ADVANCED METHODS FOR RESULT AND SCORE CACHING IN WEB SEARCH ENGINES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ERMAN YAFAY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEP 2019

Approval of the thesis:

ADVANCED METHODS FOR RESULT AND SCORE CACHING IN WEB SEARCH ENGINES

submitted by **ERMAN YAFAY** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. İsmail Sengör Altıngövde
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Pınar Karagöz
Computer Engineering, METU _____

Assoc. Prof. Dr. İsmail Sengör Altıngövde
Computer Engineering, METU _____

Assist. Prof. Dr. Tayfun Küçükyılmaz
Computer Engineering, TED University _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Erman Yafay

Signature :

ABSTRACT

ADVANCED METHODS FOR RESULT AND SCORE CACHING IN WEB SEARCH ENGINES

Yafay, Erman
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. İsmail Sengör Altıngövde

Sep 2019, 51 pages

Search engines employ caching techniques in main memory to improve system efficiency and scalability. In this thesis, we focus on improving the cache performance for web search engines and present two main contributions in this direction. Firstly, we investigate the impact of the sample size for frequency statistics for most popular cache eviction strategies in the literature, and show that cache performance improves with larger samples, *i.e.*, by storing the frequencies of all (or, most of) the queries seen by the search engine. By adopting a previous approach from the literature, we mitigate the cost of storing a large history of frequencies by using a Counting Bloom Filter based data structure that is able to store frequency statistics in a compact manner, while still providing comparable cache performance to keeping all frequencies in a raw manner. Secondly, we propose a new cache type for systems that employ dynamic pruning strategies (*e.g.* WAND or BMW) for query processing. We store the k-th highest result score for a query at the index nodes as a static cache. Whenever a result cache miss occurs at the broker, we use k-th score of the subsets of the original query as an initial threshold value for dynamic pruning. Our method reduces

the query processing time by increasing the number of documents skipped and, to our knowledge, it is unique in the sense that it can improve processing times for compulsory result cache misses and singleton queries.

Keywords: Search Engine, Cache, Efficiency

ÖZ

WEB ARAMA MOTORU SONUÇ VE SKOR ÖNBELLEKLERİ İÇİN İLERİ YÖNTEMLER

Yafay, Erman
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. İsmail Sengör Altıngövde

2019 , 51 sayfa

Arama motorları, önbellekleme yöntemlerini sistem verimliliği ve ölçeklendirilebilirliği artırmak üzere ana bellek üzerinde sıklıkla kullanır. Bu çalışmanın kapsamı, web arama motoru önbellek performansını iyileştirmeye yönelik olup, katkıları iki ana kısımda incelenebilir. Öncelikle, sorgu geçmişi frekansı örneklerinin hepsinin (ya da bir çoğunun) literatürde önde gelen önbellek yöntemlerinde tahliye için bir sinyal olarak kullanıldığında önbellek performansının arttığını göstermekteyiz. Bellekte kullanılan uzun sorgu geçmişinin kaplayacağı büyük bellek alanının, daha önceki çalısmalarda da kullanılan, Counting Bloom Filter tabanlı veri yapıları kullanarak kompakt bir şekilde saklanabileceğini ve frekans değerlerini olduğu şekilde saklayan yöntemlerle eşdeğer önbellek performansı elde edilebileceğini göstermekteyiz. İkincil olarak, dinamik budama yöntemleri (WAND, BMW gibi) kullanan arama sistemleri için yeni bir önbellek tipi öneriyoruz. Bu önbellekte bir sorgunun k. yüksek sonuç skorunu, sonuç önbelleğine ilaveten saklamaktayız. Herhangi bir sonuç önbellek kaybı oluştuğunda, sorgunun alt kümelerinin k. sıradaki skorunu sorgu işleme sürecinde başlangıç eşik değeri olarak kullanıyoruz. Bu yöntem ile sorgu işleme sürelerini daha fazla do-

küman budayarak azaltıyoruz, ve bilgimiz dahilinde, yöntemimiz zorunlu sonuç önbellek kayıpları ve bir kez gözlemlenen sorguların işleme zamanını azaltabilen eşsiz bir önbellek tipi.

Anahtar Kelimeler: Arama Motoru, Önbellek, Verimlilik

To my family and friends

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor Assoc. Prof. Dr. Ismail Sengor Altingovde for his guidance and genuine efforts to encourage me on my thesis research. Even before supervising me, he shared his research ideas in a classroom setup and valued the outcome of what I have accomplished.

I acknowledge the support I received from The Scientific and Technological Research Council of Turkey (TUBITAK) under the grant 117E861.

Lastly, I would like to thank some of the senior colleagues from ASELSAN, Ali Sezgin, Alper Tolga Kocatas and Sener Yilmaz who did not mind me being off work when I had deadlines to meet during my thesis studies. Without their understanding things would've been much harder.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

BF	Bloom Filter
SE	Search Engine
CBF	Counting Bloom Filter
LRU	Least Recently Used
LFU	Least Frequently Used
HTML	Hyper Text Markup Language
WAND	Weak-AND
BMW	Block Max WAND
URL	Uniform Resource Locator
BM25	Best Matching 25
TaaT	Term at a time
DaaT	Document at a time
TF	Term Frequency
IDF	Inverse Document Frequency
SERP	Search Engine Results Page

CHAPTER 1

INTRODUCTION

1.1 Motivation

Large scale search engines accommodate hundreds of data centers all over the globe that are operational 7/24. Indeed, the maintenance costs for such systems are in the scale of millions of dollars. Therefore, even minor improvements over computation or storage requirements can provide substantial financial and environmental benefits. On the other hand, considering the current retrieval time of web search engines, which are in the order of couple of hundred milliseconds, faster retrieval is an important aspect to attract new users in a highly competitive market.

A typical web search engine processes queries and generates result sets in a decreasing fashion of interest to the user *i.e.*, results are ranked by the similarity of the documents to the query. To cope with the large web documents which are in the order of billions, search engines work in a distributed manner. This way, not only the system throughput can be increased by parallel execution, but also the index of the collection can fit into the disks of multiple machines for faster access. Such horizontal scaling has shown to be promising for web search engines, since in theory, it can scale infinitely by just adding more commodity computers to the clusters, whereas a single but more advanced machine usually does not offer a reasonable price-performance ratio or maybe does not even exist.

The Web pages, or more generally, the documents that comprise the collection are first processed to create an index, for faster retrieval (see Fig. 3.1), then the index is partitioned *i.e.*, sharded across several index nodes. On top of the index nodes, the brokers serve to redirect queries (possibly also pre-processing them; such as query ex-

pansion) to index nodes where each node processes the query with its own portion of the index and computes top-k partial results that are ranked via TF-IDF like retrieval functions. Next, these results are sent back to the brokers where they are merged and ranked again by more sophisticated machine learned models to obtain the final result set. Obviously, the initial retrieval and ranking stage finds relevant documents among millions and thus is more efficient compared to ranking stages at the brokers. But still, a huge portion of the query processing time is spent at the index nodes where the initial matching occurs.

Caching is one of the methods that is extensively employed in web search engines to reduce retrieval times. Search engines use caching techniques in different layers of the system *e.g.* brokers, index nodes etc. and with various content such as HTML result snippets and posting lists [1]. Typically, the former one resides at the broker nodes since they are ready-to-serve content *i.e.*, no additional processing is required whereas the latter one, is used to reduce processing times by forming an intermediate result. Indeed, not all of the queries can benefit from a cache since only 56% of queries repeat [10]. The rest are singletons *i.e.*, occur only a single time and practically cannot be cached. Therefore, search systems developed methods to maximize their benefit from caches, either or together by proposing methods of different cache types or cache replacement policies [2, 3, 4].

1.2 Contributions and Thesis Outline

In this thesis, we focus on caching in search engines and propose methods to improve caching techniques both in terms of space and time. First, in Chap. 2, we focus on result caches and show that keeping the entire frequency history would yield significant improvements over the hit rate; unlike the previous work where it has been only experimented with either none or a short frequency history. We then show and adapt a previously proposed compact storage scheme, namely the Tiny [5, 6] to mitigate the cost of storing such large cache metadata by modeling the storage requirements for various well-known cache eviction policies. To make our contribution more concrete, we extend the saved space by both a result and a doc-id cache; and show that our savings are meaningful for both of them. Yet we show that the additional doc-id cache

can further exploit the gained space compared to its result counterpart in a cost-based evaluation setup.

Secondly, Chap 3 presents some preliminary concepts in information retrieval such as the inverted index, ranking and query processing. We provide rigorous algorithms for query processing both in Chap 3 and Appendix A, as well as explain dynamic pruning strategies. Next, we introduce our idea of the score cache to be placed at the index nodes. Score cache is cheap, as it only requires a space of 1% compared to a typical result cache. It stores the k-th highest score for a query and the score can be exploited to speed up query processing for systems that use dynamic pruning strategies. We acknowledge that real-life systems already employ a result cache that answers queries with virtually no cost. Thus, we use score cache only when the query is not resident in the result cache and use the score to obtain a lower-bound for the initial threshold of dynamic pruning strategies using subsets of the to-be-processed query. Since, generating all of the subsets would be expensive, we propose heuristics that offer high probability of being a cache hit and also yield tighter lower-bounds for score threshold. This way, we further increase the number of documents skipped and reduce processing times. Our idea is unique as a caching application such that it offers improvements for compulsory cache misses (queries occurring for the first time) and singleton queries in which a typical result cache offers no benefits.

Finally, we conclude this thesis in Chap 4 while discussing future research directions.

The work described in Chap 2 and Chap 3 were published in:

- E. Yafay and I. S. Altingovde. "On the Impact of Storing Query Frequency History for Search Engine Result Caching." *European Conference on Information Retrieval.*, ECIR 2019, (pp. 155-162). Springer, Cham., 2019.
- E. Yafay and I. S. Altingovde. "Caching Scores for Faster Query Processing with Dynamic Pruning in Search Engines." *Conference on Information and Knowledge Management*, CIKM 2019, ACM, 2019.

respectively.

CHAPTER 2

IMPACT OF QUERY FREQUENCY HISTORY

2.1 Introduction

Large-scale search engines extensively employ caching of query results in main memory to improve system efficiency and scalability [7, 8, 9, 10]. As in many other domains, frequency of past data items (or, requests) is a strong signal to decide on the items to be evicted from the query result caches of search engines, and used on its own in the well-known eviction policy Least Frequently Used (LFU) or combined with other signals, like item recency and size, in other policies [11].

In this chapter, our contributions are three-fold: First, we investigate the impact of storing a large query frequency history versus just keeping the frequency of the queries that are in the cache. We show that keeping the entire history (i.e., frequency of all seen queries by the search engine) may improve the cache performance (i.e., hit ratios) for the policies that employ frequency as a signal for eviction. While similar findings have been shown for other caching applications (e.g., see [5]), as far as we know, this issue has not been explored in depth for result caching in search engines.

Secondly, we adopt a recently proposed storage scheme for exactly this purpose, i.e., storing past request frequencies in a compact manner for caching, to our application domain. The latter scheme, referred to as Tiny here, can store the query frequency history by using Simple and Counting Bloom Filters (CBFs) [5]. Our experiments reveal that the storage space for query history can be significantly reduced while cache performance still remains comparable to storing the entire query history. This is an important finding as the number of queries submitted to search engines has reached to very large numbers and storing an (almost) full history may have very demanding

memory storage requirements.

As our third contribution, we investigate the performance gains when the saved memory space (using the Tiny approach) is further exploited to cache additional query results. To this end, we consider not only storing query results in HTML format (i.e., top-k results' URLs and snippets), but also adopt techniques that allow storing just the document identifiers for certain query results [1, 3]. For this latter case, we evaluate the cache performance in terms of the query processing cost, and show that using the saved space to store identifiers yields significant gains. To our best knowledge, no earlier work conducted such an exhaustive analysis for search result caching where compact schemes are employed for storing query frequency history, and our findings here shed light on the potential gains in terms of storage space, hit-rate and even query processing cost, all of which would be worthwhile in practical search systems.

Through Sec. 2.2 to Sec. 2.4, we layout the foundational information about BFs, sketches and finally review the Tiny storage scheme as proposed in [5]. Sec. 2.5 presents our exhaustive experimental evaluation and finally Sec. 3.5 concludes this chapter.

2.2 Bloom Filters

Bloom Filters (BFs) [12] are probabilistic data structures that allow querying the existence of an item a_i in a set $A = \{a_1, a_2, \ldots, a_n\}$ of length n. Each element a_i can be inserted to the BF by applying k distinct hash functions and computing bit positions $h_1(a_i), h_2(a_i), \ldots h_k(a_i)$ each within range $[0, m)$ where the bits at these positions are set to 1. Similarly, to query the existence of an element a_i the bits at the same positions are read and if any of the them are equal to 0 then it is certain that $a_i \notin A$. Otherwise, we conclude that $a_i \in A$ with a probability of false positive since some or all of the bit positions can be set due to hash collisions. Figure 2.1 visualizes the insertion operation to the BF. Unlike the *set* data structure (usually implemented with hash tables or balanced trees) which requires, at least, keeping each element in memory, a BF merely needs a bit vector to be allocated. Hence, for systems that are scarce in memory and are tolerant to false positives; Bloom Filters are compact alternatives.

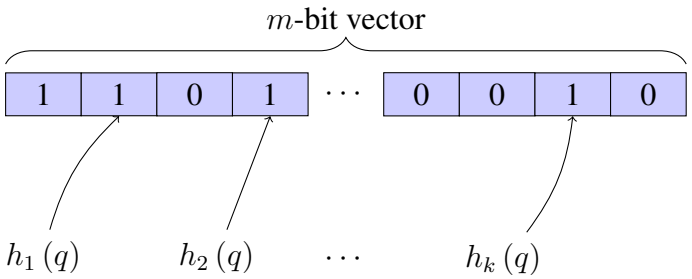


Figure 2.1: Setting bits at calculated hash positions for an element q.

In order to fully benefit from BFs, one should keep the probability of false positives (P_{fp}) to a minimum. Using a BF of length m and k distinct hash functions, the probability of a certain bit to remain 0 after the insertion of n elements is $P_0 = \left(1 - \frac{1}{m}\right)^{kn}$ and thus the probability of all hash functions to collide *i.e.*, a false positive to occur is given in Equation 21.

$$P_{fp} = (1 - P_0)^k \tag{21}$$

Therefore, to minimize P_{fp} the optimal number for k becomes $\ln 2 \times \frac{m}{n}$ which shows that for a large m it is feasible to use more hash functions. On the other hand, a too large number for k would significantly increase the number of bits set *i.e.*, the density of the filter and thus reduce P_0 which would yield an increase in P_{fp}. Additionally, Bloom Filters are analyzed more in detail [13, 14] and recently has been proposed to used as an alternative for inverted indices [15].

2.3 Approximation Sketches

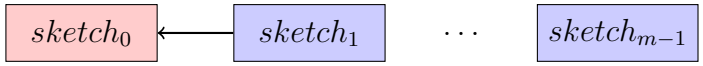
Approximation Sketches (*sketches* for short) use similar techniques to BFs such as hashing, but their purpose is to approximate count of occurences (or frequencies) of keys a_i from a data stream $S = (a_1, a_2, \ldots, a_n)$. Again, *sketches* are probabilistic data structures that offer compact representation of frequency values. Systems that are dealing with large streams but have limited resources can benefit from *sketches* as we show in Section 2.5, how we adapted a special purpose *sketch* called Tiny in the context of SE caching.

2.3.1 Count-min Sketch

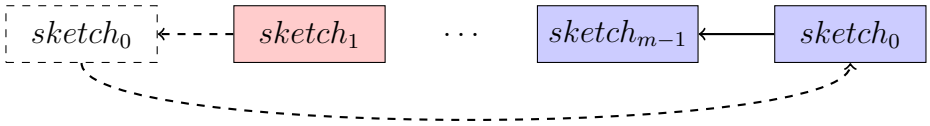
The most classical example of a *sketch* is the *count-min sketch* [16] that only supports incrementing the frequency value f_i corresponding to the key a_i. Unlike the simple BF, rather than allocating a single vector, *count-min sketches* allocate a matrix M of size (k, w) where each cell contains a fixed width counter. Each row of the matrix M is reserved for a hash function h_j where $j \in [1, k]$ and $h_j \in [0, w)$. Consequently, when a key a_t is observed at time t in S, we compute all h_j and increment the frequency values at M_{j-1,h_j}. Similarly, to estimate the frequency value, the minimum of the cells i.e. $\min_j M_{j-1,h_j}$ is used as the answer, since values that are larger than the minimum are hash collisions and thus guaranteed to be overestimations. By choosing the minimum value, *count-min sketch* minimizes the estimation error. Note that it is not possible for a *count-min sketch* to count lower frequencies for keys than their actual value.

2.4 Mitigating Saturation of Sketches

In most large scale applications such as SE's, the length of the input data stream is unknown and possibly quite large. Over time, most of the counters in the sketch are incremented (increasing the bit density) and the number of hash collisions increase, resulting in an high error rate. In such scenarios we say that the sketch is *saturated*. As a solution, it could be feasible to just zeroize all of the counters in the sketch after some fixed number of increment operations. However, it is likely that, the information related to recent or active keys are still important and can have adverse effects on the system performance if they are completely omitted. To overcome this issue, we briefly explain techniques and data structures that avoid or entirely negate saturation by selectively resetting or decrementing counters. Main idea of these methods focus on exploiting the phenomenon of temporal locality (keys that are occurring for short period of times) and/or the heavy tailed distributions of web data streams where most of the keys appear only once (singletons) or just a few times (Zipf distribution [17]). In their general structure, these methods divide the input stream into parts called the window W, and accumulate keys which are in that range. Afterwards, bit density of



(a) Initial sketch ordering



(b) After l elements, $sketch_0$ is inserted into the back while $sketch_1$'s counter are all reset.

Figure 2.2: m identical sketches are in a queue where the head sketch is drawn in red to indicate that l elements are being inserted to it only.

the sketch is reduced by resetting and / or reducing counters.

2.4.1 Bit Marking

Bit marking allocates a bit vector B such that its elements correspond to the each counter in the sketch. Initially all bits in B are set to 0. While incrementing counters, corresponding bits in B are also set. At the end of the current window (after processing W elements from the stream), counters which their bits remain unset are set to 0 so that keys that are inactive for the period can be removed. Obviously, bits in B are also set to 0 at the end of the window to detect the inactive keys of the next window. This way, inactive keys and singletons which make up a big portion of the input stream are removed.

Dimitropoulos *et al.* [18] show that although this method reduces the saturation it does not halt it. Inactive and singleton keys would eventually collide into the bucket of active keys and remain in the sketch, saturating it over time.

2.4.2 Sliding Window

In this approach m identical sketches are being allocated in a queue. The input data stream is divided into segments of length l where $W = l \times m$. While processing each segment, keys are inserted only into the head of the queue. At the end of a segment,

the head is popped and inserted to the back while the new head sketch's counters are all set to 0. This way, window of length W is slid over the stream where at every l'th processed element the least recently active keys' counters are decremented. Figure 2.2 summarizes the process of using m sketches in turn to avoid saturation.

Estimation of frequencies are done by summing up the frequency values from all m sketches. Since all of the sketches are identical, only a single computation of hash positions are sufficient for the estimation whereas, it still requires $m \times k$ memory accesses and m additions where k is the number of hash functions. Therefore, a large number for m simulates a better sliding window while increasing the time it takes to make estimates. Parameter configuration and further analysis are given by the original authors in more detail [18].

2.4.3 Tiny Storage Scheme

Einziger *et al*. introduced TinyLFU [5], a caching admission policy that proposed the use of a sketch data structure, referred as *Tiny*, that is specially designed for caching applications that use LFU eviction policy. At its core, Tiny consists of a simple Counting Bloom Filter (CBF) to keep track of the frequencies. Similar to *count-min sketch* (2.3.1), a CBF allocates fixed width counters to summarize the input stream in which the counters are flattened *i.e*., allocated on a vector, compared to that of a matrix. Although the estimation of frequencies are the same *i.e*., the minimum counter's value is returned; CBF's use Minimal Increment to accumulate its counters where only the minimum of the counters are incremented to further delay the saturation. For instance, if counters at hash positions are $\{2, 2, 3\}$, Minimal Increment increases only the minimum counters *i.e*., 2's. Therefore, new counters would be $\{3, 3, 3\}$ respectively.

Additional to the CBF, Tiny allocates a simple BF called the *doorkeeper*. The purpose of the additional storage is to capture the singleton keys in the *doorkeeper*, before submitting them to the CBF. In other words, counters of a corresponding key are only incremented in the CBF if they already exist in the *doorkeeper*. Therefore, one method Tiny uses to mitigate saturation is by capturing singleton keys; which for our purposes, make up to 44% [10] of web search engine query logs. Figure 2.3 illustrates

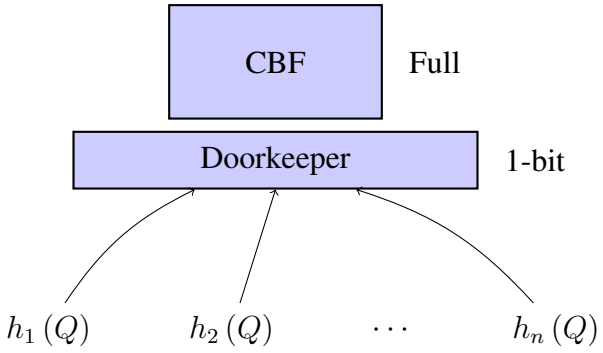


Figure 2.3: An illustration of the Tiny storage scheme.

the CBF and the *doorkeeper*.

Another method is called the *reset*; where Tiny accommodates a counter called the *window counter* that keep tracks of the number of keys processed. When it reaches W, counters in the CBF as well as the *window counter* itself are divided by 2 and bits of the *doorkeeper* are all set to 0. This way, similar to the *Sliding Window* (2.4.2), counters are reduced to both overcome saturation and adapt to the changes imposed by temporal locality.

2.4.3.1 Space Optimizations

Doorkeeper does not only serve to reduce approximation error, but also reduce space requirements by allowing Tiny to allocate fewer full counters in the CBF which are larger in size per bucket. Another space optimization is specific to caching applications and it is based on the observation that for a cache size of C and a window size of W, a key $a_i \in A$ would be reserved a slot in the cache if it satisfies the predicate $f(a_i) \geq \frac{W}{C}$ where $f(.)$ is a function in $A \rightarrow \mathbb{N}$ that approximates the frequencies from set A. For instance, intuitively, assume we have $W = 6$ and $C = 2$; if a key occurs 3 times, it already guarantees to be in the cache. No other key from the remaining stream can obtain a high enough frequency to discard it (it can be only equal to it which guarantees the other slot). Therefore, Tiny only requires to count up to $\frac{W}{C}$, and noticing that *doorkeeper* is able to count up to 1, full counters in the CBF can be capped to $\frac{W}{C} - 1$. As a result, unlike the naive approach where each counter is capped to W, Tiny require $\log_2\left(\frac{W}{C} - 1\right)$ bits. We give the required number of bits to allocate

the Tiny in Equation 22 where n is the number of full counters and d is the width of the *doorkeeper*. Note that typically $d > n$ and Equation 22 omits the required space by the *window counter* since its cost would be negligible in large scale web search engines where d and n are relatively larger and thus are the dominating factors.

$$d + n \times \log_2 \left(\frac{W}{C} - 1 \right) \tag{22}$$

Due to its small storage requirements and optimizations that fit our needs for SE result caching, we configure and adapt Tiny to our needs and describe it in the following sections. As a side note, we experimented with TinyLFU's ([5]) admission policy where a key is only admitted to the cache if its estimated frequency is larger than the victim's, and observed no differences to the eviction only scenario; where the victim is always evicted in terms of hit-ratio. Therefore, we only use the Tiny storage scheme as an approximation sketch for different eviction policies.

2.5 Experimental evaluation

Our experiments firstly focus on result caches that use single signal eviction policies. We measure the cache performance by hit-ratio (or hit-rate) which measures the ratio of queries (or items in general) served from the cache *i.e.*, the *hits* to the total number of queries including *miss*es which are queries that are missing from the cache and has to be processed.

We aim to find answers to following questions:

- What are the gains of storing the entire frequency history (both in raw (Pure LFU) and compact manner (Tiny LFU)) in hit-ratio?
- Does the adapted Tiny storage model reduce storage costs without surpassing the benefits of keeping the entire frequency history?
- Is the saved space is meaningful *i.e.*, can it be exploited to further increase cache performance by either extending the result cache or employing an additional doc-id cache?

Table 2.1: Parameters

Parameter	Value	Parameter	Value (bytes)
Number of distinct queries (U)	≈ 6.7 M	Size of a pointer (P)	4
Ranking per posting (P_r)	200 ns	Size of an int I (long int L)	4 (8)
Decompression per posting (P_d)	100 ns	Avg. size of a query ($AvgQ$)	16.5
Snippet computation per byte (P_s)	100 ns	Avg. size of a doc (d_{avg})	16384

2.5.1 Query log and simulation

AOL query log [19] is a chronologically ordered query log that is submitted by real users to the AOL search engine. The entire log contains $\approx$ 17 million queries. We use first 10 million as a training set and the remaining part ($\approx$ 7 million) is used for testing where the first 10% of it warm-ups the cache.

Note that, the training set is only useful to accumulate past query frequencies for the caching policies that employ the frequency as a signal (i.e., LFU [11] and GDSF [20]); however, it is not directly exploited to fill the cache, as there is a separate warm-up set. Since the LRU eviction policy does not require any training *i.e.*, the frequencies are not used during the eviction stage, the training set is not used at all.

2.5.2 Results for Single-Signal Caching Policies

We begin with describing two well-known cache eviction policies, namely, LRU and LFU. Additionally, our implementation details and memory models for aforementioned policies are presented briefly. These policies are *single-signal*, in the sense that they either use query recency or frequency as an indicator for eviction, respectively. Single-signal policies are rather simpler to implement compared to their *multi* counterparts.

2.5.2.1 Least Recently Used (LRU)

LRU eviction policy always chooses the least recently accessed item as the victim and evicts it from the cache whenever the cache is full. We used a doubly-linked list

Table 2.2: Memory space cost of LRU, LFU, and GDSF variants.

Policy	Memory Cost	Formula (bytes)
Pure (Metadata)	M^{pure}	$U \times (AvgQ + I)$
Tiny (Metadata)	$M^{tiny}(c)$	$\left(6W + 3W \times log_2\left(\frac{W}{c} - 1\right)\right)/8$
LRU	$M_{lru}(c)$	$c \times (AvgQ + 3P)$
In-memory LFU	$M_{lfu}^{inmem}(c)$	$c \times (AvgQ + 6P + I)$
Pure LFU	$M_{lfu}^{pure}(c)$	$M_{lfu}^{inmem}(c) + M^{pure}$
Tiny LFU	$M_{lfu}^{tiny}(c)$	$M_{lfu}^{inmem}(c) + M^{tiny}(c)$
In-memory GDSF-K	$M_{gdsf}^{inmem}(c)$	$c \times (AvgQ + 6P + L + I) + L$
Pure GDSF-K	$M_{gdsf}^{pure}(c)$	$M_{gdsf}^{inmem}(c) - c \times I + M^{pure}$
Tiny GDSF-K	$M_{gdsf}^{tiny}(c)$	$M_{gdsf}^{inmem}(c) - c \times I + M^{tiny}(c)$

to keep the access order of queries from least recently accessed to the most where the head of the list is the victim. In order to query the existence of items, a hash-table is used as a mapping from query strings to pointers to the linked-list nodes. Based on the parameters given in Table 2.1, we provide the worst-case space requirements for our LRU implementation in Table 2.2.

2.5.2.2 Least Frequently Used (LFU)

LFU chooses the least frequent item to be the victim and similarly it is evicted when the cache is full. Usually, there are two ways for storing the frequency. Frequencies can be either tracked through history (as in Pure and Tiny LFU) or only for the cached items *i.e.*, the *In-memory LFU*. Similar to LRU, a list of doubly-linked lists keep the order of nodes but for an efficient eviction these lists are connected with special frequency nodes (cf. [21] for details). Worst-case space requirements occur when the number of frequency nodes are equal to the cached items (and hence, we have the storage cost component $6P$ in Table 2.1). The second LFU variant, *Pure LFU*, keeps the full frequency history of queries in raw format and thus requires a hash table that maps query strings into frequencies (incurring the storage cost of M^{pure}) in addition to the cost of storing the aforementioned linked lists of typical LFU.

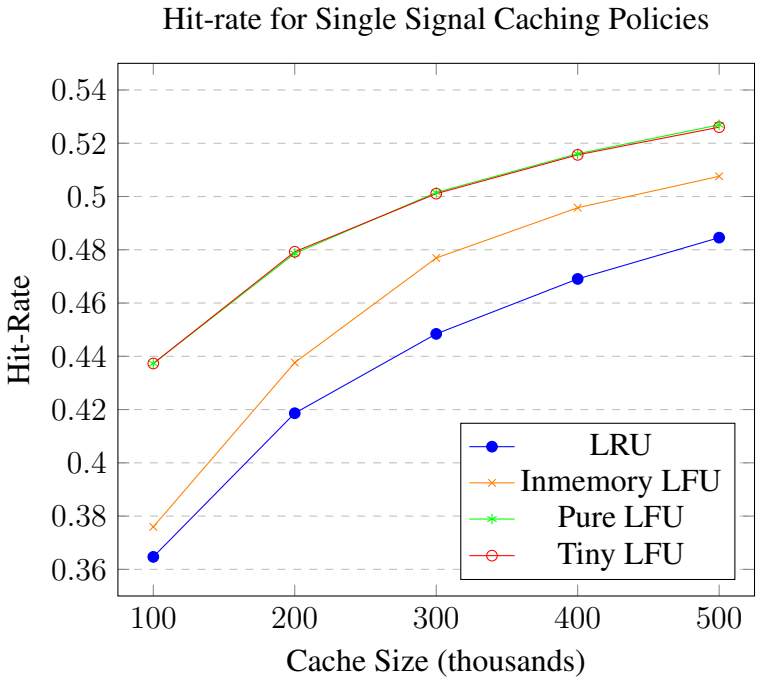


Figure 2.4: Hit-rates for LRU and LFU variants (In-memory, Pure and Tiny).

The *Tiny LFU* variant is the implementation that employs the Tiny storage scheme that is described in Section 2.4.3 for query history. In our setup, we allocate $6W$ bits to doorkeeper and $3W$ full counters (where W denotes the window size and is in range $[8M, 16M]$ in our experiments) and use 4 hash functions to minimize the false positive errors [13] (and thus obtain M^{tiny} as in Table 2.1). For both Pure and Tiny LFU, during warm-up and testing, frequency statistics are updated as queries are streamed.

Figure 2.4 shows the hit-rates of single signal caching algorithms with respect to the cache size. Pure LFU improves the hit-ratio compared to In-memory LFU and LRU, especially for small cache sizes. This answers our first question; keeping the full history improves LFU performance. This is an important finding, because although some earlier works mentioned using a larger query history than the memory size for LFU in search engine result caching (e.g., [4]), there was no experimental evidence, as we provide here. More crucially, we see that Pure and Tiny LFU perform almost the same (i.e., the curves in Figure 2.4 overlap) indicating that using the compact storage scheme (and hence, less precise frequency values) in the latter case do not reduce the cache performance.

Table 2.3: Performance of LFU and GDSF-K variants, $\mathcal{C}$: cache size, $\mathcal{M}$: metadata (query history) size (in MBs), $\mathcal{R}$: result cache size (in MBs), $\mathcal{H}$: hit-rate.

$\mathcal{C}$	$\mathcal{R}$	**LFU**				**GDSF-K**					
		Pure		**Tiny**		**In-memory**		**Pure**		**Tiny**	
		$\mathcal{H}$	$\mathcal{M}/\mathcal{R}$	$\mathcal{H}$	$\mathcal{M}/\mathcal{R}$	$\mathcal{H}$	$\mathcal{M}/\mathcal{R}$	$\mathcal{H}$	$\mathcal{M}/\mathcal{R}$	$\mathcal{H}$	$\mathcal{M}/\mathcal{R}$
$100K$	244.1	0.436	0.56	0.437	0.15	0.422	0.02	0.451	0.56	0.457	0.12
$200K$	488.3	0.479	0.29	0.479	0.08	0.46	0.02	0.498	0.29	0.498	0.11
$300K$	732.4	0.502	0.20	0.501	0.08	0.483	0.02	0.518	0.2	0.518	0.08
$400K$	976.6	0.516	0.15	0.516	0.06	0.497	0.02	0.529	0.15	0.529	0.07
$500K$	1220.7	0.527	0.13	0.526	0.05	0.508	0.02	0.536	0.13	0.536	0.05

Next, we report the ratio of the storage for metadata, M, (i.e., size of the query history stored in plain format or using the Tiny scheme) to that for the actual content of the result cache, R. For a given cache that can store C queries, the cache (content) size R is computed as $\mathcal{R} = C \times k \times S$. We set $k = 10$ (since a typical result cache includes top-10 answers) and $S = 256$ as each query result including document's title, URL and snippet may add up to 256 bytes, as in [3]. Obviously, for a fixed cache size, the denominator R is the same for both Pure and Tiny LFU. Table 2.3 shows that M/R ratio varies from 0.56 to 0.13 for the smallest and largest values of C, i.e., 100K and 500K queries, respectively. In contrary, for Tiny LFU, M/R ratio is much smaller, between 0.15 and 0.05, indicating that Tiny scheme allows considerable gains in memory space. Note that, this is a finding shown for caching in other application domains [5] but not for search result caching. To our best knowledge, only [22] mentioned the possibility of using BFs for storing the query history in result caching, but their work did not provide any experimental evaluation of this idea, while we provide an exhaustive analysis in terms of the storage and cache performance metrics.

2.5.3 Results for Multi-Signal Caching Policy

2.5.3.1 Greedy Dual Size Frequency(GDSF)

GDSF [20] offers a good compromise between recency and frequency, as well as the entry size and cost. We use a variant of GDSF, so called GDSF-K [23], where the frequency component is weighted by an exponent K to balance for the power-

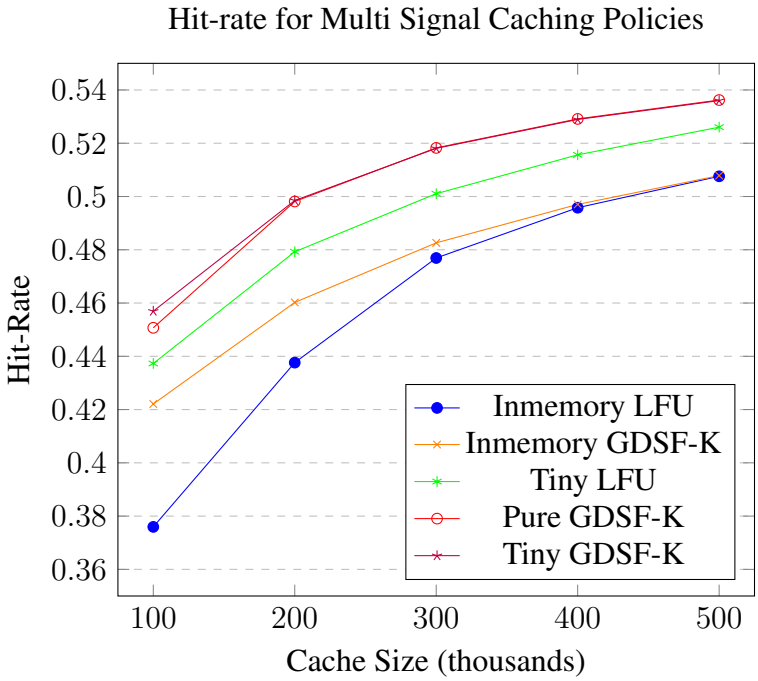


Figure 2.5: Hit-rates for GDSF-K variants (In-memory, Pure and Tiny).

law distribution of queries in our setup. GDSF-K evicts the query result i with the minimum $\mathcal{H}_i$ value where $\mathcal{H}_i = \mathcal{F}_i^K \times \frac{\mathcal{C}_i}{\mathcal{S}_i} + \mathcal{L}$. In this equation, $\mathcal{F}_i$ is the frequency of the query, $\mathcal{C}_i$ and $\mathcal{S}_i$ are size and cost of the query result, respectively. In our experiments, result size is set to 1, whereas we consider different alternatives for the cost (as will be discussed later). Lastly, $\mathcal{L}$ is the aging factor that is updated to the $\mathcal{H}_i$ of the cache victim whenever an item is evicted.

We again have three variants, namely In-memory (frequency information is kept only for the queries in the cache), Pure (with a raw frequency history) and Tiny. Table 2.2 presents the worst-case memory space usage of these variants.

While experimenting with GDSF-K variants, we first assume that all query costs $\mathcal{C}_i$ as 1, and measure the traditional hit-rate. Figure 2.5 reveals that, as before, the variants using the entire history outperform the In-memory GDSF-K, and furthermore, Pure and Tiny variants of GDSF-K perform comparably. We also see that GDSF-K formulation that combines both frequency and recency is superior to using each on its own; i.e., Pure (or, Tiny) GDSF-K outperforms the corresponding case with LFU. Finally, Tiny again provides considerable memory space gains over Pure GDSF-K, as shown

in Table 2.3.

To investigate our third research question, i.e., assessing the value of the saved memory space, we focus on Tiny GDSF-K as the best-performing policy in the previous experiments. To this end, we first assume the saved space (i.e., the difference of memory space between Pure and Tiny variants) is also filled with query results, and measure the hit-rate. This case is denoted as 'Tiny GDSF-K with Result' in Table 2.4, and we see that the hit-rates improve for each cache size (absolute gains being around 1.7% for the smallest cache size in comparison to the hit-rate of Tiny GDSF-K column, repeated from Table 2.3 for reference).

Furthermore, following the recent trend [23] that suggests using cost-aware caching policies and evaluating them again in terms of the cost savings –rather than hit rate-, we also experiment with Tiny GDSF-K using simulated query costs (i.e., in GDSF-K formula, we now set $\mathcal{C}_i$ using such costs). While doing so, as in [3], we consider four basic cost components: fetching the posting lists from the disk, ranking in memory, fetching top-k documents from the disk and generating snippets. Since modern search engines are known to store most of their data in memory or SSDs, here we focus on the ranking and snippet generation components. Formally, we define the former one as; $C^{rank} = |I_s| \times (P_d + P_r)$ and the latter one; $C^{snip} = 10 \times d_{avg} \times P_s$, where I_s is the shortest posting list of terms in query q (representing the conjunctive processing in the search engine, as typical) and P_d, P_r, P_s and d_{avg} are defined in Table 2.1 based on [3]. Posting list length of each query term is obtained from an index over the well-known Clueweb-2009 Part B collection.

In this case, our evaluation metric is the savings in query processing cost, C_s. To compute C_s, we obtain the cost of running a query stream over a system with a given type of cache divided by that of a system with no cache (where the cost of a query is simply the sum of C^{rank} and C^{snip}), and subtract the latter ratio from 1.

In Table 2.4, we see that exploiting the saved space as a result cache further improves cost savings of Tiny GDSF-K; e.g., for a cache of 100K results, the cost savings increase from 0.698 to 0.710 (the former value, cost savings of the Tiny GDSF-K, is computed as a baseline).

Table 2.4: Cost saving, $\mathcal{C}_s$, when saved space (by Tiny) is reserved for extending the result cache (Tiny GDSF-K w. Result) and implementing a docID cache. $\mathcal{H}$: hit-rate.

$\mathcal{C}$	**Tiny GDSF-K**		**Tiny GDSF-K w. Result**		**Tiny GDSF-K w. Doc-ID**
	$\mathcal{H}$	$\mathcal{C}_s$	$\mathcal{H}$	$\mathcal{C}_s$	$\mathcal{C}_s$
$100K$	0.457	0.698	0.474	0.710	0.740
$200K$	0.498	0.724	0.504	0.728	0.744
$300K$	0.518	0.734	0.521	0.736	0.746
$400K$	0.529	0.739	0.53	0.740	0.747
$500K$	0.536	0.742	0.537	0.743	0.748

Note that, recent studies in the literature propose using hybrid result caches, i.e., some part of the cache stores only doc-IDs of the query results [1, 3]. In this case, if the cache-hit is for the latter part, there will be still C^{snip} cost, but much more expensive C^{rank} will be avoided. Therefore, as a final experiment, we explore what happens if the memory space saved by Tiny is reserved as a doc-ID cache, while the original cache capacity, as in previous experiments, store the query results in HTML format (i.e., top-k results' URLs and snippets). With this hybrid cache, we employ the Second Chance algorithm that is basically intended to keep the doc-ID results of a query for a longer time even its HTML version is evicted from the cache (please refer to [3] for details).

Our findings for the latter experiment are presented in Table 2.4 with column denoted as "Tiny GDSF-K with Doc-ID". We see that in terms of cost savings, there is an absolute improvement of up to 3% (note that, since the cache includes both HTML and docID results, hit-rate is meaningless, and not reported here). Furthermore, by reserving this saved space as a Doc-ID cache, the cost saving in case of a 100K cache is as good as that of a 500K result cache. These final experiments answer our third research question: the memory space saving using Tiny can yield non-negligible gains in terms of both hit-rate and query processing cost, especially for small and moderate size caches.

2.6 Conclusion

In this chapter, we introduced BFs and BF-like data structures such as sketches, then described Tiny storage scheme and our configuration of it. We couple Tiny with well-known cache eviction policies and our exhaustive experiments reveal that i) Storing the entire query frequency history yields better hit-rate which was only proposed but not experimentally supported by previous work, ii) *Tiny* successfully reduces memory space for the history, and iii) The space saving is valuable, as it can be exploited to yield non-negligible gains in hit-rate and query processing cost can be further reduced when the gained spaced is used for an additional doc-ID cache.

CHAPTER 3

THE SCORE CACHE

3.1 Introduction

Modern Web search engines adopt a two-stage ranking strategy to process top-k queries [24]. The first stage typically employs an inverted index and ad hoc scoring functions (such as BM25) to generate a small set of candidate documents, while the second stage re-ranks these candidates using machine-learned models.

To cope with the huge index size and demanding requirements for throughput and response time [25], search engines distribute the index across several nodes, so that each node processes the query on its own portion of the index, in parallel. Still, the overall query processing cost is likely to be dominated by the first stage retrieval, where traversing and scoring the large number of documents in the posting lists take place [25, 24]. To speed up the latter stage, various dynamic pruning (aka., early termination) strategies are proposed in the literature (e.g., [26]). These strategies aim to skip scoring certain documents in the posting lists to improve the processing efficiency. A rank-safe pruning strategy guarantees exactly the same result as the exhaustive processing of the query, while unsafe strategies may sacrifice effectiveness in return for higher efficiency.

In this chapter, we focus on two such strategies, namely, WAND (Weak-AND) [26] and its successor, BMW (Block-Max WAND) [26], and their application in a rank-safe scenario for (ranked) disjunctive queries (as in [26, 27]). In a nutshell, both strategies assume that the maximum possible score contribution (e.g., using BM25) for each term (either over the entire posting list [26], or the blocks of a list [26]) is pre-computed. During query processing, they keep track of the k·th highest score so

far, the *threshold*. If a document's estimated score (based on the upper-bound scores of query terms) cannot exceed the threshold, the document is skipped; otherwise, its actual score is computed to see whether it can get into the current top-k list. WAND and BMW employ intelligent pivoting strategies that can jump to the first document with an estimated score that exceeds the threshold, rather than checking each document in the lists.

We propose to improve the performance of WAND and BMW strategies by exploiting an orthogonal technique, namely, caching. A search engine employs various types of caches at different layers of its architecture [1]. Result caches including the final SERPs for queries are located at the broker nodes [7, 2, 3], while caches with posting lists are located at the index nodes [2, 28].

As our first contribution in this chapter, we introduce a new cache type to be located at the index nodes, *i.e.*, the *score cache*, which will store the score of the k·th result of a query computed at each node. The significance and novelty of our approach lies in exploiting cached scores for previously unseen queries. For the queries seen before, the answer will be provided from the result cache in the broker with virtually no cost, as usual. In contrary, for a query that is not found in the result cache and forwarded to the index nodes, we generate its subsets and probe the score cache to locate those that are cached. The scores of these subset queries are exploited to obtain a lower-bound for the score threshold, which is fed to the dynamic pruning strategy, to allow skipping more documents.

Since considering all subsets of a query can cause additional run-time overhead, as our second contribution, we introduce several heuristics to obtain the score threshold efficiently. These heuristics aim to generate subsets of a query that are most likely to be cached, using the features such as the query length. In our simulations, we fill the result and score caches using 1.8M distinct queries, and measure the performance for around 50K test queries that are result cache misses. In comparison to recent works that evaluate WAND and BMW [27, 29], our test query set is at least an order of magnitude larger. Furthermore, our evaluations provide realistic/practical insights on the query processing costs with dynamic pruning, as we take into account the impact of the other system components, such as the filtering effect of result caches at the

broker nodes.

Our experiments reveal that using the score thresholds obtained by our heuristics, query processing cost at a node can be reduced by up to 8.6% (and further reductions are possible when our heuristics are combined with a previously proposed strategy). The space overhead for storing scores (per node) is reasonable; *i.e.*, around only 1% for a typical top-10 result cache. We also show that using this space in favor of a larger result cache may not always improve the efficiency (see Sec. 3.4 for experimental evidence). This is because there is an experimental upper-bound for the hit-ratio of a search engine's result cache, which is around 50% (as 44% of a query stream is made up of singleton queries, which are submitted only once, and there are also compulsory misses [2]). Indeed, we believe that the latter point is crucial to make our contributions worthwhile for practical systems: Our approach would improve the efficiency for the singleton queries (as well as compulsory misses) that can never benefit from a result cache, even if the latter is infinitely large.

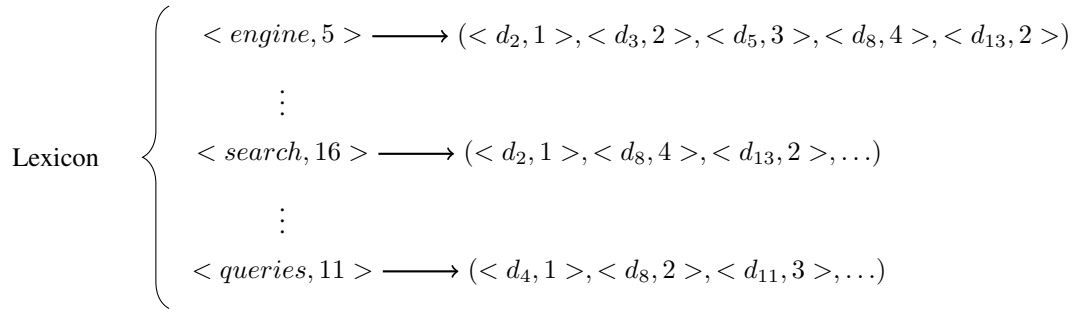
In the next section, we review background and related studies. Sec. 3.3 describes the score cache and associated heuristics for query processing. In Sec. 3.4, we provide experimental results and conclude.

3.2 Background and Related Work

3.2.1 Inverted Index & Query Processing

A document-ordered inverted index is composed of two main parts: First part contains the terms that all together create the lexicon of the document collection. Each term is paired up with a pointer which points to the second part of the inverted index *i.e.*, the postings lists. Posting lists are lists that contain document identifiers (doc-ids) in an ascending order only if the document contains the corresponding term. Each doc-id in the postings lists can contain additional information such as the term frequency that is used in the ranking stage. Inverted index layout enable various types of optimizations to be applied over the index in terms of both space and time. Figure 3.1 illustrates an inverted index where each term-document frequency (df) pair and its corresponding

Figure 3.1: A Simple Inverted Index



postings list is shown with an edge between. Postings lists are ordered pairs of doc-ids and term frequencies.

Query processing can be divided into two regarding the creation of the query result set. Conjunctive (a.k.a, AND) processing requires that; for a document to be in the result set it has to exist in the all of the terms' postings lists. For instance, in Figure 3.1, if we would process query $q = \{search, engine, queries\}$ the result set would include only documents $\{d_8\}$, discarding any other document since they are not included in all terms of q. Conversely, disjunctive (a.k.a OR) processing requires only a single term to exist in any of the posting lists of q. Using the previous example we would obtain the result set $\{d_2, d_3, d_4, d_5, d_8, d_{11}, d_{13}\}$. Typically, conjunctive processing can be more efficient by skipping many documents without computing scores, since a document has to be in the all of the postings lists to be scored. However, for a query that has many terms, the length of the result set might not be sufficient to retrieve k documents due to most of the documents being eliminated in consecutive intersection of posting lists. Therefore, disjunctive processing is a viable option although being less efficient compared to the conjunctive alternative, especially for long queries *i.e.*, queries with many terms.

Another way to categorize query processing is by the processing order of terms. Again the index can be traversed in two ways, either term-at-a-time (TaaT) or document-at-a-time (DaaT) manner. In this thesis, we focus on the latter case. DaaT approach fetches the postings lists of all the terms in the query, and traverses these lists in parallel by maintaining pointers to the each document in the postings list whereas TaaT processes a single posting list before moving on to the other and thus requires an ad-

ditional accumulator array to keep partial scores of documents. Appendix A supplies algorithms for possible different strategies for ranked retrieval over an inverted index.

In ranked disjunctive processing (aka., Ranked-OR) using a retrieval model like BM25, a score is computed for each document in each list, *i.e.*, as if merging the lists by doc-ids. Note that, in DaaT, a document is fully scored before moving the next document; hence, it is efficient to maintain a top-k min-heap during query processing, see Algorithm 2 for an example pseudocode. If a document's score is larger than the k·th document's score in the heap, the latter is extracted from the heap and the newly scored document is inserted; otherwise, the document is simply discarded, see Algorithm 1 for a pseudocode that adds doc-id, score pairs to the heap.

Algorithm 1 Inserting a doc-id δ to a k-size min-heap together with its score σ

```
function HEAP-INSERT(H, δ, σ, k)
    if |H| = k then
        (τ, θ) ← TOP(H)      ▷ Top doc-id and its score which is the heap threshold
        if σ > θ then
            POP(H)                          ▷ Remove doc-id with minimum score
            INSERT(H, δ, σ)
        end if
    else
        INSERT(H, δ, σ)
    end if
end function
```

3.2.2 Index Pruning

In order to speed up the query processing, the documents that wouldn't make it into the top-k can be skipped *i.e.*, not computed a score. Index pruning methods are usually applied in two different stages; index creation (static pruning) or query processing (dynamic pruning). Static pruning techniques remove documents from postings lists entirely during index creation and thus also reduce the size of the index, optimizing for both space and time. Conversely, dynamic pruning techniques may increase the size of the index by keeping maximum scores that can be contributed by a term of en-

Algorithm 2 Ranked-OR in a DaaT manner

function RANKED-OR($\mathcal{Q}, \mathcal{I}, k$) ▷ Parameters query, index and the number of doc's to retrieve

  $\tau \leftarrow |\mathcal{Q}|$

  for $t = 1$ to τ **do**

    $\mathcal{L}_t \leftarrow \mathcal{I}_{\mathcal{Q}_t}$ ▷ Initialize postings lists

    $\mathcal{P}_t \leftarrow 1$ ▷ List pointers initialized to first elements

  end for

  $\mathcal{H} \leftarrow$ HEAP()

  while $\exists t \in [1, \tau], \mathcal{P}_t \leq |\mathcal{L}_t|$ **do** ▷ Loop until all postings lists are exhausted

    $\mu \leftarrow \min\{\text{DOCID}(\mathcal{L}_{t,\mathcal{P}_t}) \mid t \in [1, \tau]\}$

    $\sigma \leftarrow 0$ ▷ Score of μ

    for $t \leftarrow 1$ to τ **do** ▷ Compute the score

      $\gamma \leftarrow \mathcal{L}_{t,\mathcal{P}_t}$

      if $\text{DOCID}(\gamma) = \mu$ **then**

        $\psi \leftarrow \text{WEIGHT}(\gamma)$

        $\sigma \leftarrow \sigma + \text{SCORE}(\psi, \mathcal{I}, \mathcal{Q}_t)$ ▷ Accumulate partial scores

        $\mathcal{P}_t \leftarrow \mathcal{P}_t + 1$ ▷ Advance list pointer to next doc

      end if

    end for

    HEAP-INSERT($\mathcal{H}, \mu, \sigma, k$) ▷ See Algorithm 1

  end while

  return CREATE_ANSWERS($\mathcal{H}$)

end function

tire postings lists and/or for chunks of the postings lists. However, it can be difficult to determine which documents to statically prune from the index independent of the queries. Here, our ideas apply to dynamic pruning strategies and we provide results for two well-known approaches *i.e.*, WAND and BMW, for faster DaaT processing in a rank-safe scenario for Ranked-OR retrieval. We summarize WAND and BMW, as follows (see [27] for a performance comparison of these strategies):

3.2.2.1 WAND

In this algorithm, for each posting list, the maximum possible score, U_t, that can be contributed by its postings (based on an additive retrieval model, like BM25) is pre-computed and stored. During query processing with Ranked-OR, each document is scored as usual, until there are k documents in the heap. After this point, in each round, the posting lists are sorted in ascending order of doc-ids that are pointed to, *i.e.*, to be processed next. The U_t scores of the lists are summed until the sum exceeds the threshold, i.e., the k·th score in the heap. The document pointed in the last list (that contributed to the sum) is selected as the pivot, i.e., this is the first document that has a potential to get into the heap. Thus, all lists move their pointers to the pivot document, so that the true score of the document can be computed and compared against the heap threshold. Algorithm 3 summarizes this procedure to give an idea.

3.2.2.2 BMW

Instead of using a single (and possibly large) U_t for the entire list as in WAND, BMW partitions the posting list into *blocks* and computes an upper-bound score for each block. Due to this modification, the pivot selection and skipping becomes slightly more complicated. In return, BMW computes a tighter bound on the estimated score of a document, and hence, can skip larger number of documents by skipping entire blocks, or documents within blocks without actual score computation.

Algorithm 3 WAND Algorithm

function WAND($\mathcal{Q}, \mathcal{I}, k$) ▷ Parameters query, index and the number of doc's to retrieve
  $\tau \leftarrow |\mathcal{Q}|$
  for $t = 1$ to τ **do**
    $\mathcal{L}_t \leftarrow \mathcal{I}_{\mathcal{Q}_t}$ ▷ Initialize postings lists
    $\mathcal{U}_t \leftarrow$ MAX-SCORE$(\mathcal{I}_{\mathcal{Q}_t}, k)$ ▷ Initialize maximum scores for each term
    $\mathcal{P}_t \leftarrow 1$ ▷ List pointers initialized to first elements
  end for
  $\theta \leftarrow 0$ ▷ Initialize heap threshold
  $\mathcal{H} \leftarrow$ HEAP()
  while $\exists t \in [1, \tau], \mathcal{P}_t \leq |\mathcal{L}_t|$ **do** ▷ Loop until all postings lists are exhausted
    $\phi \leftarrow 0, \lambda \leftarrow 0, \rho \leftarrow 0$ ▷ Initialize sum of max term scores, pivot term and pivot doc-id
    $\mathcal{S} \leftarrow \arg\operatorname{sort}(\mathcal{L}_{t,\mathcal{P}_t}, t \in [1, \tau])$ ▷ Argument sort by ascending doc-id's
    for all $t \in \mathcal{S}$ **do**
      $\phi \leftarrow \phi + \mathcal{U}_t$ ▷ Accumulate maximum scores
      if $\phi > \theta$ **then**
        $\lambda \leftarrow t$ ▷ Set pivot term
        $\rho \leftarrow$ DOCID$(\mathcal{L}_{t,\mathcal{P}_t})$ ▷ Set pivot doc-id
        break
      end if
    end for
    if $\lambda = 0$ **then**
      break ▷ When pivot term is 0 no doc-id can enter heap
    end if
    $\alpha \leftarrow true$ ▷ Initialize flag to check all doc-ids equal to pivot doc-id
    for all $t \in \mathcal{S}$ **do**
      if $t = \lambda$ **then**
        break
      end if
      while DOCID$(\mathcal{L}_{t,\mathcal{P}_t}) < \rho$ **do**
        $\mathcal{P}_t \leftarrow \mathcal{P}_t + 1$ ▷ Advance pointers until equal or greater doc-id
      end while
      $\alpha \leftarrow$ DOCID$(\mathcal{L}_{t,\mathcal{P}_t}) = \rho$ ▷ Update α if current doc-id is equal to pivot
    end for
    if α **then**
      Compute score σ for ρ as usual
      HEAP-INSERT$(\mathcal{H}, \sigma, \rho, k)$
      Update θ to heap minimum
    end if
  end while
  return CREATE_ANSWERS$(\mathcal{H})$
end function

3.2.3 Term-score (TS) Strategy

One of the closest approaches to our work proposes pre-computing the k·th highest score for each *term*, and storing them in the index [30, 29]. While processing a query $q = \{t_1, \ldots, t_k\}$, the score threshold is set to the maximum of these term scores (TS), i.e., threshold (q) = $\max(TS(t_1), TS(t_2), \ldots, TS(t_k))$. Although there are similarities, our approach differs from the latter in two critical ways. First, by leveraging a score cache, we propose a dynamic setting: As the k·th score values for the cached queries can be computed on the fly (when the query is first submitted), there is no need for pre-computing and maintaining the term scores. That is, in our approach, if the scoring model of the search engine is changed, it is adequate to flush the cache, and new scores will accumulate in the cache as queries arrive, as typical. In contrary, storing term-score values in the index will require re-scoring all the posting lists. A second and more crucial difference is that, by caching scores for multi-term queries, it is possible to obtain tighter lower-bounds for the score threshold of an unseen query. In Sec. 3.4, we first justify the latter intuition (see Table 3.1) and also experiment with a hybrid strategy that combines the TS approach with ours.

3.2.4 Predicting the Threshold Score

In a recent work, Petri *et al.* [31] proposed predicting a lowerbound threshold for each query to speed up the query processing in a similar manner to ours. To make predictions, the index has to be extended to keep additional feature information such as k-th highest score and the inverse document frequency (IDF). If the prediction is larger than the actual threshold *i.e.*, an over-prediction for the threshold yields an unsafe-ranked retrieval. To overcome this problem authors detect the over-prediction during the query processing, and recalculate the threshold in a more conservative manner. Indeed, this process requires the query to be processed again and too many repetitions can easily shadow the benefits of obtaining a lowerbound threshold. Nevertheless, they report similar improvements to ours compared to the TS strategy which is explained in Sec. 3.2.3. However, their improvements are only limited to BMW unlike ours; where we show that our query processing gains are significant for both WAND and BMW and are easily applicable to other dynamic pruning strategies.

Finally, in a complementary work to ours, Fontoura et al. [32] proposes to compute the score threshold using only few terms (*i.e.*, those with short lists) of a given query, and then continue processing with the remaining terms and computed threshold. In contrary, we exploit the score thresholds for previously seen queries, but do not make any document scoring to obtain them. In a sense, their approach is intra-query while ours is inter-query; and experimenting with a hybrid approach seems as a promising future direction.

3.2.5 Caching in Search Engines

A result cache is typically located at the broker nodes of a search engine, and stores top-k results as SERPs (i.e., with titles, URLs and snippets for each document [7, 2]) and/or in terms of the doc-ids [7, 1, 3] (Note that, in [1], the latter, a *result cache including docIDs*, is called as a *score cache*; while we also prefer to call our cache the same, we clearly mean something different.). Posting lists and their intersections can be cached at index nodes to speed up the ranking process [2, 28].

Caches are broadly categorized as static or dynamic. As the name implies, the content of a static cache does not change during run time, until the next periodic update. In contrary, as the new queries arrive, items stored in a dynamic cache are evicted and new ones are inserted. In the literature, purely static or dynamic caches as well as their hybrids for aforementioned data types in search engines are exhaustively evaluated(e.g., [7, 2, 1, 3]. However, as far as we know, none of these earlier works propose to cache the score of k·th query result to accelerate dynamic pruning.

3.3 Score Caching for Dynamic Pruning

We assume a typical distributed search setup (e.g., see [25]) as shown in Fig. 3.2. When a new (previously unseen) query arrives to a broker node, it forwards the query to all index nodes. Each index node executes the first-stage ranking (with WAND or BMW) and obtains its local top-k rankings, that are sent back to the broker. The broker creates the candidate set, on which second-stage rankers are executed to obtain the final top-k results. The final ranking is sent to the user, and also stored in the result

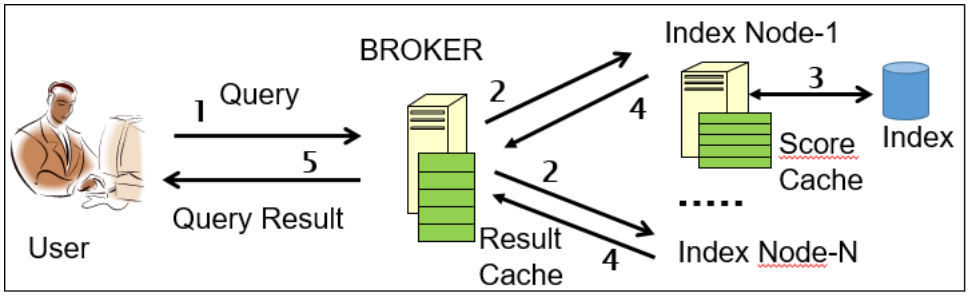


Figure 3.2: Search Engine Architecture with Score Caches

cache at the broker, so that if the same query is re-submitted, it can be answered from the cache.

We introduce a new cache type, score cache, to be located at the index nodes, as also shown in Fig. 3.2. For each query processed at an index node n, its score cache stores the pair <q, s>, i.e., the query string and score of the k·th result of q at n. In this modified search architecture, for a previously unseen query (i.e., causing a cache-miss at the broker), our query processing algorithm exploits the score cache as follows.

First, we generate a number of subsets for the query, and by looking up these subsets in the cache, we determine a lower-bound for the score threshold of dynamic pruning algorithm.

Example. Assume that the score cache at a particular node includes the entries: q_{c1} :(“search engine", s_1), q_{c2} : (“search engine books", s_2), q_{c3} : (“search engine resources", s_3). The new query q_n: “search engine books surveys" will cause a cache-miss for the result cache at the broker (as the result and score caches have the same keys, i.e., query strings, but with different values). When the query arrives to the index node, all of its subsets in the score cache are found, i.e., q_{c1} and q_{c2}. Then, the score threshold of q_n is computed as max(s_1, s_2). This approach is rank safe, as long as the same retrieval model is applied for processing the cached and new queries.

Obviously, generating all subsets of a query and probing the cache for each can be expensive (see Table 3.2), especially for the long queries. Therefore, we propose three different heuristics to generate a query’s subsets and compute its score threshold.

- **Heuristic-1 (HR1)** In this heuristic, we exploit the observation that a typical

score cache is more likely to include frequently asked queries, and frequent queries are more likely to be shorter (i.e., up to 3 terms), as rare queries (such as singletons) are typically longer (e.g.,see [3]). Indeed, around $\%87$ of the queries employed in our experiments are shorter than 5 terms. Thus, our first heuristic works as follows. Considering our query as a set of terms, we first generate its 3-term subsets (i.e., sub-queries with exactly three terms) and look for them in the score cache. If one or more of these subsets are found in the cache, we set the score threshold to the maximum score of them. If cache probing fails, i.e., no 3-term subsets of this query are cached, we repeat the same for 2-term and eventually 1-term subsets. If none of them reside in the score cache, we initialize the score threshold to 0, which is the typical setup in WAND or BMW. Note that, we prefer to search the cache starting from the subsets with more terms, as we expect that a hit for a longer query would yield a higher threshold, and allow skipping more documents.

- **Heuristic-2 (HR2)** This heuristic is similar to HR1, but this time we conduct an exhaustive search of all 3-term, 2-term, and 1-term subsets of a query. That is, we do not quit searching even when we a find subset with more terms. This is a more expensive approach, but we aim to catch the cases where a smaller subset may have a larger score. For instance, consider a query $q = \{a, b, c\}$ of which subsets $q_1 = \{a, b\}$ and $q_2 = \{c\}$ are in the score cache, with score values of 6 and 7 respectively. Clearly, if we stop when we find the larger subset, q_1, we are settling for a smaller threshold, while proceeding with smaller subsets yields a larger threshold.

- **Heuristic-3 (HR3)** For a query with N terms, one can expect that highest scores are likely to belong to its largest, i.e., $(N-1)$-term, subsets, of course, if they are found in the cache. This heuristic is expected to run faster than the previous two heuristics, as its complexity is only $O(N)$, yet its success will depend on the cache hit-ratio for the $(N-1)$-term subsets.

3.4 Evaluation

3.4.1 Setup

We used the standard TREC GOV2 collection with $\approx 25M$ documents. We assumed that the collection is stored at a single index node, where the query processing takes place and a score cache is placed. Since query processing would be in parallel across all nodes, our findings for a single node are representative.

We employed the publicly available WAND and BMW implementation[1] that first calls the ATIRE[2] system to build a quantized index for the collection, and then generates index variants that are compatible to work with WAND and BMW processing techniques [27]. We modified the latter codes[3] so that before actual query processing begins, query subsets were generated using the aforementioned heuristics and searched in the score cache, which is implemented as a hash table (i.e., an unordered map in C++). Note that, the latter steps are included while measuring the execution time for a query, so the reported efficiency figures capture all the time overhead for our approach.

Following the practice in [27, 29], we used the quantized BM25 scores and kept stop words both in the index and queries. To obtain the query sets, the well-known AOL log in time-stamp order is employed. We used 4 million consecutive queries ($\approx 1.8M$ distinct queries) to fill the static result and score caches[4]. Next $100K$ queries (again in time-stamp order) constitute our test set. Note that, due the presence of a result cache in our framework, cache-hits are assumed to be answered at the broker; and we simulated this by simply filtering the test query log, i.e., removing all occurrences of the training queries (resulting in $\approx 50K$ queries) from the latter set. Thus, our evaluation is based only on the result cache misses.

We conducted experiments on a HP Z840 Workstation with two Intel Xeon E5-2630

[1] `https://github.com/JMMackenzie/Quant-BM-WAND`
[2] `http://www.atire.org`
[3] `https://github.com/yfy-/Quant-BM-WAND`
[4] While the result cache may be dynamic in practice (and we have such an experiment reported later), dynamicity is less mandatory for the score cache: since the latter won't provide query answers but just lower-bounds for thresholds, it may be adequate to update it periodically. Hence, we leave the dynamic score caching as a future work.

Table 3.1: Median, 99th percentile and mean processing time (ms) of WAND and BMW for $k \in \{10, 1000\}$. Original and TS are baselines, Heuristics HR1-3 are proposed. The best result for each k in each row is shown in boldface. The means that are statistically significantly different (using paired t-test at 0.05 level) wrt. ORG and TS are denoted with β and $*$, respectively.

Method	Time	ORG		TS		HR1		HR2		HR3		HR2+TS	
		10	1000	10	1000	10	1000	10	1000	10	1000	10	1000
WAND	P_{50}	7.2	27.3	6.4	21.4	6.5	21.3	6.5	20.3	6.8	24.2	**6.0**	**19.7**
	P_{99}	227.7	456.9	**223.3**	456.1	223.7	451.7	223.7	452.3	225.1	452.8	225.1	**453.3**
	Mean	22.3	58.8	21.5^{β}	55.5^{β}	21.6^{β}	$54.8^{\beta*}$	21.6^{β}	$54.3^{\beta*}$	21.9^{β}	56.6^{β}	**21.2**$^{\beta*}$	**54.2**$^{\beta*}$
BMW	P_{50}	6.5	22.4	5.8	17.1	5.8	17.0	5.7	16.2	6.3	19.5	**5.5**	**15.9**
	P_{99}	178.1	417.7	176.2	416.6	175.9	416.8	**175.8**	416.6	179.6	418.8	176.1	**415.0**
	Mean	19.1	52.3	18.5^{β}	49.3^{β}	18.5^{β}	$48.5^{\beta*}$	$18.4^{\beta*}$	$47.8^{\beta*}$	19.0^{β}	50.4^{β}	**18.3**$^{\beta*}$	**47.7**$^{\beta*}$

Table 3.2: Mean processing times (ms) for generating all subsets. Processing time changes are denoted in parentheses compared to ORG.

Method	$k = 10$	$k = 1000$
WAND	$24.0(+7.0\%)$	$58.0(-1.3\%)$
BMW	$21.4(+11.0\%)$	$50.0(-4.3\%)$

CPU, 128 GB of RAM, 512 GB HP Z Turbo Drive PCIe SSD and 4 TB HDD, running Ubuntu Linux v14.04.

3.4.2 Efficiency evaluation

In Table 3.1, we present the *in-memory* execution time statistics (in *milliseconds*) for the query processing with the original pruning strategy (i.e., score threshold is initially set to 0) and with Term-score (TS) strategy [30, 29] vs. processing with our heuristics based on a score cache, for WAND and BMW. We report mean as well as median (P_{50}) and tail (namely, 99th percentile) statistics. Table 3.1 shows that the heuristic HR3, which only checks a small number of subsets, can outperform ORG, but not the stronger baseline, TS. In contrary, the heuristic HR2, which generates

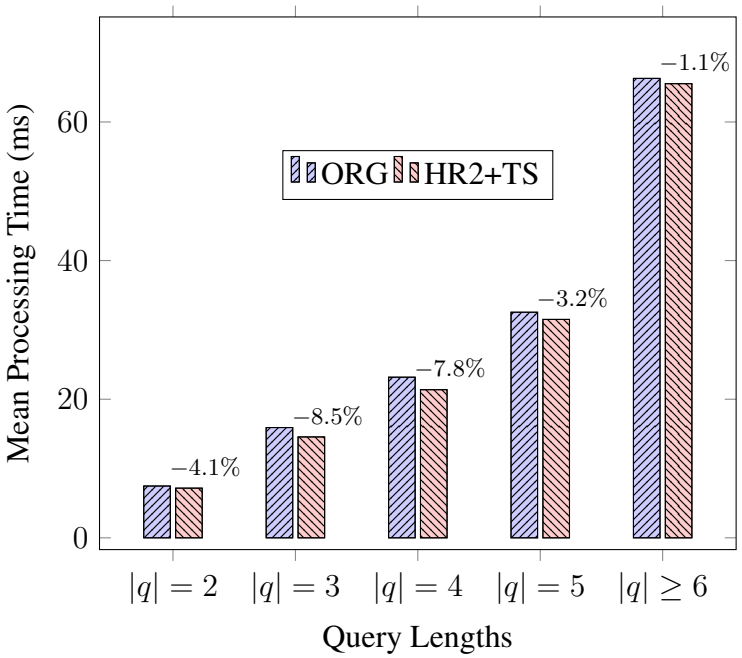
and probes a larger number of subsets than others (and hence, more likely to find a tighter score threshold), is the best performer: It outperforms ORG and TS for WAND (for $k \in \{10, 1000\}$) and BMW (for both values of k) in terms of mean processing time. The gains are more visible for $k = 1000$, where HR2 cuts the mean processing time by 7.7% (8.6%) and 2.2% (3.0%) w.r.t. to ORG and TS baselines for WAND (BMW), respectively. HR2 also outperforms ORG and TS in terms of other statistics (P_{50} and P_{99}) especially for BMW, the faster of two pruning methods employed in our experiments. As an addition to the Table 3.1, we provide mean processing times for the strategy that generates all subsets (a.k.a ALL) and compare it to the ORG in Table 3.2. As we claimed before, such an exhaustive search increases processing times by 7% and 11%, respectively for WAND and BMW when $k = 10$ and for larger k, improvements of ALL are still surpassed by our heuristics.

Recall that, our time measurements include the overhead for generating subsets and probing the cache, which means that these costs are well-compensated by the savings in time spent for actual scoring. Indeed, we found that the overhead of our HR2 is only ≈ 10 *microseconds*, and hence, we do not report these costs separately.

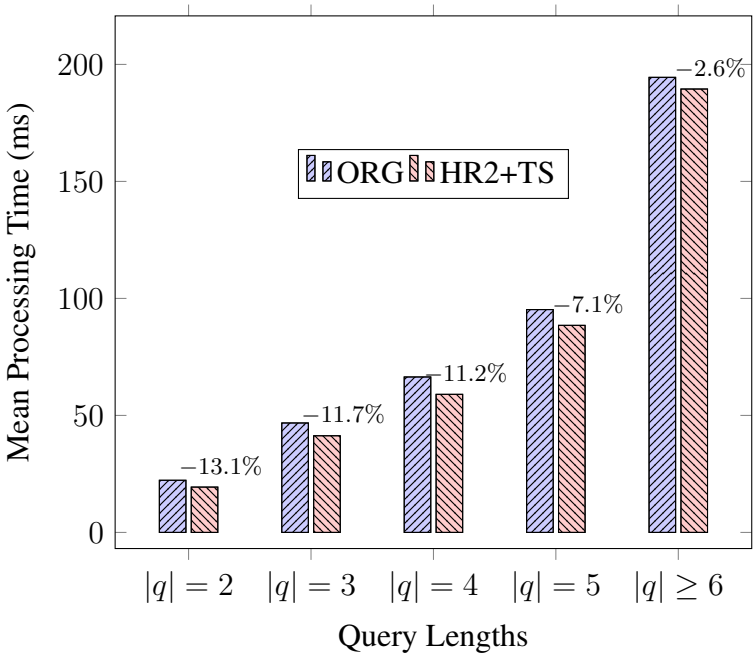
3.4.3 Combining Score-Cache Heuristics with TS

As TS approach [30, 29] can be considered as a special case of our framework (i.e., like a cache with only and all terms in the index), combining them seems to be a promising direction. That is, TS may serve as a last-resort if our heuristics fail to find any subset in the score cache.

In this case, we obtain two score thresholds, one using our best-performing heuristic (namely, HR2) and the other one with TS, and choose their maximum to employ in query processing (as discussed above, overhead is negligible for our heuristics, same applies also for TS). The last column in Table 3.1 presents this hybrid HR2+TS approach. As expected, it yields even larger reductions in mean processing time against both ORG and TS baselines. For instance, in case of $k = 1000$, the hybrid approach outperforms ORG (TS) by 7.8% (2.3%) for WAND, and by 8.8% (3.3%) for BMW, respectively. Note that, as the score cache capacity gets larger, all or most terms of a query may be found in the cache (as 1-term subsets), in which impact of TS may

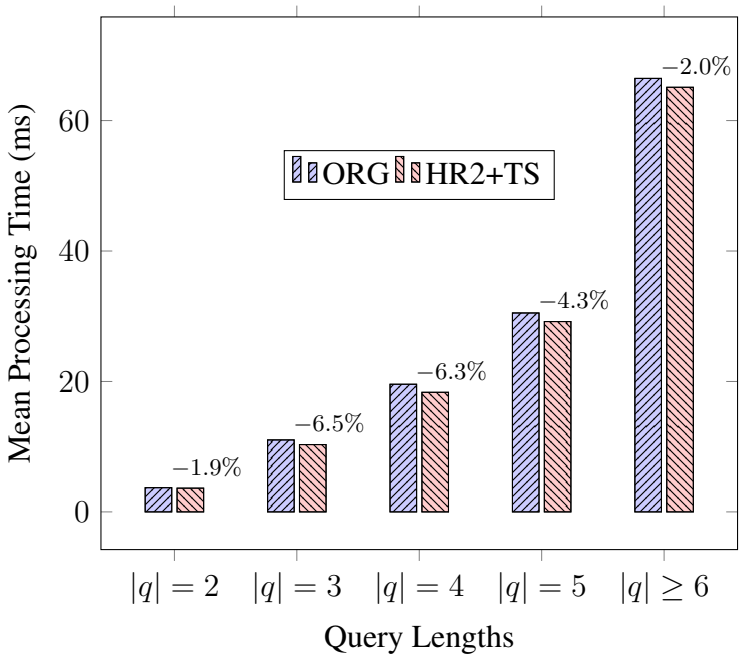


(a) $k = 10$

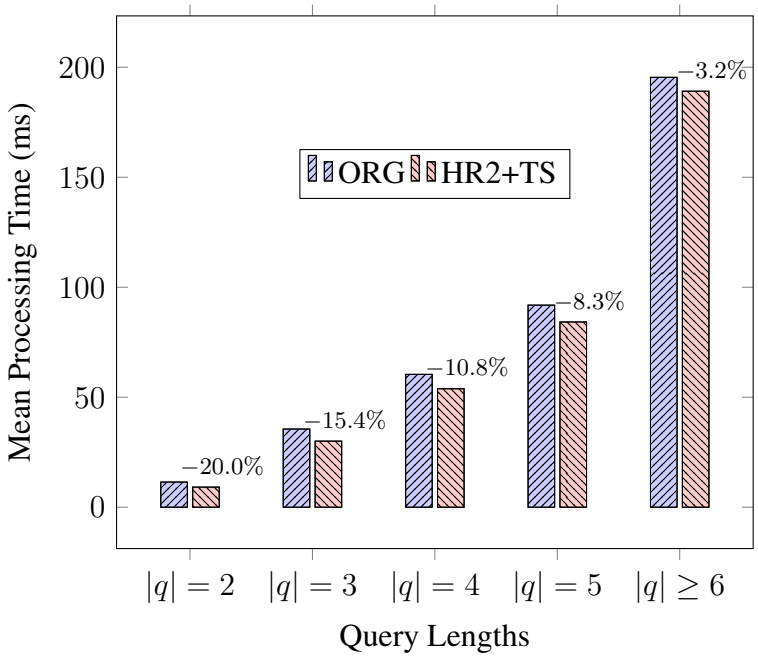


(b) $k = 1000$

Figure 3.3: Mean processing times by query length for WAND



(a) $k = 10$



(b) $k = 1000$

Figure 3.4: Mean processing times by query length for BMW

diminish and our heuristics might be sufficient on their own. We will explore if and when this case occurs in our future work.

In Figure 3.3 and 3.4, we provide a break-down of the mean execution times by query length for the original and best-performing HR2+TS approaches for WAND and BMW, respectively. We see that the performance gains are mostly obtained for the queries with 3 and 4 terms. For 2-term queries, the room for improvement is not much when the $k = 10$ since only single query terms can be exploited to estimate a threshold. However, when $k = 1000$ the query processing times are reduced up to 20% and 13% for BMW and WAND. For queries with more than 4 terms, we think that further gains are possible with heuristics that are more effective and/or faster in identifying useful subsets among many candidates. This is left as an exciting future direction.

Finally, Table 3.3 provides more insight on how our heuristics improve the processing time. For TS, HR2 and HR2+TS, we present how close the score threshold lower-bound is to the actual k·th score of a query, averaged over all test queries. For $k = 10$, TS can obtain a score that is equal to 66% of the correct score, while HR2 and HR2+TS reach 72% and 78%, respectively. The ratio increases for all three approaches for larger k, that is possibly because at lower ranks the retrieval scores might be smaller and/or more stabilized, yielding tighter lower-bounds. Another important observation from Table 3.3 is that there is still room for improvement, i.e., to tailor heuristics to obtain even tighter lower-bounds on the score threshold.

3.4.4 Storage space overhead

Assuming that a typical result (URL, title and snippet) takes around 256 bytes, for $k = 10$, the space required in a result cache would be 2560 bytes per query [3]. In contrary, in our score cache, a query string (with 3 terms on the average) and a single quantized score value would take around 25 bytes, i.e., around 1% of the space for the result cache (per index node), and seems like a negligible cost for storing even millions of entries in practice. Anyway, we investigated whether using the score cache's space to extend the result cache capacity would yield efficiency gains similar to ours, and found out that for the setup reported here, such an approach would allow

Table 3.3: Ratio of the score threshold lower-bound of a query (as found by different heuristics) to actual k·th score.

	$k = 10$	$k = 100$	$k = 1000$
TS	0.66	0.70	0.72
HR2	0.72	0.77	0.81
HR2+TS	0.78	0.81	0.83

storing an additional 25K queries (obtained from the training log) in the result cache. By running the test queries over this extended result cache (with original WAND or BMW) , we only observed up to 0.1% gain in mean processing time in comparison to ORG figures shown in Table 1. That is, extending the result cache is not as useful as the proposed score cache.

As our analysis above focuses on a single node, an astute reader may ask what happens if the total storage for the score caches in all nodes is allocated to a result cache? Obviously, this would yield a huge result cache, but there is a catch: Earlier works have consistently shown that approximately 44% of a web search query volume is made up of singleton queries (i.e., asked only once); and taking compulsory misses into account, hit-ratio is typically bounded by 50%, even for an infinitely large result cache [2]. That is, increasing result cache size won't help after this hit-ratio bound is reached; while our score cache can still improve performance for such singletons and compulsory misses. To simulate this, we assumed that each distinct test query, once processed, is inserted into the result cache, which is now assumed to be a dynamic cache with infinite size. Thus, our measurements on the test query stream includes processing time of only compulsory misses and singletons. We found that, the best-performing strategy HR2+TS with WAND (BMW) still improves TS by a relative 0.9% (1.1%), 1.7% (2.2%) and 2.4% (3.2%) for $k \in \{10, 100, 1000\}$, respectively, in terms of mean processing time. Note that, these savings in processing time are the exclusive benefits of the score cache (in return for the storage space used); and cannot be obtained by a result cache of even infinite size.

3.5 Conclusion

We show that using a score cache and appropriate heuristics to access the cache, we can compute score thresholds to improve the performance of dynamic pruning. Our gains in query processing times, although statistically significant, may seem numerically small. However, given that WAND and BMW are time-tested efficient processing strategies, they are not trivial to improve; and earlier optimizations applied on top of them also report actual gains in a similar range, i.e., a few milliseconds [32, 29] or even microseconds [32]. For search engines with a large query volume, such savings would be still useful, as they may add up and help improving other performance metrics (such as throughput) or services.

CHAPTER 4

CONCLUSIONS

In this thesis, we introduced advanced methods for caching that concentrated on improving the efficiency of retrieval systems, especially useful for that are on the web. Our contributions focus on different layers of search engine caching, and are beneficial in multiple aspects *e.g.* both space and time.

Firstly, we focused on the result cache layer that serves the HTML snippets upon query submission without any additional processing required. We presented both single and multi signal cache eviction policies in a realistic setup using different cache sizes and a large query log. Elaborating over the previous work, that is described in Chap. 2, we showed that keeping the entire query frequency history poses significant improvements for result cache hit ratio. We then revealed the costs of storing large data structures for the history, by modeling the memory requirements in which the model is based on our implementation that reflects real life scenarios. We discussed ways of reducing storage requirements using BF-like data structures as well as briefly presented the theory of Bloom Filters and the Tiny storage scheme. Our findings are important since they are applicable to well-known cache eviction policies and shown significant improvements in terms of space requirements (up to 78% metadata storage gains) without adversely effecting the cache performance. For result caching, we showed that our space savings are meaningful when the result cache is extended; there is still room for improvement in hit ratio. Lastly, we modeled query processing costs and suggested that the saved space can be further exploited with an additional doc-id cache, and showed that the improvements are concrete compared to that of using the extended result cache.

Next, in Chap. 3 we first presented basic information retrieval techniques such as the

inverted index and query processing by introducing algorithms of different traversal techniques in detail. We then described dynamic pruning strategies in which documents are skipped without computing a score for them by useful pointer advancement techniques and index layouts. We then proposed an alternative cache layer, namely the score cache for search engines that employ dynamic pruning strategies. Our additional cache stores the k-th score for each query that is already resident in the result cache. The scores of the subset queries are used as an initial threshold value upon result cache misses to allow skipping more documents for dynamic pruning *e.g.* WAND and BMW. Since generating the of the subsets would take significant time, we proposed various heuristics to subset generation. For experiments, we used AOL query log and TREC GOV index for evaluation. Our comprehensive experiments revealed that accommodating the score cache would yield significant time reductions for both WAND and BMW (up-to 8.6%) which are already optimized algorithms such that the path to the improvement is narrow. Our experiments are realistic in the sense that it improves processing times for queries that are already filtered through the result cache and also, offers improvements for compulsory cache misses.

4.1 Future Work

While our current work only experimented with a static score cache, it is possible to extend our idea to a dynamic score cache to obtain a higher subset found ratio. Such en extension can cause an interesting trade-off as dynamic cache eviction policies introduce an additional overhead such as determining the victim. Therefore, it is possible that the costs of employing the dynamic cache can easily surpass the benefit of having it. This deserves further investigation.

To test the robustness of the proposed methods in Chap. 3, our heuristics can be coupled with additional dynamic pruning algorithms such as the *max-score*.

As an another test, our ideas can be applied to the rank-unsafe dynamic pruning algorithms to see if there are any additional benefits. Moreover, our heuristics can be improved to obtain a rank-unsafe score threshold to further increase skipping of documents. In such a setup our gains can be more concrete.

Another future direction is the improvement or addition of more complex heuristics or even machine learned models can be used to generate subsets that are both likely to be a cache hit and also yield high threshold values.

REFERENCES

[1] R. Ozcan, I. S. Altingovde, B. B. Cambazoglu, F. P. Junqueira, and Özgür Ulusoy, "A five-level static cache architecture for web search engines," *Information Processing & Management*, vol. 48, no. 5, pp. 828–840, 2012.

[2] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *SIGIR 2007*, pp. 183–190, 2007.

[3] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, and Ö. Ulusoy, "Second chance: A hybrid approach for dynamic result caching and prefetching in search engines," *ACM TWEB*, vol. 8, no. 1, pp. 3:1–3:22, 2013.

[4] Q. Gan and T. Suel, "Improved techniques for result caching in web search engines," in *Proc. of WWW 2009*, pp. 431–440, 2009.

[5] G. Einziger, R. Friedman, and B. Manes, "Tinylfu: A highly efficient cache admission policy," *ACM Transactions on Storage (ToS)*, vol. 13, no. 4, p. 35, 2017.

[6] G. Einziger and R. Friedman, "Tinylfu: A highly efficient cache admission policy," in *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 146–153, 2014.

[7] 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 TOIS*, vol. 24, pp. 51–78, Jan. 2006.

[8] F. Silvestri, "Mining query logs: Turning search usage data into knowledge," *Foundations and Trends in Information Retrieval*, vol. 4, no. 1-2, pp. 1–174, 2010.

[9] S. Jonassen, B. B. Cambazoglu, and F. Silvestri, "Prefetching query results and its impact on search engines," in *Proc. of SIGIR 2012*, pp. 631–640, 2012.

[10] B. B. Cambazoglu and R. Baeza-Yates, "Scalability challenges in web search engines," in *Advanced topics in information retrieval*, pp. 27–50, Springer, 2011.

[11] S. Podlipnig and L. Böszörményi, "A survey of web cache replacement strategies," *ACM Comput. Surv.*, vol. 35, no. 4, pp. 374–398, 2003.

[12] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[13] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[14] J. Bruck, J. Gao, and A. Jiang, "Weighted bloom filter," in *2006 IEEE International Symposium on Information Theory*, pp. 2304–2308, IEEE, 2006.

[15] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He, "Bitfunnel: Revisiting signatures for search," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 605–614, ACM, 2017.

[16] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[17] C. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[18] X. Dimitropoulos, M. Stoecklin, P. Hurley, and A. Kind, "The eternal sunshine of the sketch data structure," *Computer Networks*, vol. 52, no. 17, pp. 3248–3257, 2008.

[19] G. Pass, A. Chowdhury, and C. Torgeson, "A picture of search," in *Proc. of InfoScale 2006*, p. 1, 2006.

[20] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 4, pp. 3–11, 2000.

[21] K. Shah, A. Mitra, and D. Matani, "An o (1) algorithm for implementing the lfu cache eviction scheme," *Technical report*, 2010.

[22] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. Witschel, "Admission policies for caches of search engine results," in *Proc. of SPIRE 2007*, pp. 74–85.

[23] R. Ozcan, I. S. Altingövde, and O. Ulusoy, "Cost-aware strategies for query result caching in web search engines," *ACM Transactions on the Web*, vol. 5, no. 2, pp. 9:1–9:25, 2011.

[24] Q. Wang, C. Dimopoulos, and T. Suel, "Fast first-phase candidate generation for cascading rankers," in *Proc. of SIGIR*, pp. 295–304, 2016.

[25] M. Jeon, S. Kim, S. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: taming tail latencies in web search," in *SIGIR*, pp. 253–262, 2014.

[26] S. Ding and T. Suel, "Faster top-k document retrieval using block-max indexes," in *Proc. of SIGIR*, pp. 993–1002, 2011.

[27] M. Crane, J. S. Culpepper, J. J. Lin, J. Mackenzie, and A. Trotman, "A comparison of document-at-a-time and score-at-a-time query evaluation," in *Proc. of WSDM*, pp. 201–210, 2017.

[28] X. Long and T. Suel, "Three-level caching for efficient query processing in large web search engines," in *WWW*, pp. 257–266, 2005.

[29] A. Kane and F. W. Tompa, "Split-lists and initial thresholds for wand-based search," in *SIGIR*, pp. 877–880, 2018.

[30] C. M. Daoud, E. S. de Moura, A. L. da Costa Carvalho, A. S. da Silva, D. F. de Oliveira, and C. Rossi, "Fast top-k preserving query processing using two-tier indexes," *Inf. Process. Manage.*, vol. 52, no. 5, pp. 855–872, 2016.

[31] M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck, "Accelerated query processing via similarity score prediction," in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 485–494, ACM, 2019.

[32] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien, "Evaluation strategies for top-k queries over memory-resident inverted indexes," *PVLDB*, vol. 4, no. 12, pp. 1213–1224, 2011.

APPENDIX A

ADDITIONAL INDEX TRAVERSAL STRATEGIES

A.1 Index Traversal Pseudocodes

Algorithm 4 Ranked-OR in a TaaT manner

function RANKED-OR($\mathcal{Q}, \mathcal{I}, \mathcal{D}, k$) ▷ Parameters query, index, document set and the number of doc's to retrieve

  $\mathcal{A}_d = 0, \forall d \in D$ ▷ Accumulator array for scores

  for all $t \in \mathcal{Q}$ **do**

    for all $d \in \mathcal{I}_t$ **do**

      $\psi \leftarrow$ WEIGHT(d)

      $\delta \leftarrow$ DOCID(d)

      $\mathcal{A}_\delta \leftarrow \mathcal{A}_\delta +$ SCORE$(\psi, \mathcal{I}, t)$

    end for

  end for

  $\mathcal{H} \leftarrow$ HEAP()

  for $i = 1$ to $|A|$ **do**

    HEAP-INSERT$(\mathcal{H}, i, \mathcal{A}, k)$

  end for

  return CREATE_ANSWERS$(\mathcal{H})$

end function

Algorithm 5 Ranked-AND in a TaaT manner

function RANKED-AND($\mathcal{Q}, \mathcal{I}, \mathcal{D}, k$) ▷ Parameters query, index, document set and the number of doc's to retrieve

$\mathcal{A}_d = 0, \forall d \in \mathcal{D}$ ▷ Accumulator array for scores

$\mathcal{B}_d = 1, \forall d \in \mathcal{D}$ ▷ Bit vector to mark disjoint postings

for all $t \in \mathcal{Q}$ **do**

$\lambda \leftarrow 1$ ▷ Last candidate doc-id for term t

for all $d \in \mathcal{I}_t$ **do**

$\delta \leftarrow$ DOCID(d) ▷ Current doc-id

for $i \leftarrow \lambda$ to $\delta - 1$ **do**

$\mathcal{B}_i \leftarrow 0$ ▷ Invalidate doc-id's between last candidate and current

end for

if $\mathcal{B}_\delta = 1$ **then** ▷ Score if previously not invalidated

$\psi \leftarrow$ WEIGHT(d)

$\mathcal{A}_\delta \leftarrow \mathcal{A}_\delta +$ SCORE($\psi, \mathcal{I}, t$)

end if

$\lambda \leftarrow \delta$ ▷ Update last candidate doc-id to current

end for

for $i \leftarrow \lambda + 1$ to $|\mathcal{B}|$ **do**

$\mathcal{B}_i \leftarrow 0$ ▷ Invalidate rest

end for

end for

$\mathcal{H} \leftarrow$ HEAP()

for $i \leftarrow 1$ to $|\mathcal{A}|$ **do**

if $\mathcal{B}_i = 1$ **then**

HEAP-INSERT($\mathcal{H}, i, \mathcal{A}_i, k$)

end if

end for

return CREATE_ANSWERS($\mathcal{H}$)

end function

Algorithm 6 Ranked-AND in a DaaT manner

function RANKED-AND($\mathcal{Q}, \mathcal{I}, k$) ▷ Parameters query, index, document set and the number of doc's to retrieve

  $\tau \leftarrow |\mathcal{Q}|$

  for $t = 1$ to τ **do**

    $\mathcal{L}_t \leftarrow \mathcal{I}_{\mathcal{Q}_t}$ ▷ Initialize postings lists

    $\mathcal{P}_t \leftarrow 1$ ▷ List pointers initialized to first elements

  end for

  $\mathcal{H} \leftarrow$ HEAP()

  while $\forall t \in [1, \tau], \mathcal{P}_t \leq |\mathcal{L}_t|$ **do** ▷ Loop until a posting list is exhausted

    $\delta \leftarrow$ DOCID$(\mathcal{L}_{1,\mathcal{P}_1})$

    $\alpha \leftarrow true$ ▷ Flag to check if all doc-ids are equal or not

    for $t \leftarrow 2$ to τ **do**

      if DOCID$(\mathcal{L}_{t,\mathcal{P}_t}) \neq \delta$ **then**

        $\alpha \leftarrow false$

        break ▷ Break when a doc-id is not equal to δ

      end if

    end for

    if α **then** ▷ If δ exists in all postings lists

      $\sigma \leftarrow 0$

      for $t \leftarrow 1$ to τ **do** ▷ Compute score for δ

        $\psi \leftarrow$ WEIGHT$(\mathcal{L}_{t,\mathcal{P}_t})$

        $\sigma \leftarrow \sigma +$ SCORE$(\psi, \mathcal{I}, \mathcal{Q}_t)$

        $\mathcal{P}_t \leftarrow \mathcal{P}_t + 1$

      end for

      HEAP-INSERT$(\mathcal{H}, \delta, \sigma, k)$

    else ▷ Advance pointers to the maximum doc-id in pointed by $\mathcal{P}$

      $\mu \leftarrow \max\{$DOCID$(\mathcal{L}_{t,\mathcal{P}_t}) \mid t \in [1, \tau]\}$

      for $t \leftarrow 1$ to τ **do**

        while DOCID$(\mathcal{L}_{t,\mathcal{P}_t}) < \mu$ **do**

          $\mathcal{P}_t \leftarrow \mathcal{P}_t + 1$

        end while

      end for

    end if

  end while

  return CREATE_ANSWERS$(\mathcal{H})$

end function