San Jose State University

SJSU ScholarWorks

Spring 2023

Robust Cache System for Web Search Engine Yioop

Rushikesh Padia
San Jose State University

Robust Cache System for Web Search Engine Yioop

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Rushikesh Padia

May 2023

The Designated Project Committee Approves the Project Titled

Robust Cache System for Web Search Engine Yioop

by

Rushikesh Padia

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Dr. Chris Pollett	Department of Computer Science
Dr. Ben Reed	Department of Computer Science
Dr. Robert Chun	Department of Computer Science

ABSTRACT

Robust Cache System for Web Search Engine Yioop

by Rushikesh Padia

Caches are the most effective mechanism utilized by web search engines to optimize the performance of search queries. Search engines employ caching at multiple levels to improve its performance, for example, caching posting list and caching result set. Caching query results reduces overhead of processing frequent queries and thus saves a lot of time and computing power. Yioop is an open-source web search engine which utilizes result cache to optimize searches. The current implementation utilizes a single dynamic cache based on Marker's algorithm. The goal of the project is to improve the performance of cache in Yioop. To choose a new caching system, Static-Dynamic cache along with its different variations Machine Learning Static-Dynamic Cache, Static-Semistatic-Dynamic Cache, and Static-Topic-Dynamic Cache were evaluated. Based on these experiments, Static-Topic-Dynamic was implemented in Yioop. Static-Dynamic cache exploits temporal locality by dividing cache into a static part which stores most popular queries and a dynamic part which captures the bursty behavior of queries. Static-Topic-Dynamic adds topical cache section in Static-Dynamic Cache which captures queries that are neither too popular to be in static cache nor too frequent to be in dynamic cache by creating dedicated cache for each topic. To extract topic from the queries, k-means algorithm was chosen as topic model. The results of Static-Dynamic Cache and Static-Topic-Dynamic cache showed the improvement of 2.3% and 1% over the initial performance of the cache.

Keywords: Yioop, Web Search Engine, Result Caching, Static-Topic-Dynamic Cache

ACKNOWLEDGMENTS

I would like to take this opportunity to express my gratitude to Dr. Chris Pollett, my advisor, for his guidance, support, and motivation throughout the year. His expertise and knowledge have been instrumental in shaping my research and academic career. I am also thankful to the members of my defense committee, Dr. Ben Reed and Ms. Batul Merchant, for their valuable insights and feedback on my research work. Additionally, I would like to extend my appreciation to the faculty members of the department for their support and encouragement throughout my graduate years. Finally, I want to thank my family, friends, and loved ones for their constant motivation and encouragement which have been a great source of inspiration to me.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

Introduction

With the billions of pages currently present on the internet and the number still rapidly growing, it is crucial to employ efficient caching mechanisms to search queries. The goal of caching is to reduce the response time by storing the results of previously executed operations. Most modern search engines cache data at multiple levels. Caching posting lists, posting intersections, and result sets [1] are the most common approaches used for caching in web search engines. Yioop uses result caching for caching for caching queries and the corresponding result. The goal of this project is to improve the performance of the result cache.

The main problem of result caching is detecting which query to be cached and which one to remove. These problems are called query admission problems and query eviction problems. Several approaches are available to determine admission and eviction of the query. E.g., LRU admits every query while it evicts a query that appeared least recently in the cache. Although LRU is one of the most widely used caching algorithms, it does not capture global long-term trends. To capture these trends, a static cache is often employed along with the dynamic cache [2].

The Static-Dynamic Cache (SD cache) is one of the most popular strategies used for caching search results. It divides the cache into two parts. The static part is a read-only part containing the most popular queries like "Facebook", "Reddit", "Wikipedia", etc. While the dynamic part is a read-write part that is managed using some admission and eviction policy for eg. LRU, LFU, etc. Having two separate parts allows this algorithm to capture both long-term and short-term trends in queries. Along with the Static-Dynamic Cache, there are Machine Learning Dynamic Cache, Static-Topic-Dynamic Cache, and Static-Semistatic-Dynamic Cache which have further improved the performance of the cache. Based on the experiments, Static-Topic-Dynamic Cache

is found to be the most suitable for caching in Yioop.

Static-Topic-Dynamic Cache (STD cache) is based on the SD cache having an additional partition for the topical cache. The framework is built on the underlying assumption that there are certain queries that are not sufficiently popular to be put in static cache neither they are too frequent to be put in dynamic cache. Topics in queries have different access patterns. Identifying the topic from the query can allow the cache to adapt to the temporal locality of the topic. In the topical cache, separate caches are created for different topics such as social media, education, etc. Each of these caches is an instance of a dynamic cache or SD cache.

Currently, Yioop uses a single dynamic cache based on Marker's algorithm to cache query results. Having a single dynamic cache for caching results has limited its ability to capture only short-term trends. To implement the STD cache in Yioop, static cache is populated periodically with most popular queries using Yioop's MediaJob. The dynamic part is implemented using LRU and Marker's algorithm for the dynamic part. To identify the topic from queries, the k-means algorithm [3] is used which is an unsupervised clustering algorithm. The k-means algorithm is trained using the dataset generated from search results from Yioop's web crawl. Over the course of this report implementation of new caching strategies is explained.

We now discuss the organization of the rest of the report. Chapter 2 gives the background of the Yioop, caching, and topic models. In Chapter 3, preliminary work done before the implementation is described. It walks through the different algorithms tested before finalizing the approach and finally, Chapter 4 discusses the implementation detail of caching in Yioop.

CHAPTER 2

Background

In this chapter, we will first discuss the related work done in this area and then the Yioop search engine, its important components, and query processing in Yioop. We will also discuss different caching techniques in search engines. Apart from result caching, search engines employ different caching strategies at different levels. Along with this, how topics are extracted from text will also be discussed.

2.1 Related Work

Improving search engine performance by using a result cache was proposed by Markatos [4]. In this work, the author evaluated the performance of result caching using various dynamic caching algorithms. The author also evaluated the static cache for small cache size. The results show static cache to be performing well when the cache size is small while dynamic cache to be performing better when the cache size is bigger. Fagni et al.[2] combined both static and dynamic cache in their work to create a Static-Dynamic Cache framework. In this framework, a cache was partitioned into two segments, a static part for read-only data derived from historical data and a dynamic part managed by different cache replacement algorithms. The results of this showed SD Cache to remarkably outperform any other individual static or dynamic caching algorithm.

There have been several studies that exploit different characteristics of web search engines and query patterns to improve the performance of the cache. Ozcan et al.[5] proposed a new feature query stability to be used for populating the static cache. The feature represented the stability of queries over a period of time. In [6] Ozcan et al. proposed several strategies for static and dynamic caches. They demonstrated that caching popular query does not necessarily improves the performance of the cache. Cache misses for different queries have a disproportionate impact on search

engine performance. Their strategies incorporated the actual cost of processing queries such as CPU and memory to populate the static cache and take admission eviction decisions in the dynamic cache.

Along with several individual static and dynamic caching algorithms, several authors added additional layers to the SD cache. Tayfun et al. [7] proposed a machine learning-based approach for result caching. They derived various features from historical query data to populate the static cache as well as a design strategy for admission and eviction in the dynamic cache. Tayfun [8] divided queries based on their frequency in different parts of the day and maintained a small section called semi-static cache which toggled between daytime and nighttime popular queries. The authors of [9], Mele et al. added a layer of topical cache in the SD cache. They used Latent Dirichlet Allocation (LDA) for topical classification. Their result shows the implementation to be achieving a higher hit rate than a simple SD cache algorithm.

2.2 Yioop Web Search Engine

Yioop is an open-source search engine that is designed to provide fast and efficient search results for users. Developed in PHP and licensed under the GNU License, Yioop is a powerful search engine that can index and search different types of data such as web pages, images, and documents. One of the key features of Yioop is its ability to support multiple languages, making it accessible to users from around the world. Yioop supports many different languages which ensure users can search for information in their preferred language.

For effective search, the Yioop search engine has three main components - Crawler, Indexer, and Query Processor. Yioop's crawler performs the critical task of discovering and gathering information from web pages. Crawler has a Queue server that manages URLs discovered and Fetcher servers that fetch web pages from the internet. The

indexer processes pages fetched by the fetcher to extract the textual content and builds an inverted index for processing queries. Along with this, it also generates a textual summary of the information extracted from the web page. The Query processor evaluates the query and produces search results. Search results are sorted based on a ranking algorithm and contain additional information like summary, title, and score.

The project mostly deals with the Query processor in Yioop. It evaluates query by first cleaning the query which includes case folding, stemming, stopword elimination, etc. Then the query is checked against the result cache whether the result has already been computed for the query. If results are available in the cache, results are returned without further processing. If results are not available, the processor fetches a posting list for each term in the query. The posting list contains a list of documents containing the term and is stored in the inverted index. Query processor uses a ranking algorithm to find the most relevant documents. Thus having an efficient cache can save additional cost of computing result set.

2.3 Caching In Web Search Engine

Caching is one of the most critical parts of web search engines which can create a significant impact on the response time of a search engine. Caching involves storing some data in memory or disk so that it does not have to be fetched or computed again. Typically cache memory is limited and therefore the space is governed by some admission and eviction policy. In web search engines, caching typically includes result caching, posting list caching, and cache pre-fetching.

Result caching is a widely used caching mechanism in web search engines. Once a user enters a query, the query processor cleans the query and then computes the search results. Computing the search result for each query is a costly task. Therefore,

search results are typically cached so that if the same query is entered again, the result will be pre-computed in the cache, and thus computation of the search result will be avoided. The caching result significantly reduces the response time of search engines.

Posting list caching is another caching utilized in web search engines. The posting list contains a mapping of terms and a list of documents and the positions of terms in the document. It is essential in efficiently searching for documents that contain specific terms. Caching these posting lists either in memory or on disk can significantly expedite the search process. This is because the posting lists can be quickly retrieved from the cache rather than having to be fetched from the search index.

Cache prefetching is a technique where a search engine predicts the incoming future queries and loads the cache with the relevant data even before it is requested. It exploits the spacial locality of the cache to proactively load future queries. This technique further reduces the latency of generating search results for users. Thus search engines optimizes its performance by applying caching strategies best for its use cases.

2.4   Topic Models

Topic modeling is a popular technique in natural language processing that aims to uncover the underlying themes and patterns present in a corpus of text data. It is an unsupervised learning technique, which means that it can automatically identify and extract the topics present in a corpus of text data without any prior knowledge or explicit guidance about the nature of the topics. There are several approaches to topic modeling, the two of the most widely used are Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA).

LSA is an unsupervised learning technique that identifies the latent semantic structure of a corpus. It represents each document as a high-dimensional vector. Each

dimension in the vector corresponds to a unique word in the vocabulary. LSA then applies singular value decomposition (SVD) to this matrix to reduce the dimensionality and identify the underlying topics or themes. One of the advantages of LSA is that it can be used to represent new, unseen documents in the latent semantic space, making it useful for document classification.

Latent Dirichlet Allocation (LDA) is another popular topic model used for document classification. It assumes each document is a mixture of a small number of topics and each topic is a mixture of a small number of words. LDA is a generative model which generates a document from given topic distribution but can its inverse mapping can be used to derive topics from a document. For training LDA, for each term in the document, a topic is randomly selected from the topics assigned to the document, and then a word is selected from the words assigned to the topic. After a sufficient number of iterations, LDA can estimate the underlying distribution of topics in documents.Although LSA and LDA both are excellent models for choice of topic model, they are not very intuitive and efficiently implementing both of them from scratch is a quite complex.

2.5 Topic Modeling using k-Means Algorithm

k-mean is a popular unsupervised learning algorithm that is widely used for clustering tasks in machine learning. However, k-means can also be used for topic modeling in natural language processing. To use k-means for topic modeling, documents are modeled as bag-of-words. Documents are converted into vectors using Count or Tf-Idf vectorizers. The k-means algorithm starts by randomly selecting data points in the vector space to initialize k centroids which represent a particular topic. Each document is then assigned to the nearest cluster using Euclidean distance. After one such iteration, centroids are updated with the mean of all document vectors in the

cluster. After a sufficient number of iterations, k-means converge to a set of centroids which becomes the final representative of the cluster. To predict the cluster for a new document, a cluster is chosen which is nearest to the document in the vector space. As this model is more intuitive and fairly easy to implement, this model is chosen for the project.

CHAPTER 3

Preliminary Work

This section provides a summary of the progress accomplished in the initial phase of the project. To improve the cache in Yioop, several important features of Yioop were studied. Along with this different caching algorithms like Machine Learning based Static Dynamic Cache, Static-Semistatic-Dynamic cache, and topical cache were evaluated. Their performance and limitations were considered to decide on a strategy to be used in the Yioop search engine.

3.1   Cache Refresh Media Job

Currently, Yioop refreshes its cache using an executable file. A user has to manually run this file to refresh the cache in Yioop. To automate cache refresh, CacheRefeshJob, a MediaJob is created which periodically refreshes the cache by querying the search engine using seed data. The seed data can be specified by adding a file containing queries to be cached in Yioop's "/cache/cache_queries" directory

As part of this task, Yioop's MediaJob was studied. Important classes and methods related to MediaJobs were analyzed. Yioop's MediaJobs are the activities that are run periodically by MediaUpdater in Yioop. MediaUpdater schedules MediaJobs based on their frequencies. There are several MediaJobs in Yioop like TrendingHighlightsJob to find periodically find trending topics, FeedsUpdateJob to keep the news section updated, etc. Important classes and methods of MediaJob are given in Table 1 and 2.

Table 1: Classes related to Media Job

Class	Description
MediaJob	It is a parent class for the Media Job to be executed. All jobs need to extend this class and override callback methods to execute their intended functionality
MediaUpdater	It is a class responsible for executing and managing the lifecycle of Media Jobs

Table 2: Methods of MediaJob

Method	Description
init()	Callback method called after constructor to initialize the job
checkPrerequisites()	Method that should return true if media job has to be executed, false otherwise
nondistributedTasks()	Method is called when Media Job is running in non-distributed mode. A non-distributed version of the task can be added in this method
prepareTasks()	Method is responsible for preparing data for the tasks executed on client machines in distributed mode. The method is executed on the NameServer only
getTasks()	Client machines call this method on NameServer to get their data. This method uses the output of prepareTask() method and sends the client its share of the data
doTasks()	The method to process the data fetched from getTasks() method. This is called by MediaUpdater on each client after they complete getTasks()
putTasks()	Each client calls this method on the NameServer to store their processed results on the NameServer
finishTasks()	The method is called on the NameServer to complete the final computation after all tasks are completed

3.2 Machine Learning Static-Dynamic Cache

For this task, Machine Learning Static-Dynamic Cache [7] was implemented using Python and Sci-kit learn library. The dataset was created using AOL query logs [10]. Due to the limited availability of the data, a subset of features from the original work was used to train the algorithm for admission and eviction. Table 3 describes features extracted from the query log.

To create a feature set, queries were cleaned using case folding, stemming, and stopword elimination. Features were then derived from this set of clean queries. 'IAT_NEXT' is the ground truth value that represents the next time the same query appears i.e., the number of queries between the next appearance of the query. The training data were highly skewed, more than 80% of the data had an IAT_NEXT value of less than 1% of the total data (Figure 1). To mitigate this issue, IAT_NEXT values were binned using logarithmic binning (Figure 2).

A regression model was fitted on this dataset and it achieved the hit rate of 28.33% for a cache of size 100 and 4K queries while other benchmarking algorithms LRU and optimal offline algorithm achieved a hit rate of 52.6% and 57.27%. The hit rate of this algorithm was remarkably lower than other simpler algorithms and will require significant feature engineering and fine-tuning to be useful in a production system like Yioop.

Table 3: Features extracted from query logs

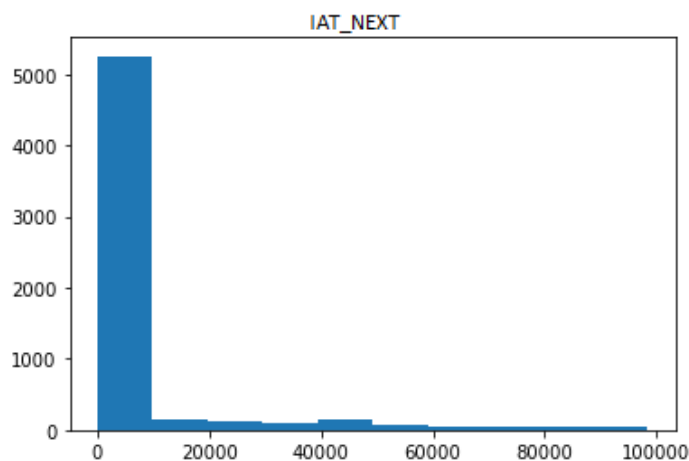| Feature	Description
QUERY_HOUR	Hour of the day, the query was fired
LAST_MIN_FREQ	Frequency of query in last minute
LAST_HOUR_FREQ	Frequency of query in last hour
LAST_DAY_FREQ	Frequency of query in last day
PAST_FREQ	Total frequency of the query
IS_TIME_COMPAT	Whether query is time compatible
QUERY_LENGTH	Number of words in the query

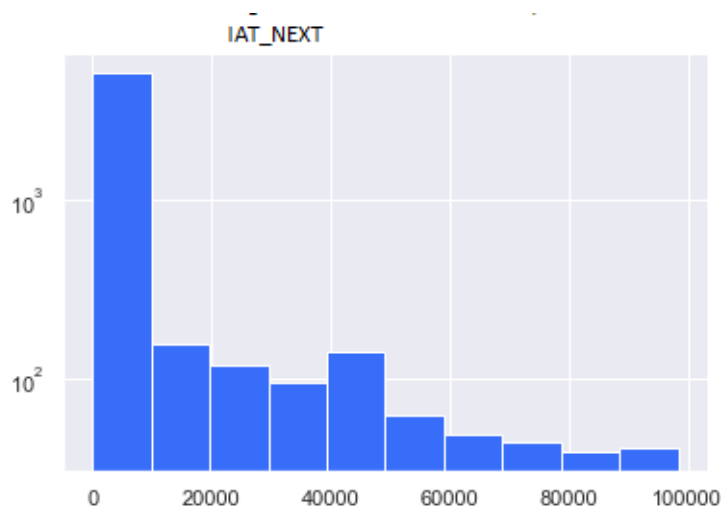Figure 1: Distribution of IAT_NEXT values

Figure 2: Distribution of IAT_NEXT values after log binning

3.3 Static-Semistatic-Dynamic Cache

This preliminary work aimed to implement Static-Semistatic-Dynamic Cache [8] which adds a semi-static segment to the SD cache. The findings from the author show that there are certain queries that are highly frequent in the daytime and other queries which are highly frequent during the night. Based on the time of the day, the daytime or nighttime semi-static segment of the cache is loaded into the memory. All time popular queries are stored in a static segment of the cache. A dynamic segment is governed using the LRU eviction policy.

Training of this algorithm was performed using 1.5M queries and results were evaluated on another set of 1.5M queries. The cache size was set to 300 for this experiment as this algorithm is effective when the cache is large. The performance of this algorithm was 1.2% lower than the SDC algorithm (Table 6). As this algorithm requires large cache space and there was no scope for improvement, the algorithm was discarded for use.

Table 4: Result of SSDC algorithm

Cache Size	Configuration	Hit Rate
300	SDC(80-20)	56.65%
300	SSDC(10-70-20)	55.76%
300	SSDC(20-60-20)	55.95%
300	SSDC(40-40-20)	56.21%
300	SSDC(60-20-20)	55.94%
300	SSDC(70-10-20)	56.56%
300	SSDC(0-80-20)	55.46%

3.4 Static-Topic-Dynamic Cache

Static-Topic-Dynamic Cache (STDC) [9] was implemented as part of this work. The topical cache was created using Latent Dirichlet Allocation (LDA) topic model. The original work proposed two methods two enrich search queries with contextual data - one using summaries of top results and the other using document clicked by the user. As search engine data was not available to train the LDA model, a corpus was generated using a news dataset. The LDA model was trained on the news dataset to identify topics in the news dataset.

The performance of this algorithm was evaluated on 10K and was a significant improvement over the LRU baseline algorithm. The algorithm improved the hit rate by 2% resulting in a gap reduction of 12% with the optimal offline algorithm (Table 5). Also, compared to SDC, the performance was not significantly lower. There was scope for improvement in performance by adding contextual information to the queries. In my final work, I extracted contextual information using the user's clicked url data.

Table 5: Result of STDC algorithm

Cache Size	Configuration	Hit Rate
100	LRU	42.33%
100	Belady	49.08%
100	SDC(30-70)	43.35%
100	STDC(30-50-20)	43.16%
100	STDC-V(30-50-20)	43.10%
200	LRU	44.49%
200	Belady	49.81%
200	SDC(15-85)	45.16%
200	SDTC(15-50-35)	45.08%
200	STDC-V(15-50-35)	45.01%

CHAPTER 4

Implementation of Static-Topic-Dynamic Cache

Static-Dynamic cache combines static and dynamic caches to adapt to both the long-term and short-term popularity of the query. The static part is populated using historically popular queries which adapts well to long-term popular queries. While the dynamic part dynamically decides on the admission and eviction of data which gives better results for short-term popular queries. This approach does not take into account different access patterns of queries belonging to different topics. For E.g. queries belonging to the topic 'Weather' is accessed more frequently during the day while queries related to 'Entertainment' may be accessed more during the evening. Having a single dynamic cache for queries results in the eviction of cached data which is supposed to be coming in the future.

To address this issue, Static-Topic-Dynamic Cache (STD Cache) adds a Topical cache along with static and dynamic cache in this architecture (Figure 3). Each topic is assigned its cache instance managed by its admission and eviction policy. Each of these caches can be an instance of an LRU cache or even an SD cache. This allows 'Weather' and 'Entertainment' to have their own cache space which results in more cache hits. The size of each topic can be adapted to its popularity resulting in higher cache space utilization.
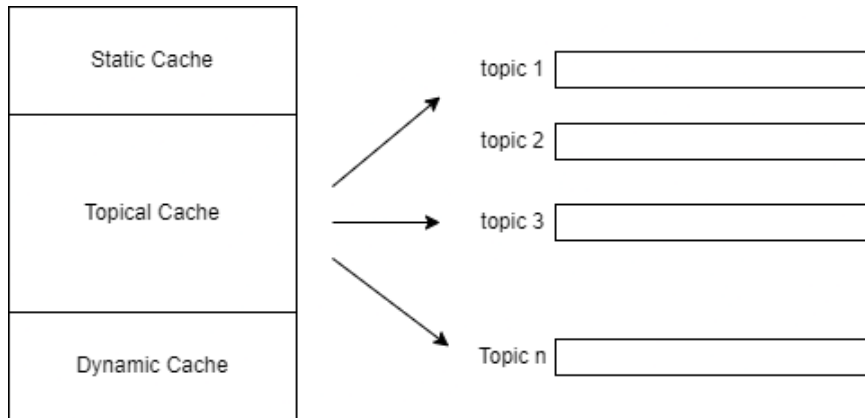
Figure 3: Static-Topic-Dynamic Cache

4.1 Caching Architecture and Strategies

The following subsection gives details about the incorporation of caching architecture and strategies in Yioop. It gives detail about all caching algorithms along with each segment of Static-Topic-Dynamic cache and how it is implemented in Yioop.

4.1.1 Incorporation of Caching Architecture in Yioop

The choice of caching algorithm depends highly upon the use case. As Yioop is a web search engine used for both general-purpose internet crawling and crawling a set of web pages on the user's website, cache requirements are different for different use cases. Therefore, it becomes of utmost importance for search engines like Yioop to allow users to choose caching strategies as per their needs. The current caching strategy of Yioop is monolithic and inflexible. It is implemented using Marker's algorithm and can not be changed by the user.

In this work, a new cache design is implemented to allow easy switching between caching strategies. The new design gives the user flexibility to choose between different caching strategies. Each caching algorithm is an instance of a Cache abstract class allowing code reuse and flexibility. StaticDynamicCache and StaticDynamicTopical-Cache are caches built using an implementation of StaticCache and LRUCache for

16

static and dynamic caching. These caches are instance of the Cache class hence can be easily swapped with another implementation of static and dynamic cache (Figure 4). Class diagram below (Figure 5, 6) shows the structure of new cache design.
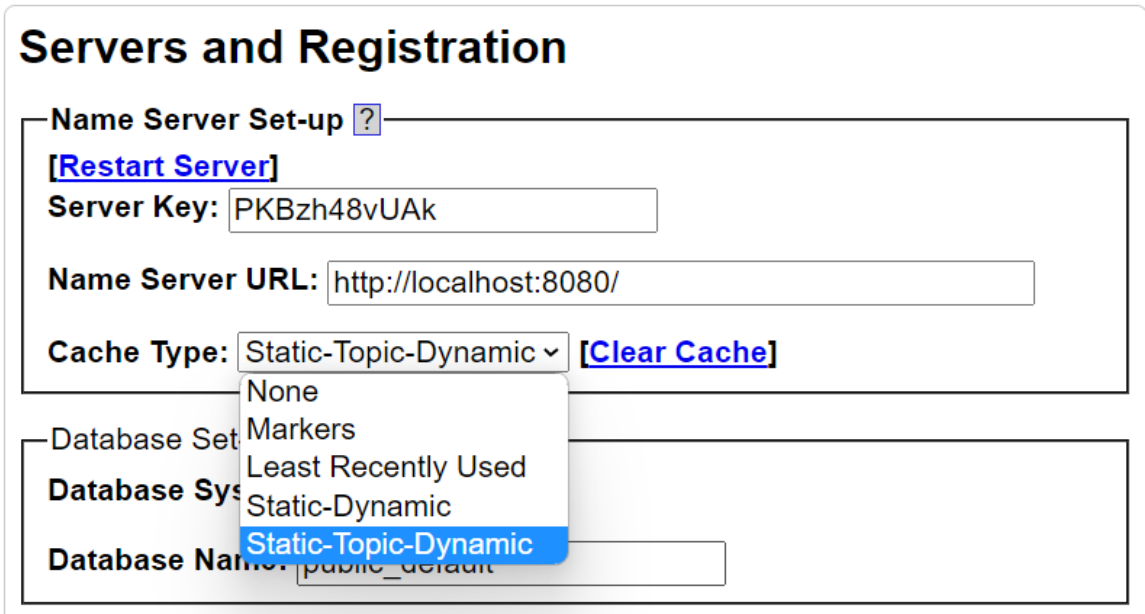

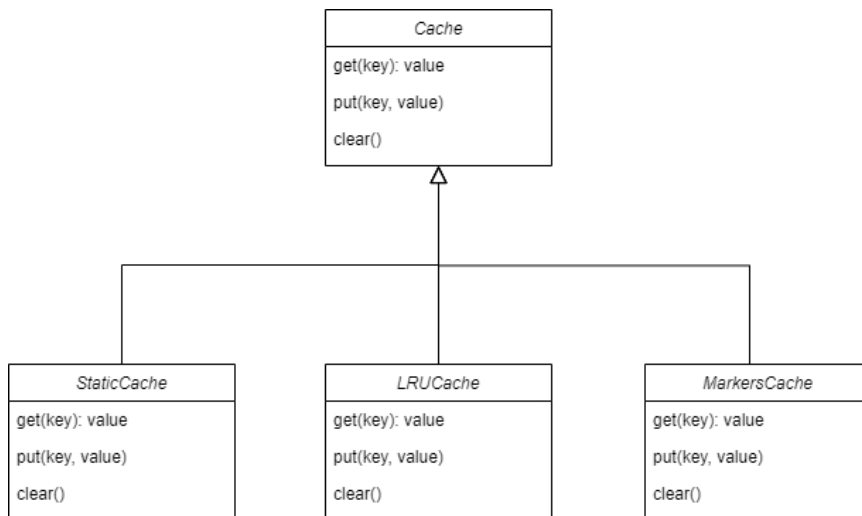Figure 4: Dropdown to select caching startegy

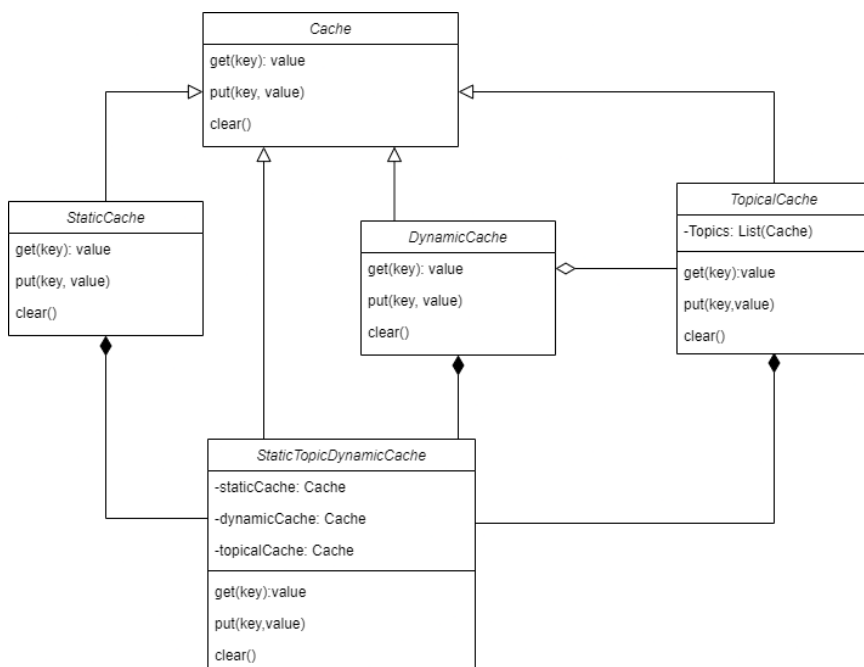Figure 5: Class Diagram of Single level caches

17

Figure 6: Class Diagram of StaticTopicDynamicCache

4.1.2 Static Cache

Static Cache is a simple read-only cache in which data is populated offline using historical query data. Top K queries are selected from query logs based on certain features from the query. In my implementation StaticCache class is a static cache and query frequency is used to populate the static cache. It stores top k frequent queries are selected from the query logs.

4.1.3 Dynamic Cache

Dynamic section of the cache deals with online data and is governed by admission and eviction policy. For this project, I have implemented LRUCache class which uses LRU algorithm for eviction. Along with these implementation MarkerCache is been separated out as an individual dynamic cache. Thus, Yioop now has two implementations of dynamic cache. Each of these cache can be used as standalone cache or can be used in conjunction with static cache described in next subsection.

Class	Description
Cache	Abstract class inherited by all cache implementations. It provides access to common methods of cache
CacheMetricWrapper	Wrapper class to collect performance metrices like hit rate, miss rate of caches
StaticCache	Read-only cache which stores long term popular queries
LRUCache	Dynamic cache which takes eviction decision based on LRU policy
MarkersCache	Implementation of Marker's algorithm which uses randomization for eviction decision
StaticDynamicCache	Implementation of Static-Dynamic cache. It uses instance of StaticCache and LRUCache for caching
TopicalCache	Uses K-Means topic model for topic extraction. Stores data in its repective topic cache
StaticTopicDynamicCache	Implementation of Static-Topic-Dynamic cache. It uses instance of StaticCache, TopicalCahe and LRUCache for caching
CountVectorizer	Converts text into sparse vector
KMeansClustering	Implementation of K-Means Clustering algorithm

Table 6: List of classes implemented in Yioop

4.1.4 Static Dynamic Cache

Static Dynamic cache is one the most widely used approach for caching and implemented in StaticDynamicCache class in Yioop. It combines static and dynamic cache to achieve results best form both worlds. StaticDynamicCache class is parameterised by static and dynamic cache, meaning implementation of SD cache can be configured. In my implementation, it configured to use StaticCache for static caching and LRUCache for dynamic caching.

Static-Dynamic Cache first checks whether the data is available in static part. If it is available it is returned to user. If it is not present, it checks dynamic cache for the data. The dynamic cache checks whether it contains the queried data. If it

is present, the data is returned to user else it returns miss to the user. Below is the
pseudocode for implementation in StaticDynamicCache.

<center>Pseudocode for Static-Dynamic Cache</center>

```
if query in StaticCache:
    return hit;
else:
    if query in DynamicCache:
        return hit;
    else
        return miss;
```

4.1.5   Topical Cache

TopicalCache class is implemented as a standalone cache in Yioop. TopicalCache
initializes K dynamic caches which essentially are the instance of LRUCache. Topic
model is used for the extraction of topic from query is deserialized from a file containing
trained K-Means clustering model. Number of topics K is derived from the deserialized
model.

TopicalCache first extracts topic from query using topic model which in my case
is a K-Means algorithm. If the topic cache of that query contains the data hit is
returned else it returns miss to user. Below pseudocode shows implementation of
TopicalCache.

<center>Pseudocode for Topical Cache</center>

```
topic = TopicModel::getQueryTopic(query)
if query in TopicalCache(topic):
    return hit;
else:
    return miss;
```

4.1.6   Static-Topic-Dynamic Cache

Static-Topic-Dynamic Cache is implemented in StaticTopicDynamicCache class
in Yioop. It is a combination of StaticDynamicCache and TopicalCache. Topical

<center>20</center>

cache is added to Static-Dynamic cache to adapt to different access pattern of topic in the queries. Like StaticDynamicCache, this cache is also configurable to use any implementation of static, dynamic and topical cache. In the current implementation it is configured to be using StaticCache, LRUCache and TopicalCache based on K-Means algorithm for its caching.

Similar to SD cache, STD cache first checks whether data is available in static cache. If it is available, it returns hit else it checks the topical cache. Topical cache does the same and forwards query to the dynamic cache. If query is present in the dynamic cache, hit is returned else miss is returned. The following pseudocode demonstrates how the Static-Topic-Dynamic cache is implemented.

Pseudocode for Static-Topic-Dynamic Cache

```
if query in StaticCache :
    return hit ;
else :
    topic = TopicModel :: getQueryTopic ( query )
    if query in TopicalCache ( topic ):
        return hit ;
    else :
        if query in DynamicCache :
            return hit ;
        else
            return miss ;
```

4.2   Query Topic Distillation

Queries given to the search engine are usually very short, containing 1 to 3 terms. Extracting query topic from such a small text is a difficult task. To extract the topic information, query data has to augmented with additional contextual information. This contextual information along with the query can be used to derive the intended topic of the query. One way to do this is to use summaries of top search results of the query to augment contextual information. The other way to do it is to use summary

of data from page clicked by the user after querying.

In Yioop, the other approach is taken to use user's clicked webpage to add contextual information to the query.The page clicked by the user gives more relevant data about the user's actual intention. To train the K-Means topic model, query was augmented with the summary from clicked webpage so that relevant topic could be discovered.

The url clicked by the user is logged in Yioop's impression tables. Yioop's impression tables are basically logs of relevant user's interaction with the Yioop. Due to the recency of these logs, sufficient number of logs are not available. Therefore for this work, we are relying on AOL query logs which contains urls clicked by the users.

4.3   Word Embeddings

Machine learning algorithms requires text to be represented as vectors. In natural language processing, there are are number of ways to convert text into vector. In Yioop, CountVectorizer is implemented which converts corpus of documents into a document-term matrix. CountVectorizer first creates vocabulary from the corpus of document which it does by iterating over all the documents, finding out all the terms in the documents and assigning unique index to each unique term in the corpus.

Once a vocabulary is created it can convert documents into vectors. To convert a document into a vector it iterates all the terms in document to store the count of each term at its unique index in the vector. CountVectorizer iterates over all the documents to convert it into the vector and finally returns the document-term matrix. To optimize the performance of the this vectorizer, sparse vectors and matrix representations are used. Also to avoid creating vocabulary every time, serialization and deserialization capability is added to persist it in a secondary memory.

Pseudocode for simple CountVectorizer

```
Vocab = []
for Document in Documents:
    for Term in Document:
        if Term not in Vocab:
            Append Term to Vocab
DocumentTerm = []
for Document in Documents:
    Vector = []
    for Term in Document:
        Vector[Index of Term in Vocab] += 1
    Append Vector to DocumentTerm
return DocumentVector
```

4.4 Topic Model

Topic modeling is technique of discovering topic from the text. K-Means algorithm is an unsupervised originally used for clustering but can also be used in topic modeling. In Yioop, KMeansClustering class is implemented to discover topics using K-Means algorithm. K clusters in K-Means represent K latent topics. To use KMeansClustering, it has to be first trained using document corpus. The algorithm discovers K clusters in the the document. To assign topic the document, it finds cluster nearest to the document and assigns corresponding topic to the document.

KMeansClustering class accepts document-term matrix for the training. K-Means algorithm starts with initializing K random cluster centroids. Each document in the corpus is iterated to assign it to a clusters closest to it. Distance is calculated using euclidean distance between document vector and centroid of the cluster. After each iteration, mean of all vectors assigned to respective clusters are taken to update their respective centroids. The algorithm converges when centroids stops updating.

To improve the performance of the algorithm, algorithm accepts sparse document-term matrix and computes distances using sparse vectors. To avoid training model every time, feature is provided for serialization and deserialization of the model.

Pseudocode for K-Means Algorithm

```
Centroids = Select K random points from Documents
for Document in Documents:
    Cluster = FindNearestCluster to the Document
    Add Document to the Nearest Cluster
CentroidsOld = Centroids
for Cluster in Clusters:
    Find mean of all Documents in the cluster
    Update Centroid with the mean
if diff(Centroids, CentroidsOld) < Threshold:
    return Convergence
else:
    repeat till maximum iterations
```

(a) Word cloud generated for topic Restaurant

(b) Word cloud generated for topic Holiday

(c) Word cloud generated for topic Technology

(d) Word cloud generated for topic Party

Figure 7: Word cloud of different topics classified by k-means

4.5 Dataset

For the experiments, dataset was generated using Yioop's indexer and AOL query logs. AOL dataset contains contains over 36 million queries of anonymized users along the clicked url data. These queries were generated over 3 months of period in year 2006. The dataset contains the user id, query, item rank, clicked url, and a timestamp (Figure 8). Yioop's summary of crawled pages was added to the dataset using the indexer. This dataset was used for both evaluation of cache as well as training of k-Means topic model.

	AnonID	Query	QueryTime	ItemRank	ClickURL
0	53	mapquest	2006-03-01 15:18:21	1.0	http://www.mapquest.com
1	66	cajun candle	2006-03-01 13:20:18	1.0	http://www.cajuncandles.com
2	66	candle jars	2006-03-01 13:22:29	1.0	http://www.sks-bottle.com
3	66	muic.com	2006-03-01 22:42:05	NaN	NaN
4	66	i need a company name	2006-03-02 12:32:59	NaN	NaN

Figure 8: AOL Query logs

AOL query log is a raw dataset is not pre-processed to be used directly. The user queries were first cleaned by removing stopwords, punctuations, extra spaces and null values. Stemming and case folding were also performed to clean the queries. As the dataset contains old, many urls were http urls, these urls were converted into https urls. After this pre-processing clean queries were available for the evaluation of cache.

To create document corpus from the query logs, queries not containing clicked urls were eliminated. All clicked urls were converted into Yioop Crawler's seed data. Yioop's fetchers fetched all the clicked documents in Yioop and created inverted index on it. Yioop's indexer also generated summary for each of these crawled pages. Using Yioop's indexer, query document pairs were generated.

4.6 Evaluation of Cache

To evaluate the performance of cache in Yioop, CacheMetricWrapper class is created. It is wrapper class (Figure: 9) which calculates the performance metrices for the cache. For each query it checks whether underlying cache contains the value, if value is present, it adds to the hits otherwise adds to misses. Currently, CacheMetricWrapper keeps track of track of hits, misses, hit rate, and total number of queries.
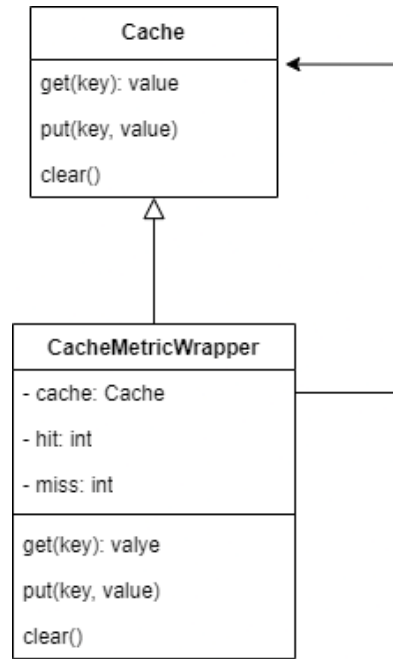

Figure 9: CacheMetricWrapper class diagram

The performance of caches was evaluated using different size of query logs, cache sizes, and cache segments. Testing was performed using separate test data created from AOL query logs. Queries were previously unseen by both topic model and caches. The performance was evaluated with 5K, 10K, and 20K queries, cache sizes between 200 to 500 entries, and 10 number of topics. For the evaluation purpose hit rate of cache is chosen. Caches having high rates require less number of result computations

which effectively reduces the response time of the search engine.

Cache Size	LRU	SDC (20-80)	SDC (30-70)	SDC (40-60)
200	69.8	70.6	71.04	71.4
300	69.96	71.12	71.72	72.34
400	69.98	71.58	72.4	73.2
500	70	72.02	73.02	74.04
Cache Size	STDC (20-40-40)	STDC (20-50-30)	STDC (20-60-20)	STDC (20-70-10)
200	70.22	69.96	69.68	69.56
300	70.98	70.74	70.54	70.36
400	71.46	71.38	71.12	70.9
500	71.92	71.88	71.72	71.56

Table 7: Hit Rates of Caches with Query Size 5K

Cache Size	LRU	SDC (20-80)	SDC (30-70)	SDC (40-60)
200	61.38	61.96	62.21	62.31
300	61.55	62.3	62.6	62.91
400	61.6	62.54	62.98	63.41
500	61.65	62.79	63.36	63.87
Cache Size	STDC (20-40-40)	STDC (20-50-30)	STDC (20-60-20)	STDC (20-70-10)
200	61.54	61.26	60.95	60.73
300	62.1	61.89	61.53	61.33
400	62.42	62.29	61.99	61.69
500	62.7	62.66	62.44	62.2

Table 8: Hit Rates of Caches with Query Size 10K

Cache Size	LRU	SDC (20-80)	SDC (30-70)	SDC (40-60)
200	60.19	60.62	60.66	60.75
300	60.37	60.87	61.11	61.25
400	60.52	61.14	61.38	61.6
500	60.97	62.15	62.75	63.36
Cache Size	STDC (20-40-40)	STDC (20-50-30)	STDC (20-60-20)	STDC (20-70-10)
200	60.11	59.8	59.41	59.04
300	60.54	60.28	59.92	59.63
400	60.91	60.73	60.42	60.13
500	61.95	61.86	61.76	61.66

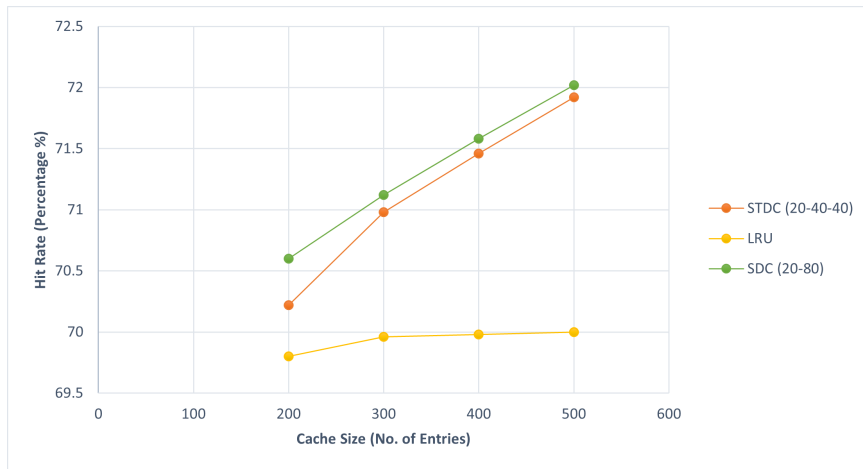Table 9: Hit Rates of Caches with Query Size 20K

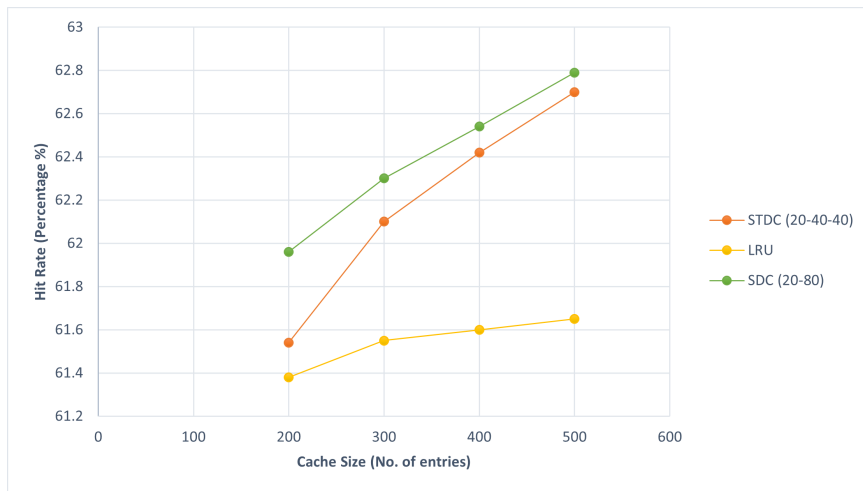Figure 10: Hit rates of LRU, SDC and STDC with 5K queries

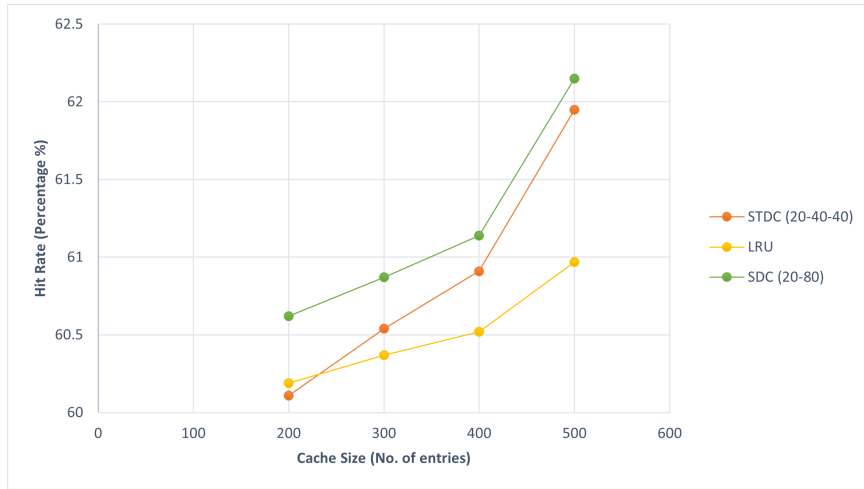Figure 11: Hit rates of LRU, SDC and STDC with 10K queries

Figure 12: Hit rates of LRU, SDC and STDC with 20K queries

In the conducted experiments, Static-Topic-Dynamic cache performs consistently better than purely dynamic LRU cache for all cache sizes and number of queries. It is also observed that performance of SD cache is slightly better than STD cache but the difference gets reduced with large cache sizes. Depending on the use case SD cache or STD cache can be chosen.

CHAPTER 5

Conclusion

Thus I have implemented Static-Dynamic cache along with its variation of Static-Topic-Dynamic cache in Yioop. Static-Topic-Dynamic cache exploits temporal locality of topic in query streams for improving the performance. It uses topic model to extract topic from the query. In this project, topic model has been created using K-Means algorithm and was trained using Yioop's own crawled pages. With this work, performance of cache in Yioop has been significantly improved. The newly implemented Static-Dynamic cache and Static-Topic-Dynamic cache performs consistently better than previously implemented caching algorithm based on Marker's algorithm. As the performance of cache also depends on different use cases, a flexible and extendable architecture for cache is created. Users are given easy interface to switch between cache strategies for specific use cases.

LIST OF REFERENCES

[1] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 183–190. [Online]. Available: https://doi.org/10.1145/1277741.1277775

[2] T. Fagni, R. Perego, F. Silvestri, and S. Orlando, "Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data." *ACM Trans. Inf. Syst.*, vol. 24, pp. 51--78, 01 2006.

[3] X. Jin and J. Han, *K-Means Clustering.* Boston, MA: Springer US, 2010, pp. 563--564. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_425

[4] E. Markatos, "On caching search engine query results," *Computer Communications*, vol. 24, no. 2, pp. 137--143, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014036640000308X

[5] R. Ozcan, I. S. Altingovde, and O. Ulusoy, "Static query result caching revisited," ser. WWW '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1169–1170. [Online]. Available: https://doi.org/10.1145/1367497.1367710

[6] R. Ozcan, I. S. Altingovde, and O. Ulusoy, "Cost-aware strategies for query result caching in web search engines," *ACM Trans. Web*, vol. 5, no. 2, may 2011. [Online]. Available: https://doi.org/10.1145/1961659.1961663

[7] T. Kucukyilmaz, B. B. Cambazoglu, C. Aykanat, and R. Baeza-Yates, "A machine learning approach for result caching in web search engines," *Inf. Process. Manage.*, vol. 53, no. 4, p. 834–850, jul 2017. [Online]. Available: https://doi.org/10.1016/j.ipm.2017.02.006

[8] T. Kucukyilmaz, "Exploiting temporal changes in query submission behavior for improving the search engine result cache performance," *Information Processing Management*, vol. 58, no. 3, p. 102533, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306457321000418

[9] I. Mele, N. Tonellotto, O. Frieder, and R. Perego, "Topical result caching in web search engines," *Information Processing Management*, vol. 57, no. 3, p. 102193, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306457319307253

[10] G. Pass, A. Chowdhury, and C. Torgeson, "A picture of search." ACM, 6 2006.