COST-AWARE RESULT CACHING STRATEGIES FOR META-SEARCH ENGINES

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

BY

EMRE BAKKAL

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

FEBRUARY 2015

Approval of the thesis:

COST-AWARE RESULT CACHING STRATEGIES FOR META-SEARCH ENGINES

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

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** ________________

Assist. Prof. Dr. İsmail Sengör Altıngövde
Supervisor, **Computer Engineering Department, METU** ________________

Examining Committee Members:

Prof. Dr. Ahmet Coşar
Computer Engineering Department, METU ________________

Assist. Prof. Dr. İsmail Sengör Altıngövde
Computer Engineering Department, METU ________________

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU ________________

Assoc. Prof. Dr. Pınar Karagöz
Computer Engineering Department, METU ________________

Assist. Prof. Dr. Engin Demir
Computer Engineering Department, UTAA ________________

Date: ________________

ABSTRACT

COST-AWARE RESULT CACHING STRATEGIES FOR META-SEARCH ENGINES

Bakkal, Emre
M.S., Department of Computer Engineering
Supervisor : Assist. Prof. Dr. İsmail Sengör Altıngövde

February 2015, 46 pages

Meta-search engines are tools that generate top-k search results of a query by combining local top-k search results retrieved from various data sources in parallel. A result cache that stores the results of the previously seen queries is a crucial component in a meta-search engine to improve the efficiency, scalability and availability of the system. Our goal in this thesis is to design and analyze different cost-aware and dynamic result caching strategies to be used in meta-search engines. To this end, as our first contribution, we utilize the well-known cost-aware eviction policies in the literature in three different caching approaches, namely, query-level, resource-level and entry-level caching; that arise naturally in the meta-search setup. Next, we propose a novel entry-level caching approach that is again cost-aware and fits well to the special embarrassingly-parallel nature of the meta-search scenario. The proposed approaches are evaluated using the cache miss-cost metric in a large-scale simulation setup where the impact of various parameters (such as the number of resources, cache size and query cost distribution) is also investigated. Our simulation results show that the highest performance is obtained by using the entry-level caching approaches; and furthermore, our newly proposed approach outperforms both the traditional baselines and other cost-aware competitors.

Keywords: Meta-Search Engines, Caching, Performance Evaluation, Simulation

ÖZ

META-ARAMA MOTORLARI İÇİN MALİYET TABANLI SONUÇ ÖNBELLEKLEME YÖNTEMLERİ

Bakkal, Emre
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Yrd. Doç. Dr. İsmail Sengör Altıngövde

Şubat 2015 , 46 sayfa

Meta-Arama motorları fakli veri kaynaklarndan paralel olarak gelen yerel ilk-k arama sonuçlarını birleştirerek ilk-k arama sonuçları üreten araçlardır. Daha önceden görülmüş sorguların sonuçlarını tutan bir sonuç önbelleği, etkinlik, ölçeklenebilirlik ve kullanılabilirlik açısından bir arama motoru sisteminin kritik bir parçasıdır. Bu tezdeki amacımız meta-arama motorlarında kullanılmak üzere farklı maliyet tabanlı ve dinamik önbellekleme yöntemleri tasarlamak ve analiz etmektir. Bu amaçla, ilk katkımız olarak, literatürde iyi bilinen maliyet tabanlı tahliye mekanizmalarını, meta-arama kurulumu altında doğal olarak oluşan, sorgu-seviyesi, kaynak-seviyesi ve girdi-seviyesi önbellekleme isimlerinde, üç faklı önbellekleme yaklaşımı içerisinde değerlendirdik. Ardından girdi-seviyesinde çalışan ve meta-arama senaryosunun özel paralel doğasıyla uyumlu yeni bir önbellekleme yaklaşımı önerdik. Önerilen yaklaşımlar geniş ölçekli bir simulasyon kurulumu yardımı ile önbellek ıskalama maliyeti açısından farklı parametrelerin (kaynak sayısı, önbellek boyutu ve sorgu maliyet dağılımı gibi) etkileri göz önünde blundurularak ölçülmüş ve değerlendirilmiştir. Simulasyon sonuçlarımız en yüksek performansın girdi-seviyesinde önbellekleme yaklaşımları kullanılarak elde edildiğini göstermektedir. Dahası, yeni öne sürdüğümüz yaklaşımımız tüm geleneksel ve diğer maliyet tabanlı rakiplerini performans bakımından geride bırakmıştır.

Anahtar Kelimeler: Meta-Arama Motorları, Önbellekleme, Performans Değerlendir-
mesi, Simulasyon

To my family...

ACKNOWLEDGMENTS

I would like to thank my supervisor Professor İsmail Sengör Altıngövde for his constant guidance and friendship, and his limitless patience and tolerance. I feel myself lucky for getting the chance to work with him for the last two years and it was a great honor. Our study together influenced my academic career in a very positive way. I believe without his motivation and constant help this work could not be able to reach its current level.

I would also like to thank all committee members for spending their time and effort to read and comment on my thesis. Their valuable comments and suggestions significantly improved the quality and content of this thesis.

I also feel that I owe a debt of gratitude to my colleagues from my professional career. Their understanding helped me all the way through.

Lastly, sincerest thanks to all of my family members for staying with me through good times and bad times.

This research project is jointly supported by Ministry of Science, Industry and Technology of Turkey and Huawei Telekomünikasyon Ltd. Co. under SAN-TEZ funding program with the project number 0441.STZ.2013-2.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AOL	AOL Incorporation
API	Application Program Interface
CPU	Central Processing Unit
CS	Cache Size (in number of queries)
EL	Entry-Level
FedWeb	Federated Web Search
GDS	Greedy Dual Size
GDSF	Greedy Dual Size Frequency
GPACS	Greedy Parallel-Aware Caching Strategy
GUI	Graphic User Interface
LCU	Least Costly Used
LFCU	Least Frequently and Costly Used
LFU	Least Frequently Used
LRU	Least Recently Used
MSE	Meta-Search Engine
OS	Operating System
QL	Query-Level
RL	Resource-Level
TREC	Text Retrieval Conference
WSE	Web Search Engine

CHAPTER 1

INTRODUCTION

Combining search results obtained from different heterogeneous sources is a well-known problem in the information retrieval and database literature with various different names such as meta-search, federated-search and data-aggregation (or, fusion) [31] . From a web-search point of view, a meta-search engine is a tool that works by means of forwarding submitted queries to component search systems (referred to as *resources* hereafter), collecting their local top-k results, and obtaining a global top-k result by merging these local results. Although we have witnessed the domination of market by general purpose web search engines (WSE) in the last decade, the idea of a meta-search engine can be still useful in some specialized domains, such as shopping, entertainment, healthcare and education. In such cases, the results of a query submitted to a meta-search tool can be either directly presented to end-user, or they can be used to enrich the content of a web service (e.g., presenting the latest social-media updates about a movie the user is watching), or they can be post-processed to be used as input to other applications (e.g., product information and price extraction for a price-comparison engine). For all these scenarios, re-usability of the results of the previously submitted queries becomes an important matter regarding the performance, scalability and availability (i.e., when some of the resources are temporarily unavailable) of the meta-search engine system. Re-using previous results can also reduce the financial costs of a meta-search engine if some resources charge a processing fee per query (such as some well-known general purpose search engines' APIs[1]). Therefore, it is crucial for a meta-search engine to include a result cache

[1] `http://www.programmableweb.com/news/yahoos-new-search-api-pricing-compared-to-google-bing/analysis/2014/11/24`

component to store the results of the previously seen queries.

The goal of this thesis is to design and analyze different cost-aware and dynamic result caching strategies to be used in meta-search engine systems. Different from the earlier works (e.g., [13, 17]), we assume that a meta-search engine cache stores all the local results (instead of storing only the global top-k results). This assumption allows maximum flexibility while exploiting the cached results (as it allows applying different algorithms for result merging and for refreshing the expired entries in the cache), but also leads to new research challenges as addressed in this thesis. Furthermore, to the best of our knowledge, none of the earlier works consider cost-aware caching strategies for meta-search engines, as it is a recent trend even for WSEs [25].

As our first contribution in this thesis, we introduce three different levels of eviction, namely, query-level, resource-level and entry-level, that arise naturally in the meta-search setup and indicate the granularity of the entries to be evicted from the cache when it is full. For each such eviction level, we utilize the well-known traditional and cost-aware eviction policies [25] to identify the actual entries that will be evicted, and hence, end up with several combinations presenting alternative caching approaches. Second, we propose a novel entry-level caching approach that is again cost-aware and exploits the special requirements of the meta-search scenario, i.e., the embarrassingly-parallel nature of processing a given query at each resource. The proposed approaches are evaluated using the cache miss-cost metric (as in [25]) in a simulation setup where the impact of several parameters (such as the number of resources, cache size and cost distributions) is also investigated. In these experiments, a large excerpt of 500,000 queries is used from AOL query log [26]. Simulation results show that our new approach is superior to baseline strategies.

The rest of the thesis is organized as follows. In Chapter 2, we review the related work on meta-search engines and caching strategies for both WSEs and meta-search engines. Chapter 3 presents the cost-aware caching policies proposed for meta-search. Chapter 4 is devoted for the simulation setup and evaluation results. Finally, Chapter 5 concludes the thesis and also points to directions for future research.

CHAPTER 2

RELATED WORK

In this chapter, we first review the basic principles for federated search, then focus on the earlier works on meta-search engines in Section 2.1. Next, the previous caching approaches for both general purpose Web search engines and meta-search engines are described, in Sections 2.2 and 2.3, respectively. Finally, in Section 2.4, the metrics to evaluate the performance of caching in the context of a search engine are discussed.

2.1 Federated Search and Meta-Search Engines

Federated Search means searching multiple data resources simultaneously by submitting queries to them and merging returned result to a single query result list. Three major challenges in federated search area are *resource selection problem*, *resource representation problem* and *result merging problem*. [31] describes the principles of federated search and provides a summary of related previous work about these problems.

A meta-search engine is basically a web search engine which uses other general purpose web search engines as data sources instead of crawling and indexing real data sources and using those indexes to generate search results. In other words a meta-search engine is the application of federated search techniques on the web search problem.

In one of the earlier works, Wu et al. focuses on resource representation and resource selection problems of a meta-search system [39], by proposing an integrated metadata representation of the contents of the resources and applying a ranking mechanism which checks the similarity values of most similar documents of each resource on top of it.

2.2 Caching in Web Search Engines

Caching is a widely employed mechanism in general purpose Web search engines to improve the system efficiency (i.e., by reducing the query response time) and scalability (i.e., by reducing the query processing load at the backend) [32]. It is possible to cache different types of data items at different layers of a typical search engine architecture. Furthermore, the caching strategies can be categorized as either static or dynamic [21]. In a static cache, items to be cached are determined periodically based on previous usage statistics (typically, query logs) and the cache content is kept intact until the next periodic update [4]. In contrast, dynamic caching strategies that operate on a cache employ an eviction mechanism to decide on the item to be removed when the cache is full; and they rely on the frequency and recency of the cached items, as typical in well-known Least Frequently Used (LFU) and Least Recently Used (LRU) policies (e.g., see [27]), respectively. In what follows, for each type of data that can be cached in a search engine, the static and/or dynamic strategies in the literature are reviewed.

2.2.1 Cache Types

During query processing in a search engine, it is necessary to generate (as an intermediate or final output) and/or fetch (from secondary storage) different types of data items. Following the practice in [22], these data types and strategies to cache them are categorized as follows:

Result and score caching: One of the earliest studies that explore caching techniques for WSEs is by Markatos[21], which shows that queries submitted to a WSE exhibit temporal locality and hence, caching the result pages for these queries would improve the performance. Following the practice in the literature, it is assumed in this thesis that a result page includes the title, URL and snippet of the top-ranked (i.e., typically top-10) results for a query. Lempel and Moran address the prefetching issue in the context of result a cache [18]. In [14], the authors construct a hybrid result cache by reserving some portion of the cache capacity for the static caching, and the rest for the dynamic caching. The latter work also proposes an adaptive prefetching algorithm. In a related study, Baeza-Yates et al. again partition the cache space into two parts, one of which aims to store the results of the queries that are more likely to repeat by using some admission heuristics [3]. In [24], Ozcan et al. propose using the temporal stability of the query frequency while deciding on the content of a static result cache. Cambazoglu and Altingovde show that the cache hit-rate drops when the cached results for a given query should differ due to regionalization; and propose to use prefetching as a remedy [9]. The impact of prefetching query results is further analyzed in [16].

A more recent trend in result caches is to also take into account the miss-costs; i.e., designing policies that aim to cache the queries that are more expensive to process when they are not found in the cache. A pioneering work in this direction is by Ozcan et al., where several static and dynamic cost-aware caching policies are described [25]. Some of these policies are discussed in detail in Chapter 3. Gan and Suel also consider query costs during caching in a hybrid caching setup [15]. The idea of using query processing costs in designing (and also, evaluating) caching policies also allows a space efficient alternative of storing query results. Instead of storing the URL, title and snippet for each result, it is only possible to store the internal document identifiers (as in [14]) along with the scores (hence, called as score caching in [22]). In [23]), a hybrid cache that stores results in either formats is proposed, along with an algorithm to transfer a cached result from one portion to another.

List and intersection caching: While our focus in this thesis is on caching query results, the issue of caching posting lists of the underlying index is also well explored in the literature. In one of the earliest studies in this direction [34], LRU is employed as the eviction policy in a dynamic caching setup. Similar approaches are also taken in [28, 5], where it is assumed that the search engine includes both a result cache and list cache. The latter setup is called as a two-level cache, as different cache types are likely to be stored at different layers of the search architecture (e.g., the result caches can be stored at the broker machines and list caches at the index-servers). In

[19], a three-level cache is proposed where, in addition to result and postings lists, the intersection of certain posting lists are also cached, to improve the query processing efficiency. More recently, Tolosa et al. analyze the performance of a static cache that includes both posting lists and their intersections [33].

Document and snippet caching: In [22], a five-level cache is proposed that also considers the caching of documents at the document servers, in addition to caching all other data types discussed above. Note that, it is necessary to access actual documents to generate query-biased snippets, and hence, caching frequently accessed documents can further improve the query processing efficiency. Marin et al. also propose a multi-level cache, but instead of caching documents, they construct a location cache that captures the location information for the documents [20]. It is also possible to cache the snippets, rather than the documents, as proposed in [12, 36, 35]. Finally, the impact of using solid state drives for constructing different types of caches in real-life search engines is analyzed in [37, 38].

2.2.2 Expiration and Invalidation in Result Caches

An important dimension of the caching problem for WSEs is to detect and refresh the cached items that are not fresh anymore. As the dropping hardware prices allow larger cache capacities, the importance of the latter problem increases [10]. Since detecting stale entries in list and/or document caches is relatively easy, the focus in the last few years is detecting such entries in search engine result caches.

To this end, the most practical strategy is to attach a time-to-live (TTL) to each cached query result. Cambazoglu et al. further exploit idle cycles of the backend processing system (i.e., when the backend has less load than it can support), and propose to refresh the results of the queries that are more frequently submitted and "older" in the cache [10]. Instead of assigning a fixed TTL to all cached results, Alici et al. [1] propose to adaptively compute a TTL value tailored for each query. As a complementary approach, Bortnikov et al. [8] marks a result in the cache as stale if the number of times it is presented to users exceeds a predefined threshold, which, in a sense, serves as a frequency-based TTL. Sazoglu et al. [30] further extends this idea and defines a click-based TTL; and also shows that combining these different TTL types serves better for detecting stale results in the cache.

An alternative line of research to aforementioned blind strategies is using an informed approach, which exploits signals obtained from the indexer component to detect the stale results. For instance, Blanco et al. suggest to forward all the updated documents (or their summaries) to an invalidator component that matches these documents to cached queries to detect stale results [7]. In contrast, Alici et al. utilize timestamps attached to posting lists, documents and cached results and detects stale result by policies that are based on comparing these timestamps [1]. Bai et al. also exploit timestamps in a slightly different manner for the same purpose [6]. Note that, even in these informed approaches, a TTL strategy is still applied to enforce an upper-bound on the validity of a cached result.

2.3 Caching in Meta-Search Engines

In contrast to the case of general purpose search engines, caching for meta-search engines is an area that is left unexplored; with the exception of a few works. In one of the earliest studies, Chidlovskii et al. propose a semantic cache at the client-side for a meta-search system [13]. In this cache, semantically relevant data is grouped together to be able to obtain some of the query results from the cache, even if the query result is not completely stored in the cache. Their work also mention the possibility designing eviction policies that take into account the cost of these semantic regions; however the experimental evaluation only considers typical LRU policy and hit-rate as the efficiency metric. Lee et al. describe a popularity-driven caching strategy for meta-search engines; but again do not consider the notion of cost during caching and evaluation [17].

2.4 Evaluating Cache Performance

Typical metrics for evaluating the performance of a cache is hit-rate or its complement, miss-rate. More recently, following the similar ideas in other contexts (e.g., see [11]), it is proposed to use miss-cost as a metric to evaluate the cache performance in search engine result caches [25, 15]. In [25], the cost of generating the query results is considered as the time cost of processing a query that caused a cache-miss. In this thesis, a similar approach is also taken while designing and evaluating caching approaches for meta-search engines. It is also possible to employ alternative cost metrics; for instance, Sazoglu et al. utilize the financial cost of processing a query (in terms of the electricity prices) [29].

CHAPTER 3

COST-AWARE RESULT CACHING IN META-SEARCH

In this chapter, firstly we will explain the meta-search scenario this thesis is based on in Section 3.1. Then, we will summarize some well-known baseline and cost-aware eviction policies in Section 3.2. After that, we will introduce three eviction-levels that naturally occur in meta-search scenario, and analyze their effects on the eviction policies in Section 3.3. And at last, we will propose a new eviction policy in Section 3.4.

3.1 Meta-Search Scenario

We consider a meta-search framework where a broker search system forwards the query to component search systems that may include general purpose search engines as well as APIs of Web 2.0 platforms, like YouTube or Twitter. The top-*k* result list, r_i, from each such resource s_i, is returned to the broker; and merged there to obtain the global query result. We assume that for a given query *q*, there is an associated cost C_i to obtain the result r_i, which is the elapsed time for query processing at the target resource plus the network transfer time. Remarkably, the same resource can yield different costs for different queries, and the same query can incur different costs from different resources. For a given query *q* and assuming there are *n* different resources s_1 to s_n, we store the vector $R_q : \langle (q, r_1, C_1), \ldots, (q, r_n, C_n) \rangle$ in the result cache. We believe it is preferable to store the local results from each resource rather than storing the merged (global) result (as in the earlier works like [13, 17]), as it allows more flexibility to switch to a different result merging algorithm and to apply separate

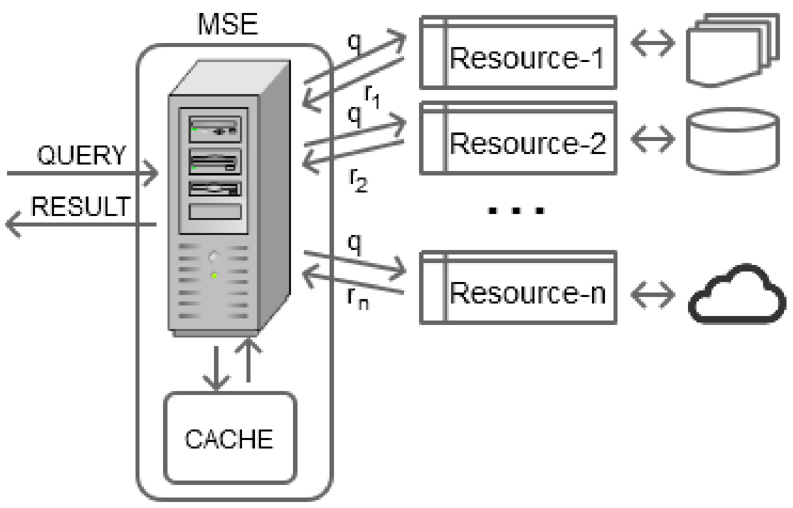


Figure 3.1: Meta-Search Engine Structure

mechanisms to invalidate cached results (e.g., results from more dynamic resources can be assigned a smaller time-to-live value [10]).

In this context, the cost of a query becomes the maximum cost of all the (q, r_i, C_i) triples in the R_q vector, since the query is processed in an embarrassingly-parallel manner (i.e., the meta-search engine sends the query to resources in parallel), and hence the overall latency would be the latency of the slowest response, such that:

$$C(q) = \max_{i=1}^{n} C_i, \ where \ R_q : \langle (q, r_1, C_1), \ldots, (q, r_n, C_n) \rangle \tag{3.1}$$

We consider only a dynamic cache scenario, since a reasonably large dynamic cache would anyway store the most valued entries that would be stored in a typical static cache. Markatos[21] also reports that static cache is better only when the cache size is quite limited.

In what follows, we first review the cost-aware eviction strategies for web search engines, and then discuss the unique features of meta-search setup that allows us to define caching approaches based on evicting entries at different granularities from the cache.

3.2 Eviction Policies

In this subsection, we will summarize well-known *LFU* and *LRU* policies as base-line eviction policies along with four different cost-aware cache eviction policies we have adopted from previous works in the literature that proposed them in the context of a general purpose WSE [25].

Note that, as we will discuss hereafter, a cache entry *e* can denote either the query result vector R_q, or a triple (q, r_i, C_i) in this vector, depending on the eviction-level. *C(e)* denotes the cost of generating this cache entry and *F(e)* denotes the frequency of this cache entry (i.e., submission frequency of the corresponding query in the past). *S(e)* denotes the cache entry size (in bytes). In this thesis, for the sake of simplicity, all cache entries' sizes considered as equals (i.e., $S(R_{q_x}) = S(R_{q_y})$ and/or $S((q_x, r_i, C_i)) = S((q_y, r_j, C_j))$). Lastly *s(R)* denotes the number of resources.

Least Recently Used (LRU) : This well-known eviction policy chooses the least recently used cache entry as the eviction victim. Note that, for a given query *q*, every cache entry (q, r_i, C_i) has the same recency.

Least Frequently Used (LFU) : In this well-known eviction policy each cache entry has a frequency value that increases with every submission of the corresponding query. The policy chooses the cache entry with the lowest frequency value as the eviction victim. The policy is called "in-cache LFU" in [27]. Note that, for a given query *q*, every cache entry (q, r_i, C_i) has the same frequency, and the frequency information of a query gets lost if all entries related to the query gets evicted.

Least Costly Used (LCU) : This most basic cost-aware eviction policy introduced in [25], evicts the cache entry with the lowest *C(e)* cost value.

Least Frequently and Costly Used (LFCU) : This cost-aware eviction policy again introduced in [25], evicts the cache item with the lowest *V(e)* value such that:

$$V(e) = C(e) \times F(e)^K, where\ K > 1 \tag{3.2}$$

The *K* value in the Equation 3.2 serves as a bias to emphasize the effect of frequency, since its observed in [15] and [25] that frequent queries tend to keep appearing frequently while the least frequent queries tend to totally fade away with time.

Greedy Dual Size (GDS) : This cost-aware eviction policy introduced in [11], evicts the cache item with the lowest *H(e)* value such that:

$$H(e) = \frac{C(e)}{S(e)} + L \tag{3.3}$$

The *L* value in the Equation 3.3 serves as an aging factor. It is initialized to 0 and every time an entry gets evicted the *L* value of the cache gets updated with the *H(e)* value of the evicted entry. When a cached query gets submitted again *H(e)* values of all entries related to the query gets re-calculated since *L* value of the cache could be changed during the time the entries are stored in the cache. Note that the *S(e)* value can be ignored in our case, since it's assumed that all result pages takes same amount of space as mentioned before.

Greedy Dual Size Frequency (GDSF) : This cost-aware eviction policy is the same policy utilized in [25] with the name *GDSF_K*, which is a modified version of GDS replacement policy of [2]. It evicts the cache item with the lowest *H(e)* value such that:

$$H(e) = \frac{C(e)}{S(e)} \times F(e)^K + L, where\ K > 1 \tag{3.4}$$

The *K* and *L* values of the Equation 3.4 serve the same purposes they served in Equations 3.2 and 3.3 (i.e., as a frequency emphasizing bias and as an aging factor), and work exactly the same way as in Equations 3.2 and 3.3. Note that, this strategy combines all three dimensions; namely, frequency, recency and cost of the cached entries, while deciding the entry to be evicted.

3.3 Eviction Levels

Our choice of storing the entire result vector $R_q : \langle(r_1, C_1), \ldots, (r_n, C_n)\rangle$ in the cache naturally and uniquely allows us to consider different granularities of eviction, as it is possible to evict the entire result vector of a query; or one or more entries from one or more result vectors. In this paper, we define three different caching approaches based on the eviction granularity (level). A good understanding of the differences between these eviction-levels and their effects on the explained cache eviction policies, will reveal our motivation behind the proposal of our novel *GPACS* policy.

The eviction levels do not change the calculation method of the cost of a cache miss, but they affect the list of evicted items (E) and thus the resulting miss cost. In meta-search scenario, the cost of a cache miss for a given query q would be the maximum cost of all the (q, r_i, C_i) triples which are not stored in the cache (i.e.missed), because of the embarrassingly-parallel nature of the meta-search scenario (i.e. in meta-search a query is processed at all resources in *parallel*; so the overall latency is the maximum of these individual costs.), such that:

$$C(q,\ miss) = \max_{e\ \notin\ Cache} C_i,\ where\, e : (q, r_i, C_i) \tag{3.5}$$

3.3.1 Query-Level (QL) Eviction

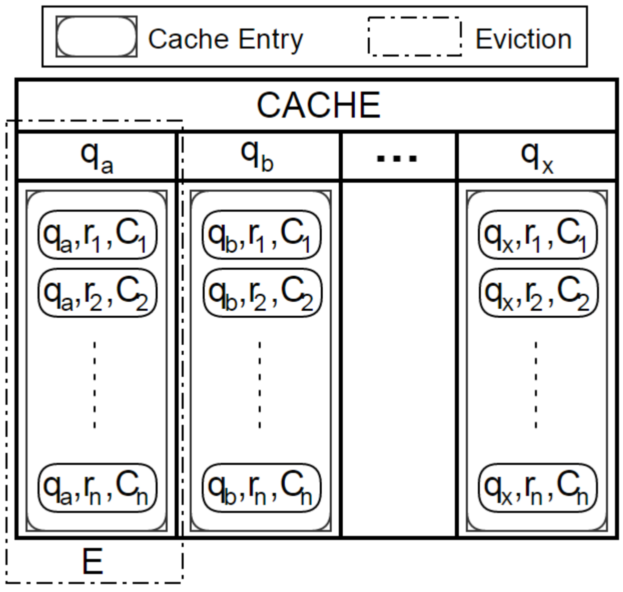


Figure 3.2: Query Level Eviction Cache Structure

In this eviction level a cache entry consists of all the (q, r_i, C_i) triples of R_q vector. Thus, when a query gets submitted either a full cache hit or a full cache miss would occur (i.e., no partial hits/misses). When the cache is full and a new query gets submitted, a query q gets selected as eviction victim and the whole result vector R_q gets evicted (Figure 3.2). We calculate the cost of this eviction as the maximum cost of all the evicted (q, r_i, C_i) triples, such that:

$$C(eviction) = \max_{i=1}^{n} C_i,\ where\, R_q : \langle (q, r_1, C_1), \ldots, (q, r_n, C_n) \rangle \tag{3.6}$$

Notice that the Equation 3.6 corresponds to the cost of the evicted query according to the Equation 3.1, and it equals to the miss cost that a future submission of the evicted query q would incur according to Equation 3.5.

3.3.2 Resource-Level (RL) Eviction

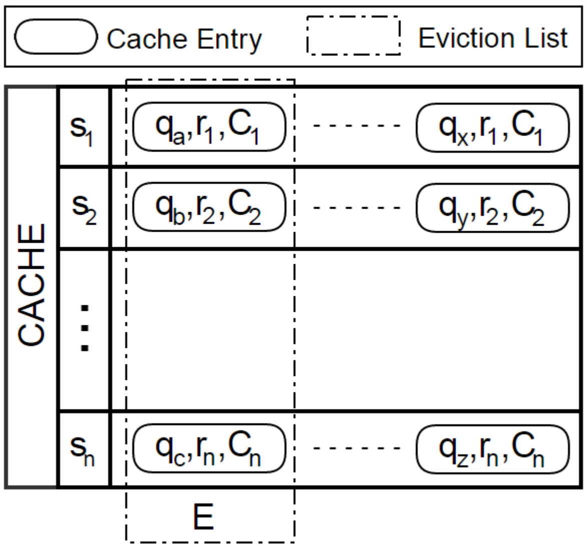


Figure 3.3: Resource Level Eviction Cache Structure

In this eviction level, the cache is uniformly partitioned for each resource s_i, and each partition acts as a separate local cache (Figure 3.3). When the cache is full, for a new query to be inserted each resource partition evicts one local (q_x, r_i, C_i) entry, and then the entries of the new query get inserted to the corresponding partitions (i.e. each entry to the partition of its own resource). For this case when a query gets submitted partial hits/misses may occur, since each partition evicts its least valuable item and these least valuable items may belong to different queries. In this context, we calculate the cost of an eviction as the sum of the maximum costs of all the evicted (q_x, r_i, C_i) entries for each unique query q_x occurred in the eviction list (*E*), such that:

$$C(eviction) = \sum_{for\ each\ unique\ q_x \in E} \max C_i \tag{3.7}$$

Notice that the Equation 3.7 corresponds to the sum of the miss costs that a future submission of the each unique query $q_x \in E$ would incur according to Equation 3.5.

3.3.3 Entry-Level (EL) Eviction

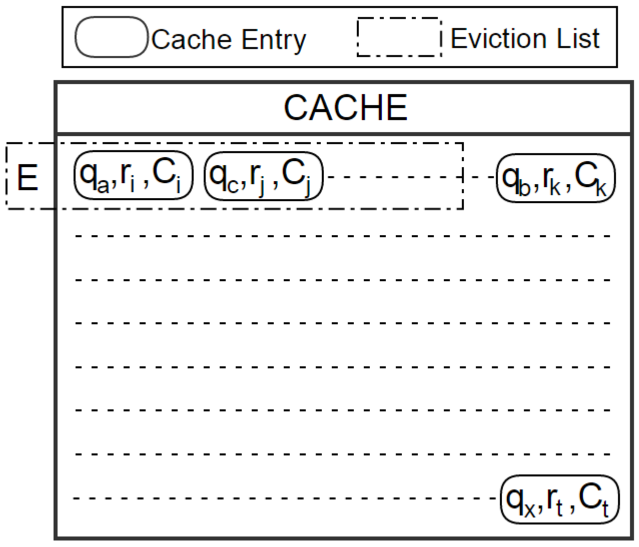


Figure 3.4: Entity Level Eviction Cache Structure

In this eviction level, each (q_x, r_i, C_i) triple is considered as an independent cache entry, resulting in the maximum granularity. Therefore when a need of eviction arises (i.e., a new R vector needs to be inserted and the cache is full), the least valuable *n* triples get evicted, regardless of the query or the resource they belong to (Figure 3.4). Similar to the resource-level eviction, in this case when a query gets submitted partial hits/misses may occur, since evictions occur at the lowest level and evicted items in a single eviction/insertion operation's eviction list *E* may belong to different queries, leaving partial results in the cache.

In this context, we calculate the cost of an eviction in exactly the same manner as we did for the resource-level eviction case (Equation 3.7). The difference between the two cases lies with the components of their eviction lists. While the resource-level eviction policy's eviction lists can have only one entry for each of the resources, entry-level eviction policy has more flexibility to evict less costly resources' results instead (i.e., it can choose to evict results from a faster resource and leave slower resources' results in the cache).

3.3.4 Effects of Eviction Levels on Eviction Policies

In order to demonstrate the difference between the eviction-levels, Query-Resource Cost Matrix in the Table 3.1 will be used as a representation of the cache content. In the matrix, each row (column) is a query (resource), respectively; and the entries denote the cost C_i of retrieving a query result from a particular resource s.

Table 3.1: Query-Resource Cost Matrix

	s_1	s_2	s_3	s_4
q_1	5	12	20	22
q_2	7	24	6	28
q_3	4	5	18	15

It is easy to see that *LRU* policy does not get affected by eviction-levels, it simply evicts the cache content related to the least recent query. *LFU* policy can get affected if there are more than one query with the same lowest frequency value, but even in a situation like that, it is more effective to evict cache content that belongs to only one of the queries. Thus, it can be said that *LFU* does not get affected by eviction-levels, neither.

Since all cost-aware policies evict the item that has the lowest value for some cost function, for the sake of simplicity and clarity assume that we apply *LCU* as the cost-aware eviction strategy.

When a query q_4 arrives, *LCU* policy evicts the following entries:

- Query-Level Eviction : $E = q_3$ with an overall cost of $max(4, 5, 18, 15) = 18$, as q_1 and q_2 have the costs 22 and 28 respectively. (See Equation 3.6)
- Resource-Level Eviction : $E = \langle (q_3, s_1),\ (q_3, s_2),\ (q_2, s_3),\ (q_3, s_4) \rangle$ with an overall cost of $max(4, 5, 15) + max(6) = 21$ (See Equation 3.7)
- Entry-Level Eviction : $E = \langle (q_3, s_1),\ (q_1, s_1),\ (q_3, s_2),\ (q_2, s_3) \rangle$ with an overall cost of $max(4, 5) + max(5) + max(6) = 16$ (See Equation 3.7)

3.4 Greedy Parallel-Aware Caching Strategy (GPACS)

As the astute reader will realize, while the caching approach with entry-level eviction has much lower miss-cost than its competitors, it is not optimal. Indeed, a better solution should take into account the observation that if two triples (q, r_i, C_i) and (q, r_j, C_j) are evicted for a query *q*; the miss-cost will not be $C_i + C_j$; but $max(C_i, C_j)$, as r_i and r_j will be retrieved from the resources s_i and s_j in parallel. For example, in Table 3.1, the optimal solution (i.e., with the least possible miss-cost) is evicting $\langle (q_3, s_2),\ (q_3, s_1),\ (q_2, s_1),\ (q_2, s_3) \rangle$, as the incurred miss-cost would be $max(4, 5) + max(7, 6) = 12$.

3.4.1 Optimal Solution for Entity-Level Eviction

The optimal solution for the entry-level eviction can be computed using dynamic programming, in a similar fashion to that of the well-known 0-1 Knapsack problem. Let's define the matrix *A* to store the cost of query *q* using *n* resources in ascending order of the costs. Then, in each step, we attempt to add a new query to the solution, and while doing so, we compute (and store in a table) the costs when we evict entries for *r* resources from this new query, where $0 \leq r \leq n$, and entries for $n - r$ resources from the earlier queries. Hence, the recursive formula for the dynamic programming solution is:

$$d(q, n) = \begin{cases} 0 & \text{if } n = 0, \\ \min\limits_{0 \leq r \leq N} (d(q-1, r) + A(q, n-r)) & \text{if } n > 0 \end{cases} \tag{3.8}$$

Obviously, even computing the optimal cost has the computational complexity $O(MN^2)$, where *M* and *N* denote the number of queries in the cache and number of resources, respectively. Based on these run time requirements, it is not affordable to use the optimal solution as a cache management approach in practical systems.

Moreover, after each query submission to the system, the cost table needs to be updated, regardless of whether a cache-hit or miss occurs. Because, in the former case, the costs need to be re-computed for strategies like *LFCU*, *GDS* and *GDSF* that takes into account the frequency and recency; and in case of a cache-miss, after an eviction, the costs need to be updated again due to the recurrence relationship in Equation 3.8. Clearly, it is unfeasible to employ the optimal solution as a cache management approach while applying entry-level eviction.

3.4.2 Greedy Solution for Entity-Level Eviction

As a remedy, we propose a novel greedy algorithm that also takes into account the aforementioned parallelism effect. In a nutshell, the algorithm works as follows: While storing the vector R_q for a query q, the entries (q, r_i, C_i) in the vector are sorted with respect to their C_i values. Next, for a pair that is at position p in the sorted list, its $C^{'}$ values are defined as C_i/p. Intuitively, this indicates that if we remove the entries up to and including position p; we will have size-p free space and the miss-cost for this query will be C_i; so the cost per space is C_i/p. Once these $C_i^{'}$ values are computed, any of the eviction strategies described before can be employed; but taking into account the $C^{'}$ values in the computations. Finally, when an entry (q, r_p, C_p) at position p is removed (along with all the entries at positions $p^{'} < p$) ; we reduce the cost of all entries at higher positions by C_p; since the C_p miss-cost for this query will be anyway incurred from this point on. The $C^{'}$ values are also recomputed with respect to modified cost values. The new algorithm is called Greedy Parallel-Aware Caching Strategy (GPACS).

Example In order to illustrate how *GPACS* operate and demonstrate its benefits, Query-Resource Cost Matrix in the Table 3.1 will be used as a representation of the cache content. Note that, for the sake of clarity, we will again use *LCU*, and disregard the frequency and aging factors in this example. According to Query-Resource Cost Matrix in the Table 3.1, sorted R_q vectors would be as in Table 3.2, and $C^{'}$ matrix with the position information would be as in Table 3.3. Note that, in practice, these value can be stored along with the actual entries; the matrix view is just for the illustration purposes.

Table 3.2: Sorted R_q Vectors

R_{q1} :	$\langle(r_1, 5),\ (r_2, 12),\ (r_3, 20),\ (r_4, 22)\rangle$
R_{q2} :	$\langle(r_3, 6),\ (r_1, 7),\ (r_2, 24),\ (r_4, 28)\rangle$
R_{q3} :	$\langle(r_1, 4),\ (r_2, 5),\ (r_4, 15),\ (r_3, 18)\rangle$

Table 3.3: $C^{'}$ Matrix

	p_1	p_2	p_3	p_4
q_1	$5/1$	$12/2$	$20/3$	$22/4$
q_2	$6/1$	$7/2$	$24/3$	$28/4$
q_3	$4/1$	$5/2$	$15/3$	$18/4$

In this case, *GPACS* policy would mark (q_3, p_2) for eviction and then evict (q_3, p_1) along with it, obtaining 2 empty spaces. After the eviction it would update the $C^{'}$ and p values of (q_3, p_3) and (q_3, p_4). The new $C^{'}$ matrix would be as in Table 3.4.

Table 3.4: $C^{'}$ After First Eviction

	p_1	p_2	p_3	p_4
q_1	$5/1$	$12/2$	$20/3$	$22/4$
q_2	$6/1$	$7/2$	$24/3$	$28/4$
q_3	$(15-5)/1$	$(18-5)/2$		

Then, *GPACS* policy would mark (q_2, p_2) for eviction and then evict (q_2, p_1) along with it, obtaining 2 more empty spaces and reaching the required 4 spaces to insert the new query results. After the eviction it would update the $C^{'}$ and p values of (q_2, p_3) and (q_2, p_4). The new $C^{'}$ matrix would be as in Table 3.5.

Table 3.5: $C^{'}$ After Second Eviction

	p_1	p_2	p_3	p_4
q_1	$5/1$	$12/2$	$20/3$	$22/4$
q_2	$(24-7)/1$	$(28-7)/2$		
q_3	$(15-5)/1$	$(18-5)/2$		

It is now pretty straight-forward to see that eviction list (*E*) includes
$\langle(q_3, r_2),\ (q_3, r_1),\ (q_2, r_1),\ (q_2, r_3)\rangle$, with final cost of $max(4, 5) + max(7, 6) = 12$.

As it can be observed, *GPACS* policy has the lowest overall cost between all the combinations of policies and eviction-levels. For this toy example it even managed to find the optimal solution.

A minor problem that can occur during the eviction is to evict more items than necessary. In our example two entries with same position value of 2 were marked. But what if one of them were to have a position value of 3? In this case *GPACS* may act according to one of two options. It either searches for an entry with a position value of 2, or it can evict five items. In our simulations, the former and latter versions are referred to as *GPACSe1* and *GPACSe2*, respectively.

The pseudo-code representation of GPACS's addition and eviction policies are given in Table 3.6. As shown in the pseudo-code, in practice, a cache entry of GPACS needs to know its previous and next items in the sorted R_q vector, in order to evict the previous ones and update the following ones. Notice that, in reality even though we can use the $C^{'}$ values of the *GPACS* policy instead of the C values of any given cost-aware eviction policy, the pseudo-code shows the application of *GPACS*'s mechanism over *GDSF*'s eviction value function (Equation 3.4). In Section 4.2.2 of the next chapter (4), we will also be presenting only the results for the application of *GPACS* over *GDSF*, since *GDSF* is the most efficient of all the other eviction policies.

Finally note that, there exists a trade-off between the query costs and cache complexity (therefore, cache retrieval costs) in the entry-level eviction policies. As the number of resources increases, complexities and retrieval costs of the caching approaches also increase. However as it will be shown in the Chapter 4, cache miss costs reduce when an entry-level eviction policy is used instead of one of its query- or resource-level counterparts.

Table 3.6: GPACS Pseudo-Code

```
Algorithm GPACS
Input: (C,q,$R_q$), where C is cache, q is a query and $R_q$
is a vector of cache entries consisting of local top-k
result pages of q for each resource.
 1.  Function Addition()
 2.    sort $R_q$ in ascending order according to costs of
         each result page
 3.    assign nextItem prevItem and index values of
         cache entries according to their new order
 4.    reqiredSize = 0
 5.    newFreq = 0
 6.    foreach cache entry ce in $R_q$
 7.    | if ce is in C
 8.    | | newFreq = cached ce's frequency + 1
 9.    | | remove ce from C
10.    | else
11.    | | requiredSize = requiredSize + 1
12.    foreach cache entry ce in $R_q$
13.    | ce.freq = newFreq
14.    Eviction1(reqiredSize) //for GPACSe1
       or
       Eviction2(reqiredSize) //for GPACSe2
15.    foreach cache entry ce in $R_q$
16.    | ce.L = C.L
17.    | ce.H = (ce.cost/ce.index)×(ce.freq^k) + ce.L
18.    | add ce to C
19.  End function
20.
```

Table 3.6 (cont.): GPACS Pseudo-Code

```
Function Eviction1(reqiredSize) //for GPACSe1
  evicted = lowest H valued cache entry with an
        index lower than reqiredSize
  remove evicted from C
  requiredSize = requiredSize - 1
  C.L = evicted.H
  for all prevItems of evicted
  | remove prevItem from C
  | requiredSize = requiredSize - 1
  for all nextItems of evicted
  | reduce cost by evicted.cost
  | reassign indexes starting from first nextItem
  | recalculate H values
  if(requiredSize > 0)
  | Eviction1(requiredSize)
End function

Function Eviction2(reqiredSize) //for GPACSe2
  evicted = lowest H valued cache entry
  remove evicted from C
  requiredSize = requiredSize - 1
  C.L = evicted.H
  for all prevItems of evicted
  | remove prevItem from C
  | requiredSize = requiredSize - 1
  for all nextItems of evicted
  | reduce cost by evicted.cost
  | reassign indexes starting from first nextItem
  | recalculate H values
  if(requiredSize > 0)
  | Eviction2(requiredSize)
End function
```

CHAPTER 4

EXPERIMENTS

In Section 4.1 of this chapter, first we describe our simulation setup by explaining the dataset we used in the simulations along with the reasons why it is chosen, and then by describing the scenario that is simulated along with the different parameters used in the simulation. The java application we implemented is also explained along with its inputs in Section 4.1. Next, in Section 4.2, we explain our evaluation metrics and present the important results of the simulations, along with a discussion of the effects of the eviction policies, eviction levels and simulation parameters on the performance of the meta-search scenario.

4.1 Simulation Setup

4.1.1 Dataset

To the best of our knowledge, there is no public dataset to evaluate the caching performance of a meta-search engine, as this task requires a large number of queries. For instance, datasets such as those provided by TREC FedWeb Track are too small (i.e., up to 1000 queries) to be of use in cache performance evaluation for all practical purposes. Since constructing such a dataset is not practical, a simulation is used in this thesis. In order to have the simulation as close to real-life situation as possible, we used an excerpt of AOL query log [26] dataset of 500,000 (306,444 unique) queries that are sorted in time-stamp order with the aim of simulating the repetition pattern of the queries (See Table 4.1 illustrating a sample of the log).

Table 4.1: Example queries from AOL dataset

	time	query
q_1	2006-03-01_00:01:08	guardianship
q_2	2006-03-01_00:01:09	tropical rain forest
q_3	2006-03-01_00:01:09	make a astronaut costume from items at home
q_4	2006-03-01_00:01:09	cat

4.1.2 Scenario

We construct a simulation scenario as follows. Using the AOL query log [26] excerpt, for each query, we assume results from N different resources are retrieved and cached. For each such resource, we define an interval that represents the cost of retrieving results from this resource. Then, for each query q and resource r, we sample a cost value C uniformly at random from the interval associated with this resource. We store these (r_i, C_i) values generated per query so that all caching strategies are evaluated under the same conditions.

4.1.3 Parameters

In our simulations, we used 500,000 queries from AOL log [26] in timestamp order. First 300,000 queries are used to warm-up the cache, and the rest are used for evaluation. We assume $N \in \{10, 50\}$, and all cost values are in $[0, 1]$ range. We set the cost intervals for these resources in an ad hoc manner to model the fast, medium-speed and slow resources that exist in a real-life setting (e.g., fast and slow resources have the cost ranges $[0, 0.3]$ and $[0.8, 1]$, respectively).

Different cost boundary variances and distributions between resources has been analyzed in the experimentation phase of this thesis. Yet it is observed that even though different variations (e.g., total random distribution between 0-1 values for all resources; a uniform distribution of slow, medium-speed and fast resources; a large number of fast resources and small number of slower ones; and vice versa) affect the cost reduction performances of the policies, the performance-wise order of the policies does not get affected by these parameter settings. We observed that, only the efficiency difference between the *GPACS* algorithm and its closest competitor policies gets affected by these settings. The difference decreases if we use total random distribution or a uniform distribution of slow, medium-speed and fast resources; and it increases if there is a non-uniform distribution (i.e., a gap between average costs of resources). The observation is in line with our expectations considering that having a non-uniform distribution increases the importance of greedy costs of the cache entries for the *GPACS* policies.

Note that, in practice as different resources would have different retrieval costs in a real-life setup, a meta-search engine can simply record the retrieval time for each query and resource, to apply the proposed caching approaches. Therefore simulation results of a distribution which is initialized in ad hoc manner, to model fast, medium-speed and slow resources will be reported in this thesis.

We report simulation results for three different cache sizes, namely, caches that can store 10K, 50K and 100K query results for $N = 10$. We assume the same cache capacity when *N* is set to 50 (for the sake of fair comparison), hence the number of queries that can be cached drops accordingly, as 2K, 10K and 20K.

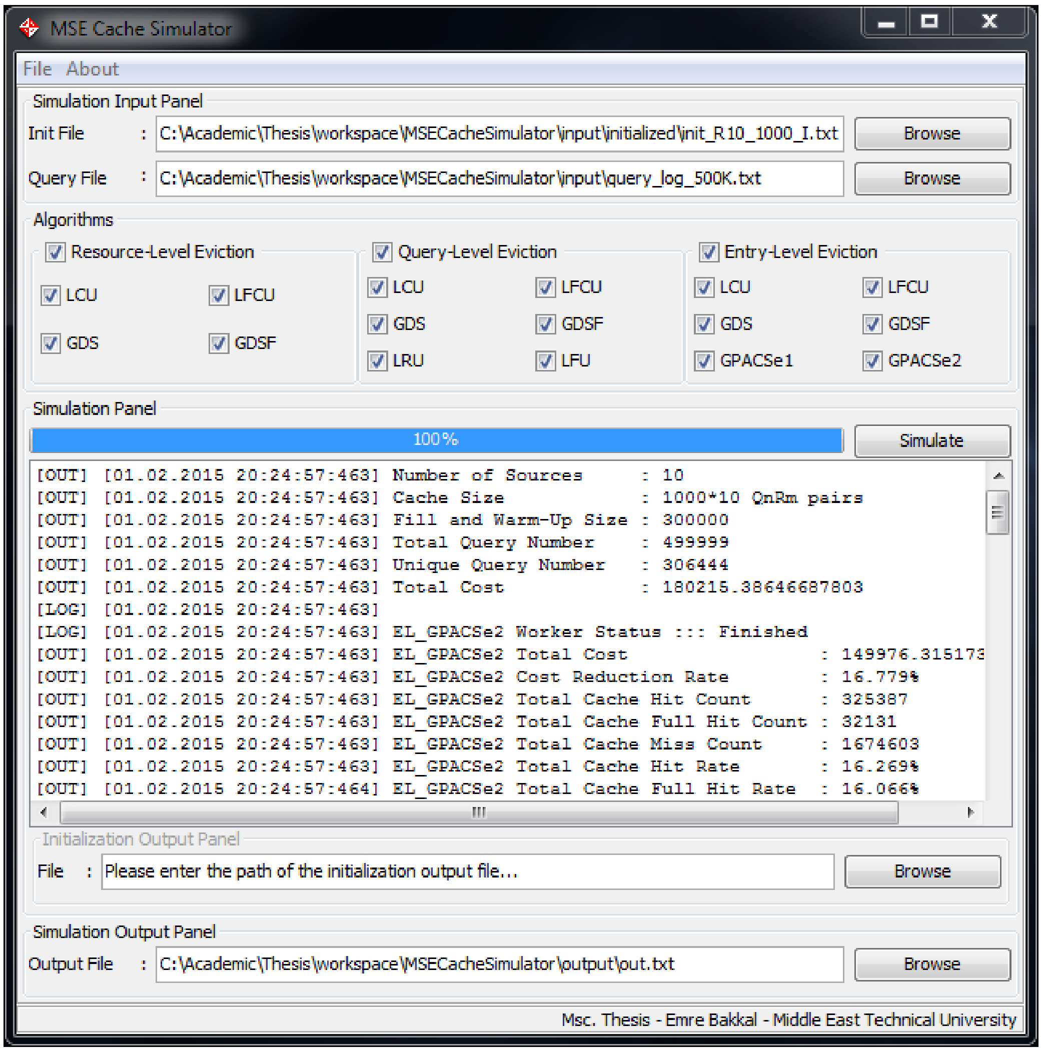
4.1.4 Application

A java application is implemented with the GUI shown in Figure 4.1 in order to simulate the caching approaches under different parameter settings. *LRU* and *LFU* policies are positioned under only query-level eviction case, since they are not affected by the eviction level (as mentioned before) and hence the results will be the same for all eviction levels.

The application takes an initialization file and the AOL query log file as inputs and simulates the selected policies. An initialization file may have one of two possible formats, namely *randomized* and *pre-initialized.*

In both cases the first line of the file gives the *cache size* and *warm-up query count* parameters as inputs.

Cache size (CS) is given in terms of the number of queries, so the actual cache capac-



MSE Cache Simulator

File About

Simulation Input Panel

Init File : C:\Academic\Thesis\workspace\MSECacheSimulator\input\initialized\init_R10_1000_I.txt

Query File : C:\Academic\Thesis\workspace\MSECacheSimulator\input\query_log_500K.txt

Algorithms

- [✓] Resource-Level Eviction: [✓] LCU, [✓] LFCU, [✓] GDS, [✓] GDSF
- [✓] Query-Level Eviction: [✓] LCU, [✓] LFCU, [✓] GDS, [✓] GDSF, [✓] LRU, [✓] LFU
- [✓] Entry-Level Eviction: [✓] LCU, [✓] LFCU, [✓] GDS, [✓] GDSF, [✓] GPACSe1, [✓] GPACSe2

Simulation Panel

100%

```
[OUT] [01.02.2015 20:24:57:463] Number of Sources      : 10
[OUT] [01.02.2015 20:24:57:463] Cache Size             : 1000*10 QnRm pairs
[OUT] [01.02.2015 20:24:57:463] Fill and Warm-Up Size : 300000
[OUT] [01.02.2015 20:24:57:463] Total Query Number     : 499999
[OUT] [01.02.2015 20:24:57:463] Unique Query Number    : 306444
[OUT] [01.02.2015 20:24:57:463] Total Cost             : 180215.38646687803
[LOG] [01.02.2015 20:24:57:463]
[LOG] [01.02.2015 20:24:57:463] EL_GPACSe2 Worker Status ::: Finished
[OUT] [01.02.2015 20:24:57:463] EL_GPACSe2 Total Cost                 : 149976.315173
[OUT] [01.02.2015 20:24:57:463] EL_GPACSe2 Cost Reduction Rate        : 16.779%
[OUT] [01.02.2015 20:24:57:463] EL_GPACSe2 Total Cache Hit Count      : 325387
[OUT] [01.02.2015 20:24:57:463] EL_GPACSe2 Total Cache Full Hit Count : 32131
[OUT] [01.02.2015 20:24:57:463] EL_GPACSe2 Total Cache Miss Count     : 1674603
[OUT] [01.02.2015 20:24:57:463] EL_GPACSe2 Total Cache Hit Rate       : 16.269%
[OUT] [01.02.2015 20:24:57:464] EL_GPACSe2 Total Cache Full Hit Rate  : 16.066%
```

Initialization Output Panel

File : Please enter the path of the initialization output file...

Simulation Output Panel

Output File : C:\Academic\Thesis\workspace\MSECacheSimulator\output\out.txt

Msc. Thesis - Emre Bakkal - Middle East Technical University

Figure 4.1: Java Application MSECacheSimulator GUI

ity would be number of resources times given cache size. For example if $s(R) = 50$ and $CS = 20K$, then the actual cache capacity would be $50 \times 20K = 1M$ local top-k results. Similarly if $s(R) = 10$ and $CS = 100K$, then actual cache capacity would be $10 \times 100K = 1M$ local top-k results.

Warm-up query count means the number of queries that will be processed to fill and warm-up the cache before starting the actual evaluation. In all of the simulations, 300,000 queries are used for the warm-up.

The following lines changes, depending on the format of the initialization file. If the initialization file is a randomized one (Table 4.2), only one line follows the first one, with the content of lower/upper cost boundary pairs for each resource. If the initialization file is a pre-initialized one (Table 4.3), one line per unique query follows the first one, with the content of query string and cost of the query at each resource. Note that, the cost values are normalized to $[0, 1]$ range.

Table 4.2: Example Randomized Initialization File

	Line Text
1.	1000 300000
2.	0.1 0.3 0.15 0.45 0.2 0.4 0.3 0.5 0.5 0.7

In the randomized case, the simulator generates random costs in the given range for each unique query and resource pair at the initialization phase of the simulation and uses these values throughout the simulations.

Table 4.3: Example Pre-initialized Initialization File

	Line Text
1.	1000 300000
2.	andy anderson motivational speaker == 0.110 0.299 0.249 0.346 0.624
3.	egg donation sacramento == 0.145 0.287 0.217 0.427 0.562

In the pre-initialized case, the simulator reads costs for each unique query and resource pair, at the initialization phase of the simulation and uses these values throughout the simulations. We have used pre-initialized inputs while collecting our simulation results (Section 4.2.2) for the sake of fair comparison of different caching approaches.

4.2 Simulation Results

4.2.1 Evaluation metrics

Since our goal by employing a cache is to reduce the total time cost for the metasearch system, we evaluate the proposed strategies in terms of the cost reduction percentage they achieve with respect to the no-caching case; i.e., when all results have to be retrieved for all 200,000 test queries from the scratch. As discussed before, for a given query q, if only some entries of R_q are located in the cache, the miss-cost incurred for this query is the maximum cost among the r_i's that caused a cache-miss.

In order to see the effects of *GPACS* on hit/miss rates, we use full and partial hit/miss numbers as a metric. As mentioned before, for a given query q, if all the entries of R_q are located in the cache, we consider a new submission of q as a full cache hit; and if only some x entries of R_q are located in the cache, we consider a new submission of q as x cache hits and $N - x$ cache misses. We calculate the *full-hit rate* and the *hit rate* values accordingly.

4.2.2 Results

As expected, using cache mechanisms significantly increases the performance (i.e., reduce the miss-costs) of meta-search engines. Furthermore, increasing cache sizes result in better caching performances.

Figures 4.2, 4.3 and 4.4 shows the respective performance of query-, resource- and entity-level caching approaches coupled with six different eviction strategies (namely, traditional *LRU* and *LFU* strategies as well as the cost-aware strategies *LCU*, *LFCU*, *GDS*, and *GDSF*) and for three different cache sizes (10K, 50K and 100K), when the number of resources is 10.

The three figures (4.2, 4.3, 4.4) reveal that traditional strategies *LRU* and *LFU* are inferior to cost-aware strategies, especially to *GDS* and *GDSF*, in all cases. Having said that, using only cost dimension is ineffective (as *LCU* is the worst performer in all three figures) as an eviction strategy, while the GDSF strategy that takes into account all available clues (i.e., the cost, frequency and recency of queries) almost always yields the highest reduction in miss-costs. These findings are in line with the previous results for web search engines [25].

If we compare the performances of different eviction levels, the three figures (4.2, 4.3, 4.4) show that, in general, query-level eviction is better than resource-level eviction; and entry-level eviction outperforms both of the latter. For instance, the reduction in miss-costs is 37.6%, 37.0%, 38.2% for QL, RL and EL eviction approaches with *GDSF* strategy (for the cache of size 100K), respectively. This indicates that EL

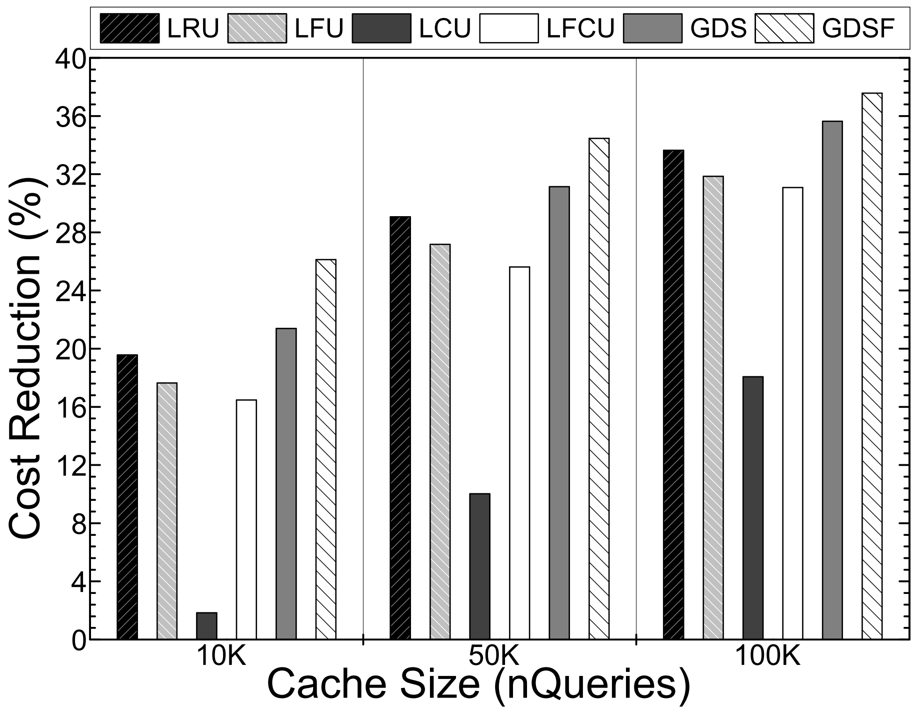


Figure 4.2: Query-Level Eviction, s(R) = 10

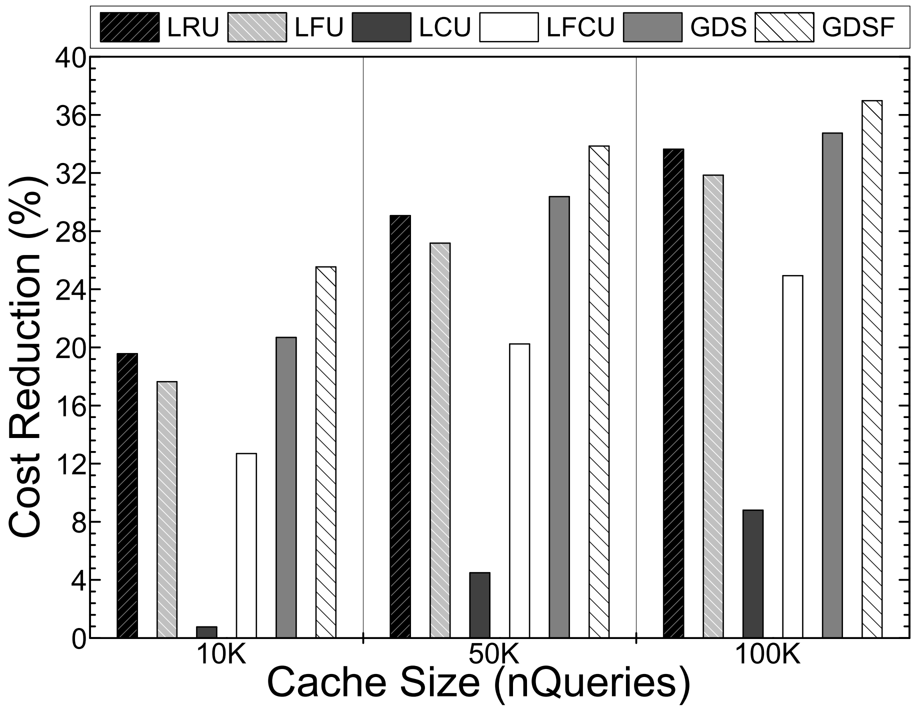


Figure 4.3: Resource-Level Eviction, s(R) = 10

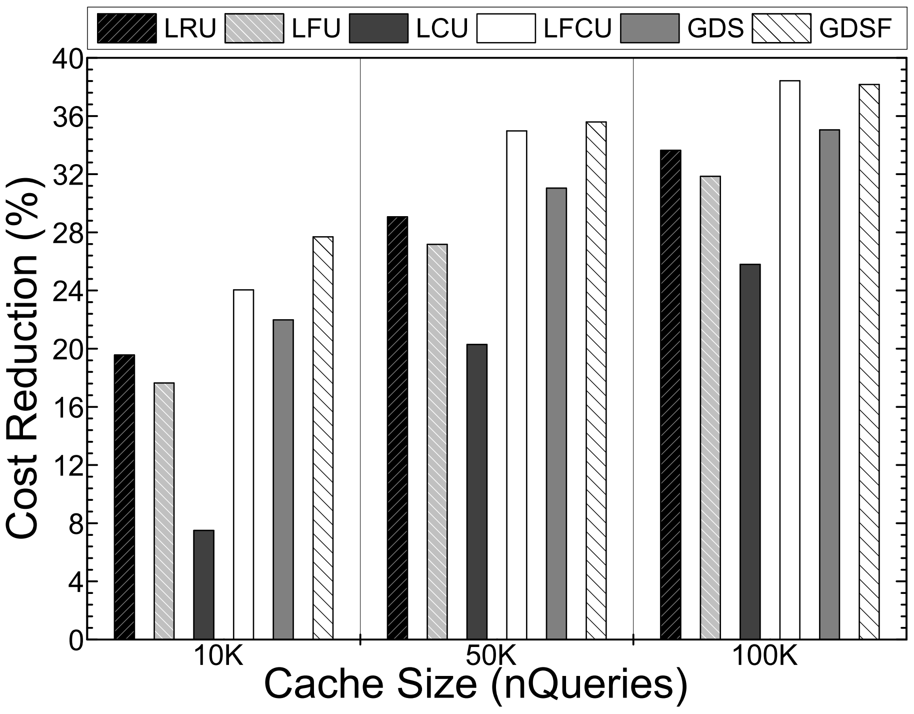


Figure 4.4: Entity-Level Eviction, s(R) = 10

eviction is the most effective approach in the meta-search scenario.

If we inspect the effects of eviction levels on caching policies, figures show that *LRU* and *LFU* policies does not get affected by eviction levels as mentioned before. Similarly *GDS* and *GDSF* get relatively less affected (up to 3% performance difference) by the eviction level. In contrast, the performance of *LCU* and *LFCU* strategies changes significantly depending on the eviction level (See Appendix A for a detailed comparison).

Figures 4.5, 4.6 and 4.7 show the respective performances for the same eviction level and eviction strategy combinations, when the number of resources is 50.

A comparison of the plots for 10 versus 50 resources shows that, absolute gains slightly drop as the same cache capacity can store a smaller number of full query result vectors for the 50 resources case. Notice that the cache sizes used in 10 and 50 resources cases are in fact pairs representing the same cache capacities (i.e., 100K, 500K and 1M local top-k results). All the trends observed before for the 10 resources case also hold for the 50 resources case, implying the robustness of our findings.

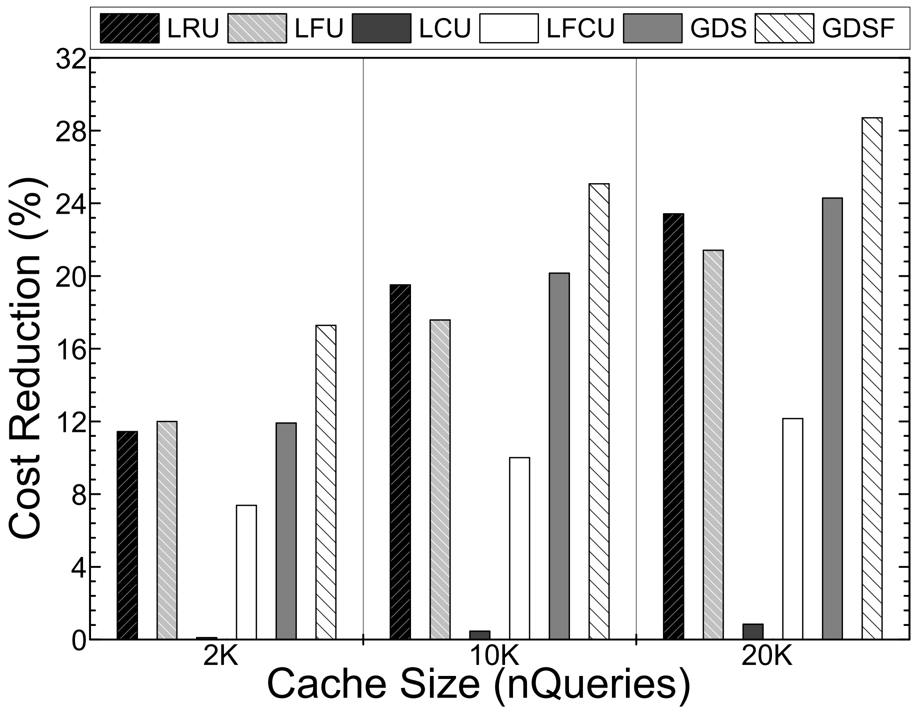


Figure 4.5: Query-Level Eviction, s(R) = 50

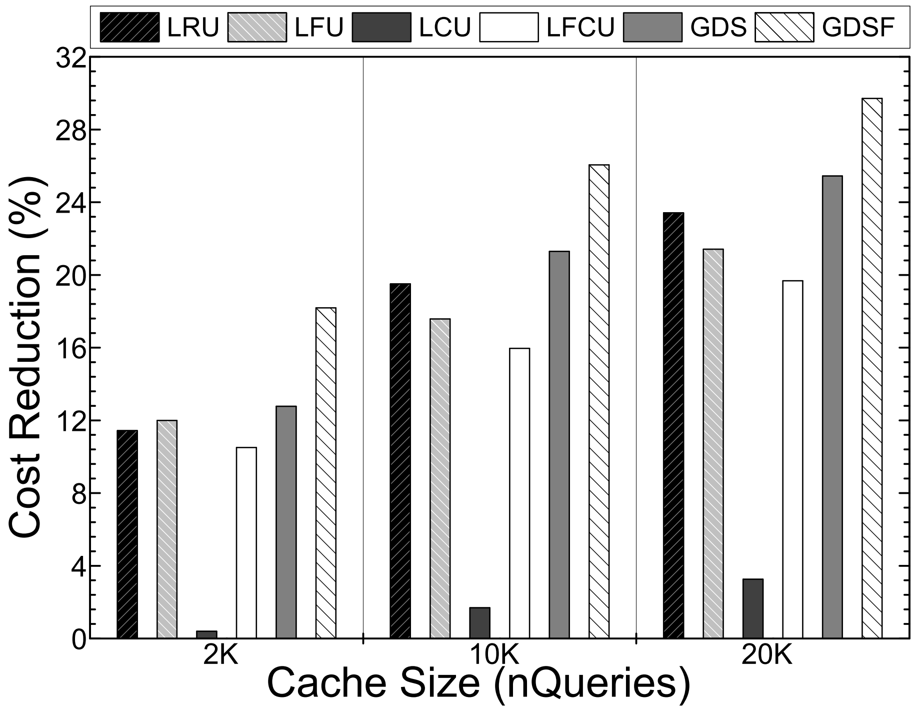


Figure 4.6: Resource-Level Eviction, s(R) = 50

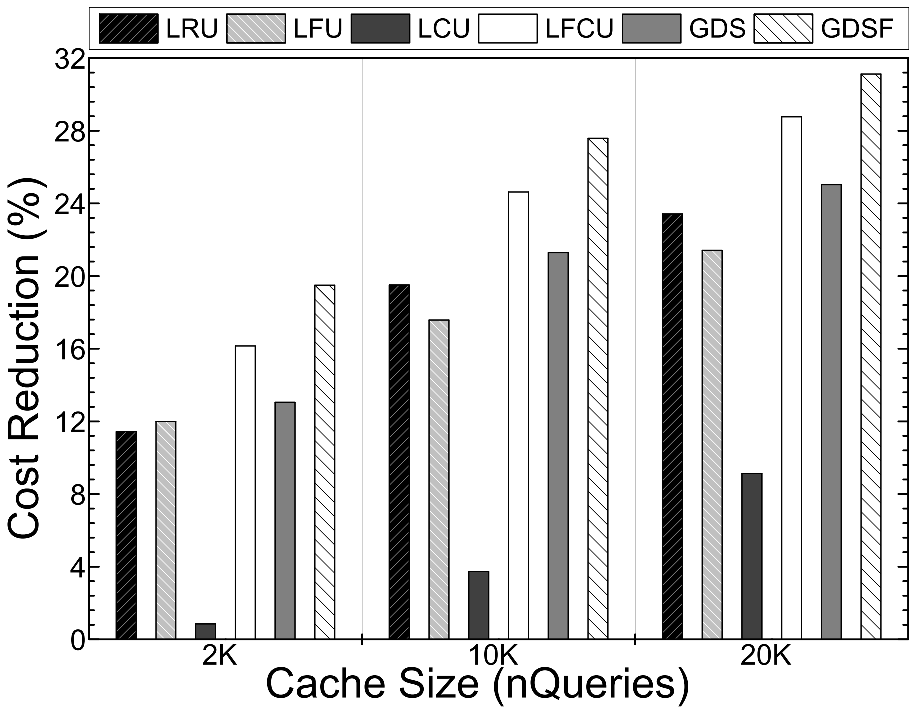


Figure 4.7: Entity-Level Eviction, s(R) = 50

Table 4.4: GPACS vs. Nearest Competitor - Cost Reduction % Comparison

	Competitor		GPACSe1	GPACSe2
s(R) = 10, CS = 100K	EL-LFCU	38.42	38.705	39.093
s(R) = 10, CS = 50K	EL-GDSF	35.589	36.162	36.164
s(R) = 10, CS = 10K	EL-GDSF	27.693	28.427	28.404
s(R) = 50, CS = 20K	EL-GDSF	31.116	31.735	31.911
s(R) = 50, CS = 10K	EL-GDSF	26.054	28.368	28.325
s(R) = 50, CS = 2K	EL-GDSF	18.185	19.973	19.94

In Table 4.4 we compare best performing approach for each parameter setting to our *GPACS* algorithm. We see that, *GPACS* yields up to 2% reduction in miss cost. The gains are more emphasized for small and moderate size caches, as all approaches tend to converge for very large cache sizes. Remarkably, *GPACS* provides larger reductions in cost when the number of resources increased.

According to the results presented in Table 4.4, between *GPACSe1* and *GPACSe2* there is generally no significant performance difference. Yet it seems that *GPACSe2* starts to outperform its counterpart as the cache size grows and passes some threshold. It is an expected result since as the cache size grows, the costs of the entries stored in the cache gets more important relative to the number of entries stored in the cache (quality over quantity).

Even though the real aim in cost-aware caching strategies is to lower the cost of query processing (i.e., achieving a higher cost reduction percentage), the performances using cache full-hit rate and cache-hit rate metrics are also briefly discussed. While doing so, we compare the *GPACS* versions with the best-performing competitor, namely, *EL-GDSF*.

Table 4.5: Comparison of GPACS and EL-GDSF Approaches Using Hit Rate and Full-hit Rate Metrics

	Policy	Full-Hit Rate	Hit Rate	Full-Hit/Hit Rate
s(R) = 10, CS = 100K	EL-GDSF	34.712%	37.738%	91.982%
s(R) = 10, CS = 100K	GPACSe1	36.492%	37.651%	96.922%
s(R) = 10, CS = 100K	GPACSe2	36.551%	37.89%	96.466%
s(R) = 10, CS = 50K	EL-GDSF	32.228%	34.805%	92.596%
s(R) = 10, CS = 50K	GPACSe1	34.04%	34.879%	97.595%
s(R) = 10, CS = 50K	GPACSe2	34.04%	34.896%	97.547%
s(R) = 10, CS = 10K	EL-GDSF	25.153%	26.979%	93.232%
s(R) = 10, CS = 10K	GPACSe1	26.722%	27.331%	97.772%
s(R) = 10, CS = 10K	GPACSe2	26.697%	27.309%	97.759%
s(R) = 50, CS = 20K	EL-GDSF	27.451%	30.602%	89.703%
s(R) = 50, CS = 20K	GPACSe1	30.057%	30.708%	97.880%
s(R) = 50, CS = 20K	GPACSe2	30.24%	30.902%	97.858%
s(R) = 50, CS = 10K	EL-GDSF	24.489%	27.082%	90.425%
s(R) = 50, CS = 10K	GPACSe1	26.893%	27.451%	97.967%
s(R) = 50, CS = 10K	GPACSe2	26.856%	27.412%	97.972%
s(R) = 50, CS = 2K	EL-GDSF	17.992%	19.168%	93.865%
s(R) = 50, CS = 2K	GPACSe1	19.134%	19.432%	98.466%
s(R) = 50, CS = 2K	GPACSe2	19.101%	19.4%	98.459%

Table 4.5 shows that *GPACS* versions yield higher ratios of Full-Hit Rate to Hit Rate than *EL-GDSF*. Since *GPACS* can evict entries with higher costs and indexes as long as they have a smaller greedy value, average number of different queries in the evic-

tion list of a single eviction operation for *GPACS* is generally lower than its competitors. Therefore, we expect the number of full vectors in the cache to be higher when *GPACS* is used. Results seem to be in line with our expectations.

As a conclusion to our simulation results, it would be safe to say that regardless of the cache size or number of resources used, our novel GPACS algorithm outperforms its competitors for all of the performance metrics (i.e., cost reduction rate, full-hit rate, and hit rate).

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis, the topic of cost-aware result caching for meta-search engines is addressed. We defined three different caching approaches, namely, query-, resource- and entry-level, based on the granularity of the query result stored in the cache. Then, via large-scale simulations, we evaluated the performance of each caching approach together with various traditional and cost-aware eviction policies from the literature. We also proposed a new entry-level caching algorithm, so-called *GPACS* that is again cost-aware and fits better to the requirements of a meta-search scenario.

5.1 Main Findings

Our simulation results revealed that query-level approaches are superior to their resource-level counterparts; and entry-level approaches are superior to both of the latter, in terms of the miss-cost metric. Moreover, the novel *GPACS* approach outperformed all of its competitors that are coupled with either traditional eviction policies (such as *LRU* and *LFU*) or cost-aware policies as described in [25].

These findings indicate that managing a meta-search engine's cache with an entry-level caching approach is the most effective solution in terms of reducing the overall miss-cost. However, there is a trade-off between the query miss-costs and actual cache retrieval times for the entry- and resource-level eviction approaches, as the latter require more complicated data structures to manage the cache content than the simple query-level approach. This trade-off should also be taken into account while constructing a result cache for a real-life meta-search engine.

5.2 Future Work

There are several exciting research directions that can be followed as a future work. We list some of these as follows:

- *Cache Complexity:* The complexity of the actual caching algorithm is not investigated in this thesis. Clearly, the approaches that yield the highest miss-cost reductions are also those that are more complicated; and hence, they need to be implemented efficiently for practical use-cases. In our future work, we plan to explore the trade-off between the performance of caching algorithm and its complexity; especially for the cases with a large number of resources; and develop efficient data structures to manage the cache content.

- *Different Cost Functions:* In this thesis, the global cost of a query is its processing time and hence, it is computed as the maximum value among all local costs; i.e., the maximum of the costs for retrieving the local results from each resource. However, not all cost functions are like this; for instance, if the cost is defined in terms of the network (bandwidth) usage or financial cost; then we would need to add the cost for each resource, rather than taking the maximum. It is an exciting research direction to explore how the caching strategies discussed in this thesis perform with such additive cost types; or with hybrid cost functions that try to capture different types of costs.

- *Cache Freshness:* Another promising future work direction is identifying and refreshing the stale results in a meta-search engine cache. As we store all the local results obtained from resources with varying dynamicity, the interaction of the caching polices with refreshment strategies in such a setup would yield new challenges that will worth to address.

REFERENCES

[1] Sadiye Alici, Ismail Sengor Altingovde, Rifat Ozcan, B. Barla Cambazoglu, and Özgür Ulusoy. Adaptive time-to-live strategies for query result caching in web search engines. In *Proceedings of the 34th European Conference on IR Research*, ECIR '12, pages 401–412, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Performance Evaluation Review*, 27(4):3–11, March 2000.

[3] Ricardo Baeza-Yates, Flavio Junqueira, Vassilis Plachouras, and Hans Friedrich Witschel. Admission policies for caches of search engine results. In *Proceedings of the 14th International Conference on String Processing and Information Retrieval*, SPIRE '07, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Ricardo A. Baeza-Yates and Simon Jonassen. Modeling static caching in web search engines. In *Proceedings of the 34th European Conference on IR Research, ECIR '12, Barcelona, Spain*, pages 436–446, 2012.

[5] Ricardo A. Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval, SPIRE '03, Manaus, Brazil*, pages 56–65, 2003.

[6] Xiao Bai and Flavio P. Junqueira. Online result cache invalidation for real-time web search. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 641–650, New York, NY, USA, 2012. ACM.

[7] Roi Blanco, Edward Bortnikov, Flavio Junqueira, Ronny Lempel, Luca Telloli, and Hugo Zaragoza. Caching search engine results over incremental indices. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 82–89, New York, NY, USA, 2010. ACM.

[8] Edward Bortnikov, Ronny Lempel, and Kolman Vornovitsky. Caching for real-time search. In *Proceedings of the 33rd European Conference on IR Research*, ECIR '11, pages 104–116, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Berkant Barla Cambazoglu and Ismail Sengör Altingövde. Impact of regionalization on performance of web search engine result caches. In *Proceedings*

of the 19th International Symposium on String Processing and Information Retrieval, SPIRE '12, Cartagena de Indias, Colombia, pages 161–166, 2012.

[10] Berkant Barla Cambazoglu, Flavio P. Junqueira, Vassilis Plachouras, Scott Banachowski, Baoqiu Cui, Swee Lim, and Bill Bridge. A refreshing perspective of search engine caching. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 181–190, New York, NY, USA, 2010. ACM.

[11] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS '97, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[12] Diego Ceccarelli, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Caching query-biased snippets for efficient retrieval. In *Proceedings of the 14th International Conference on Extending Database Technology, EDBT '11, Uppsala, Sweden*, pages 93–104. ACM, 2011.

[13] Boris Chidlovskii, Claudia Roncancio, and Marie-Luise Schneider. Semantic cache mechanism for heterogeneous web querying. *Computer Networks*, 31(11-16):1347–1360, 1999.

[14] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems*, 24(1):51–78, January 2006.

[15] Qingqing Gan and Torsten Suel. Improved techniques for result caching in web search engines. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 431–440, New York, NY, USA, 2009. ACM.

[16] Simon Jonassen, Berkant Barla Cambazoglu, and Fabrizio Silvestri. Prefetching query results and its impact on search engines. In *Proceedings of the 35th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '12, Portland, OR, USA*, pages 631–640, 2012.

[17] Sang Ho Lee, Jin Seon Hong, and Larry Kerschberg. A popularity-driven caching scheme for meta-search engines: An empirical study. In *Proceedings of the 12th International Conference on Database and Expert Systems Applications, DEXA '11, Munich, Germany*, pages 877–886, 2001.

[18] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 19–28, New York, NY, USA, 2003. ACM.

[19] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 257–266, New York, NY, USA, 2005. ACM.

[20] Mauricio Marin, Veronica Gil-Costa, and Carlos Gomez-Pantoja. New caching techniques for web search engines. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 215–226, New York, NY, USA, 2010. ACM.

[21] Evangelos P. Markatos. On caching search engine query results. *Computer Communications*, 24(2), 2001.

[22] Rifat Ozcan, I. Sengor Altingovde, B. Barla Cambazoglu, Flavio P. Junqueira, and Özgür Ulusoy. A five-level static cache architecture for web search engines. *Information Processing & Management*, 48(5):828–840, 2012.

[23] Rifat Ozcan, Ismail Sengör Altingövde, Berkant Barla Cambazoglu, and Özgür Ulusoy. Second chance: A hybrid approach for dynamic result caching and prefetching in search engines. *ACM Transactions on the Web*, 8(1):3, 2013.

[24] Rifat Ozcan, Ismail Sengör Altingövde, and Özgür Ulusoy. Static query result caching revisited. In *Proceedings of the 17th International Conference on World Wide Web, WWW '08, Beijing, China*, pages 1169–1170, 2008.

[25] Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Transactions on the Web*, 5(2):9:1–9:25, May 2011.

[26] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[27] S. Podlipnig and L. Boszormenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[28] Paricia Correia Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, pages 51–58, New York, NY, USA, 2001. ACM.

[29] Fethi Burak Sazoglu, Berkant Barla Cambazoglu, Rifat Ozcan, Ismail Sengör Altingövde, and Özgür Ulusoy. A financial cost metric for result caching. In *Proceedings of the 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland*, pages 873–876, 2013.

[30] Fethi Burak Sazoglu, Berkant Barla Cambazoglu, Rifat Ozcan, Ismail Sengör Altingövde, and Özgür Ulusoy. Strategies for setting time-to-live values in result caches. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management, CIKM '13, San Francisco, CA, USA*, pages 1881–1884, 2013.

[31] Milad Shokouhi and Luo Si. Federated search. *Foundations and Trends in Information Retrieval*, 5(1):1–102, January 2011.

[32] Fabrizio Silvestri. Mining query logs: Turning search usage data into knowledge. *Foundations and Trends in Information Retrieval*, 4(1-2):1–174, 2010.

[33] Gabriel Tolosa, Luca Becchetti, Esteban Feuerstein, and Alberto Marchetti-Spaccamela. Performance improvements for search systems using an integrated cache of lists+intersections. In *Proceedings of the 21st International Symposium on String Processing and Information Retrieval, SPIRE '14, Ouro Preto, Brazil*, pages 227–235, 2014.

[34] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

[35] Yohannes Tsegay, Simon J. Puglisi, Andrew Turpin, and Justin Zobel. Document compaction for efficient query biased snippet generation. In *Proceedings of the 31th European Conference on IR Research*, ECIR '09, pages 509–520, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] Andrew Turpin, Yohannes Tsegay, David Hawking, and Hugh E. Williams. Fast generation of result snippets in web search. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 127–134, New York, NY, USA, 2007. ACM.

[37] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. The impact of solid state drive on search engine cache management. In *Proceedings of the 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland*, pages 693–702, 2013.

[38] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. Cache design of ssd-based search engine architectures: An experimental study. *ACM Transactions on Information Systems*, 32(4):21, 2014.

[39] Zonghuan Wu, Weiyi Meng, Clement Yu, and Zhuogang Li. Towards a highly-scalable and effective metasearch engine. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 386–395, New York, NY, USA, 2001. ACM.

APPENDIX A

ALL SIMULATION RESULTS

Table A.1: Simulation Result - s(R)=10, Query-Level Eviction

Policy	Cache Size	Hit Rate	Full Hit Rate	Cost Reduction
LRU	10,000	19.448%	19.448%	19.562%
LRU	50,000	28.965%	28.965%	29.065%
LRU	100,000	33.542%	33.542%	33.636%
LFU	10,000	17.515%	17.515%	17.638%
LFU	50,000	27.062%	27.062%	27.171%
LFU	100,000	31.766%	31.766%	31.856%
LCU	10,000	1.665%	1.665%	1.836%
LCU	50,000	9.211%	9.211%	10.023%
LCU	100,000	16.889%	16.889%	18.063%
LFCU	10,000	15.931%	15.931%	16.469%
LFCU	50,000	24.93%	24.93%	25.621%
LFCU	100,000	30.376%	30.376%	31.085%
GDS	10,000	21.244%	21.244%	21.384%
GDS	50,000	30.995%	30.995%	31.137%
GDS	100,000	35.475%	35.475%	35.636%
GDSF	10,000	25.996%	25.996%	26.124%
GDSF	50,000	34.335%	34.335%	34.456%
GDSF	100,000	37.437%	37.437%	37.573%

Table A.2: Simulation Result - s(R)=10, Resource-Level Eviction

Policy	**Cache Size**	**Hit Rate**	**Full Hit Rate**	**Cost Reduction**
LRU	10,000	19.448%	19.448%	19.562%
LRU	50,000	28.965%	28.965%	29.065%
LRU	100,000	33.542%	33.542%	33.636%
LFU	10,000	17.515%	17.515%	17.638%
LFU	50,000	27.062%	27.062%	27.171%
LFU	100,000	31.766%	31.766%	31.856%
LCU	10,000	1.999%	0.039%	0.764%
LCU	50,000	8.904%	0.039%	4.494%
LCU	100,000	16.628%	0.039%	8.8%
LFCU	10,000	15.839%	8.353%	12.694%
LFCU	50,000	25.018%	12.991%	20.235%
LFCU	100,000	30.333%	15.989%	24.939%
GDS	10,000	21.075%	18.585%	20.686%
GDS	50,000	30.888%	28.223%	30.374%
GDS	100,000	35.421%	32.523%	34.752%
GDSF	10,000	25.887%	23.769%	25.538%
GDSF	50,000	34.233%	32.195%	33.856%
GDSF	100,000	37.391%	35.511%	36.975%

Table A.3: Simulation Result - s(R)=10, Entity-Level Eviction

Policy	**Cache Size**	**Hit Rate**	**Full Hit Rate**	**Cost Reduction**
LRU	10,000	19.448%	19.448%	19.562%
LRU	50,000	28.965%	28.965%	29.065%
LRU	100,000	33.542%	33.542%	33.636%
LFU	10,000	17.515%	17.515%	17.638%
LFU	50,000	27.062%	27.062%	27.171%
LFU	100,000	31.766%	31.766%	31.856%
LCU	10,000	1.936%	0.039%	7.505%
LCU	50,000	8.396%	0.039%	20.284%
LCU	100,000	15.969%	0.039%	25.802%
LFCU	10,000	23.733%	23.404%	24.046%
LFCU	50,000	33.362%	30.363%	34.98%
LFCU	100,000	35.928%	32.313%	38.42%
GDS	10,000	20.297%	14.712%	21.985%
GDS	50,000	29.655%	23.086%	31.04%
GDS	100,000	33.666%	26.172%	35.049%
GDSF	10,000	26.979%	25.153%	27.693%
GDSF	50,000	34.805%	32.228%	35.589%
GDSF	100,000	37.738%	34.712%	38.165%
GPACSe1	10,000	27.309%	26.697%	28.404%
GPACSe1	50,000	34.879%	34.04%	36.162%
GPACSe1	100,000	37.651%	36.492%	38.705%
GPACSe2	10,000	27.331%	26.722%	28.427%
GPACSe2	50,000	34.896%	34.04%	36.164%
GPACSe2	100,000	37.89%	36.551%	39.093%

Table A.4: Simulation Result - s(R)=50, Query-Level Eviction

Policy	Cache Size	Hit Rate	Full Hit Rate	Cost Reduction
LRU	2,000	11.37%	11.37%	11.431%
LRU	10,000	19.448%	19.448%	19.506%
LRU	20,000	23.355%	23.355%	23.41%
LFU	2,000	11.924%	11.924%	11.999%
LFU	10,000	17.515%	17.515%	17.575%
LFU	20,000	21.356%	21.356%	21.414%
LCU	2,000	0.388%	0.388%	0.4%
LCU	10,000	1.634%	1.634%	1.687%
LCU	20,000	3.159%	3.159%	3.259%
LFCU	2,000	10.365%	10.365%	10.507%
LFCU	10,000	15.751%	15.751%	15.963%
LFCU	20,000	19.432%	19.432%	19.676%
GDS	2,000	12.703%	12.703%	12.773%
GDS	10,000	21.23%	21.23%	21.297%
GDS	20,000	25.381%	25.381%	25.445%
GDSF	2,000	18.122%	18.122%	18.185%
GDSF	10,000	25.995%	25.995%	26.054%
GDSF	20,000	29.651%	29.651%	29.706%

Table A.5: Simulation Result - s(R)=50, Resource-Level Eviction

Policy	**Cache Size**	**Hit Rate**	**Full Hit Rate**	**Cost Reduction**
LRU	2,000	11.37%	11.37%	11.431%
LRU	10,000	19.448%	19.448%	19.506%
LRU	20,000	23.355%	23.355%	23.41%
LFU	2,000	11.924%	11.924%	11.999%
LFU	10,000	17.515%	17.515%	17.575%
LFU	20,000	21.356%	21.356%	21.414%
LCU	2,000	0.391%	0.039%	0.104%
LCU	10,000	1.809%	0.039%	0.458%
LCU	20,000	3.516%	0.039%	0.837%
LFCU	2,000	9.992%	6.334%	7.382%
LFCU	10,000	15.907%	7.035%	10.002%
LFCU	20,000	19.642%	7.781%	12.156%
GDS	2,000	12.587%	9.672%	11.905%
GDS	10,000	21.108%	17.119%	20.152%
GDS	20,000	25.259%	21.151%	24.287%
GDSF	2,000	18.024%	14.964%	17.281%
GDSF	10,000	25.91%	22.573%	25.072%
GDSF	20,000	29.571%	26.068%	28.704%

Table A.6: Simulation Result - s(R)=50, Entity-Level Eviction

Policy	**Cache Size**	**Hit Rate**	**Full Hit Rate**	**Cost Reduction**
LRU	2,000	11.37%	11.37%	11.431%
LRU	10,000	19.448%	19.448%	19.506%
LRU	20,000	23.355%	23.355%	23.41%
LFU	2,000	11.924%	11.924%	11.999%
LFU	10,000	17.515%	17.515%	17.575%
LFU	20,000	21.356%	21.356%	21.414%
LCU	2,000	0.402%	0.039%	0.847%
LCU	10,000	1.814%	0.039%	3.732%
LCU	20,000	3.535%	0.039%	9.128%
LFCU	2,000	16.044%	15.987%	16.148%
LFCU	10,000	24.535%	23.977%	24.623%
LFCU	20,000	28.726%	27.246%	28.766%
GDS	2,000	12.097%	6.495%	13.053%
GDS	10,000	20.316%	12.584%	21.29%
GDS	20,000	24.331%	15.798%	25.032%
GDSF	2,000	19.168%	17.992%	19.5%
GDSF	10,000	27.082%	24.489%	27.585%
GDSF	20,000	30.602%	27.451%	31.116%
GPACSe1	2,000	19.432%	19.134%	19.973%
GPACSe1	10,000	27.451%	26.893%	28.368%
GPACSe1	20,000	30.708%	30.057%	31.735%
GPACSe2	2,000	19.4%	19.101%	19.94%
GPACSe2	10,000	27.412%	26.856%	28.325%
GPACSe2	20,000	30.902%	30.24%	31.911%