Appendix

A. Appendix A

The cache datasets, testing datasets, and the full source code of the experiment can be found on this Github repository:

https://github.com/Nuetaari/InpoCache

B. Appendix B

The following figure shows the false positive results retrieved by InpoCache (represented by actual result).

Query:	How does the local beam search operate?
Intended Result:	How does the local beam search operate?
Actual Result:	What is a local beam search?
Query:	What is the distinction of the search tree from the state space?
Intended Result:	What is the distinction of the search tree from the state space?
Actual Result:	What is the distinction of the state space from the search tree?
Query:	How is a search tree different from the state space?
Intended Result:	What is the distinction of the search tree from the state space?
Actual Result:	What is the distinction of the state space from the search tree?

Acknowledgment

The author of this paper would like to express his gratitude to Allah Swt. for His blessing and guidance throughout the experiment and writing process of this paper. Furthermore, the author would also like to thank the course's (IF2211) lecturers, mainly Dr. Ir. Rinaldi Munir, M.T., who provided abundant learning resources and opportunities for his students, including the author. Lastly, the author would like to thank his family for their unconditional support and encouragement for the author throughout his academic journey.

References

[1] Sajal Regmi et al., "GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching," in https://arxiv.org,

December 9, 2024. [Online]. Available: https://arxiv.org/abs/2411.05276 [Acessed: June 24, 2025]

[2] Hanlin Zhu et al., "Efficient Prompt Caching via Embedding Similarity," in https://arxiv.org, February 2, 2024. [Online]. Available: https://arxiv.org/abs/2402.01173 [Accessed: June 24, 2025]

[3] Luis Gaspar Schroedder et al., "Adaptive Semantic Prompt Caching with VectorQ," in https://arxiv.org, May 27, 2025. [Online]. Available: https://arxiv.org/abs/2502.03771v3 [Accessed: June 24, 2025]

[4] Jeremy Savage, "An Introduction to Database Indexing," in medium.com, May 17, 2025. [Online]. Available: https://medium.com/@jwcsavage/an-introduction-to-database-indexing-138ac99d4b83 [Accessed: June 24, 2025]

[5] Conor Kelly, "Prompt Caching: Reducing latency and cost over long prompts," in humanloop.com, October 2, 2024. [Online]. Available: https://humanloop.com/blog/prompt-caching [Accessed: June 24, 2025]

[6] Rinaldi Munir, "Algoritma Brute Force (Bagian 1)," IF2211 Strategi Algoritma – Semester II Tahun 2024/2025, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/02-Algoritma-Brute-Force-(2025)-Bag1.pdf [Accessed: June 24, 2025]

[7] Gabriel Batista, "A Look Into Binary Search Trees," in medium.com, December 2, 2018. [Online]. Available: https://medium.com/data-science/a-look-into-binary-search-trees-ee2d69e9d0ef [Accessed: June 24, 2025]

[8] Rinaldi Munir, "Algoritma Decrease and Conquer (Bagian 1)," IF2211 Strategi Algoritma – Semester II Tahun 2024/2025, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/11-Algoritma-Decrease-and-Conquer-2025-Bagian1.pdf [Accessed: June 24, 2025]

[9] Ada Kavlakoglu and Vanna Winland, "What is k-means clustering?," in ibm.com, June 26, 2024. [Online]. Available: https://www.ibm.com/think/topics/k-means-clustering [Accessed: June 24, 2025]

[10] Katyayani Vemula, "What is a K-Dimensional Tree?," in medium.com, January 24, 2024. [Online]. Available: https://medium.com/@katyayanivemula90/what-is-a-k-dimensional-tree-8265cc737d77 [Accessed: June 24, 2025]

[11] Kuncahyo Setyo Nugroho, "Confusion Matrix untuk Evaluasi Model pada Supervised Learning," in ksnugroho.medium.com, November 13, 2019. [Online]. Available: https://ksnugroho.medium.com/confusion-matrix-untuk-evaluasi-model-pada-unsupervised-machine-learning-bc4b1ae9ae3f [Accessed: June 24, 2025]

[12] YapWH, "Chatbot Dataset (AI Q&A)," in kaggle.com, 2023. [Online]. Available: https://www.kaggle.com/datasets/yapwh1208/chatbot-ai-q-and-a [Accessed: June 24, 2025]

[13] Crowdsource, "SciQ (Scientific Question Answering)," in kaggle.com, 2022. [Online]. Available: https://www.kaggle.com/satasets/thedevastator/sciq-a-dataset-for-science-question-answering?select=train.csv [Accessed: June 24, 2025]

STATEMENT

Hereby I declare that this paper that I have written is my own work, not a reproduction or translation of someone else's work, and not plagiarized.

Bandung, 24 June 2025

Muhammad Ghifary Komara Putra (13523066)