Caching query-biased snippets for efficient retrieval

Diego Ceccarelli
Dipartimento di Informatica
Università di Pisa
diego.ceccarelli@isti.cnr.it

Claudio Lucchese
ISTI-CNR
Pisa
claudio.lucchese@isti.cnr.it

Salvatore Orlando
Università Ca' Foscari
Venezia
orlando@unive.it

Raffaele Perego
ISTI-CNR
Pisa
raffaele.perego@isti.cnr.it

Fabrizio Silvestri
ISTI-CNR
Pisa
fabrizio.silvestri@isti.cnr.it

ABSTRACT

Web Search Engines' result pages contain references to the top-k documents relevant for the query submitted by a user. Each document is represented by a title, a snippet and a URL. Snippets, i.e. short sentences showing the portions of the document being relevant to the query, help users to select the most interesting results.

The snippet generation process is very expensive, since it may require to access a number of documents for each issued query. We assert that *caching*, a popular technique used to enhance performance at various levels of any computing systems, can be very effective in this context. We design and experiment several cache organizations, and we introduce the concept of *supersnippet*, that is the set of sentences in a document that are more likely to answer future queries. We show that supersnippets can be built by exploiting query logs, and that in our experiments a supersnippet cache answers up to **62**% of the requests, remarkably outperforming other caching approaches.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - *Data Mining.*; H.3.3 [**Information Search and Retrieval**]: Search process, Query formulation.

General Terms

Algorithms, Experimentation.

Keywords

Web Search Engines, Snippet Generation, Caching, Efficiency, Throughput.

1. INTRODUCTION

Nowadays a Web Search Engine (WSE) is a very complex software system [1]. It is well known that the goal of a WSE is to answer users' queries both *effectively*, with relevant results, and *efficiently*, in a very short time frame. After a user issues a query, the WSE actually runs a chain of complex processing phases producing the Search Engine Results Page (SERP). A SERP contains a list of few (usually 10) results. Each result corresponds to a Web page, and it contains the title of the page, its URL, and a text *snippet*, i.e. a brief text summarizing the content of the page.

The true goal is to *scale-up* with the growth of Web documents and users. The services offered by a WSE exploit a significant amount of storage and computing resources. Resource demand must be however kept as small as possible in order to achieve scalability. During query processing, document indexes are accessed to retrieve the list of identifiers of the most relevant documents to the query. Second, a view of each document in the list is rendered. Given a document identifier, the WSE accesses the document repository that stores permanently on disk the content of the corresponding Web page, from which a summary, i.e. the snippet, is extracted. In fact, the snippet is usually *query-dependent*, and shows a few fragments of the Web page that are most relevant to the issued query. The snippets, page URLs, and page titles are finally returned to the user.

Snippets are fundamental for the users to estimate the relevance of the returned results: high quality snippets greatly help users in selecting and accessing the most interesting Web pages. It is also known that the snippet quality depends on the ability of producing a *query-biased* summary of each document [16] that tries to capture the most important passages related with the user query. Since most user queries cannot be forecasted in advance, these snippets cannot be produced off-line, and their on-line construction is a heavy load for modern WSEs, which have to process hundreds of millions queries per day. In particular, the cost of accessing several different files (containing the Web pages) for each query, retrieved among terabytes of data, under heavy and unpredictable load conditions, may be beyond the capability of traditional filesystems and may require special purpose filesystems [18].

In this paper we are interested in studying the performance aspects of the snippet extraction phase and we devise techniques that can increase the query processing throughput and reduce the average query response time by speeding-up snippet extraction phase. In particular, we leverage on a popular technique used to enhance performance of computing systems, namely *caching* [13]. Caching techniques are already largely exploited by WSEs at various system levels

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00

to improve query processing, mainly for storing past queries and associated sets of results along with the query-biased snippets [5].

The basic technique adopted for Web search caching is to store the whole result page for each cached query. When a cache hit occurs, the result page is immediately sent back to the user. This approach perfectly fits queries being in the "*head*" of the power-law characterizing the query topic distribution. On the other hand, SERP caching is likely to fail in presence of personalization, that is when the engine produces two different SERPs for the same query submitted by two different users. Furthermore, it fails when a query has not been previously seen or it is a singleton query, i.e. it will not be submitted again in the future. Baeza-Yates *et al.* [2] report that approximatively 50% of the queries received by a commercial WSE are singleton queries.

Unlike query-results caching, snippets caching that is the focus of this paper, has received relatively low attention. The two research efforts closest to ours are those by Turpin *et al.* and Tsegay *et al.* [18, 17]. The authors investigate the effect of lossless and lossy compression techniques to generate documents surrogates that are statically cached in memory. They argue that complex compression algorithms can effectively shrink large collection of texts and accommodate more surrogates within the cache. As a trade-off, complex decompression increases the time needed by the cache to serve a hit, and are thus unlikely to decrease the average snippets generation time [18]. Lossy compression techniques produce surrogates by reordering and pruning sentences from the original documents. They reduce the size of the cached documents, still retaining the ability of producing high quality snippets. Tsegay *et al.* measured that in the 80% of the cases snippets generated from surrogates that are the 50% of the original collection size, are identical to the ones generated from the non-compressed documents [17].

To the best of our knowledge, this is the first research studying techniques for dynamically generating document surrogates to be managed by a snippet caching system. Furthermore, the surrogates generation method is completely new, since the sentence selection is based on the past queries submitted to the WSE. Our thesis is that the knowledge stored in query logs can help in building concise surrogates, that we call *supersnippets*, which we prove to be very effective for the efficient retrieval of high quality snippets.

The rest of the paper is organized as follows. In Section 2, by analyzing a query log, we show that the the snippet generation process have a significant locality, which supports the use of caching techniques. Section 3 formalizes the architectural framework adopted and introduces two baseline snippet cache organization. In Section 4 we introduce the concept of *supersnippet*, and we propose a caching algorithm for building and maintaing supersnippets. Then, in Section 5 we report on the effectiveness of our proposed algorithm, while in Section 6 we show the performance results. Section 7 illustrates relevant related work, and finally, in Section 8 we draw some conclusions.

2. MOTIVATIONS

This paper proposes a new approach for generating query-biased snippets from concise surrogates of documents cached in main memory. Our thesis is that the knowledge stored in query logs can help in building concise surrogates that allow high-quality query-biased snippets to be efficiently generated. We thus start our study by analyzing a real-world query log, with the aim of understanding the characteristics and the popularity distribution of URLs, documents and snippets returned by a WSE.

Query Log	Queries	Distinct Queries	Distinct Urls (top 10)
D1			
training	1,000,000	601,369	4,317,603
test	500,000	310,495	2,324,295
D2			
training	9,000,000	4,447,444	25,897,247
test	4,500,000	2,373,227	12,134,453

Table 1: Main characteristics of the samples of the MSN query log used for the experiments.

In our experiments we used the MSN Search query log excerpt (RFP 2006 dataset provided by Microsoft). Such query log contains approximately 15 million queries sampled over one month (May 2006), and coming from a US Microsoft search site (therefore, most queries are in English). In particular, we consider two distinct datasets extracted from the query log:

- D1 contains 1,500,000 queries and it was used for analysis purposes and to motivate our approach;
- D2 contains 14,000,000 queries and it was used to experimentally assess the performance of our technique.

Both datasets were further split in a training and a testing segment. Table 1 reports the main characteristics of the samples of the two datasets used for the experiments.

For each query in the two datasets we retrieved the corresponding SERP via the Yahoo! Search BOSS API. Thus, for each query we stored the top-10 most relevant URLs and query-biased snippets as provided by Yahoo!. Moreover, we downloaded also the documents corresponding to all the URLs returned for the distinct queries occurring in the smallest dataset D1.

2.1 Analysis of the query logs

The first analysis conducted by using the D1 dataset tries to answer a very simple question: is there temporal locality in the accesses to documents that are processed for extracting snippets to be inserted in the SERPs? If the answer to this question is positive, we can think to minimize the cost of such accesses by means of some caching technique. The question is somehow trivial to answer if we consider the high sharing of query topics studied in several papers (e.g., [5]). If the same query topics are shared by several users, the same should happen for the top results returned by the WSE for these queries.

We measured directly on our dataset the popularity distribution of documents occurring within the top-10 URLs retrieved for user queries in D1: as expected, the plot in Figure 1 shows that document popularity follows a power-law distribution. By looking with attention at the plot, we can already claim a presence of locality in the accesses to documents that goes beyond the locality in the user queries. We can in fact note that the top-10 most frequently accessed documents do not have the same popularity. This means that some of the top-10 documents returned for the most frequent query present in the log are returned among the results of some other query. An high sharing of URLs retrieved is thus present due to the well-known power-law distribution of query topics popularity, but also due to the sharing of the same URLs in the results' sets of different queries. From Table 1 we can see that `D1` contains about 912k distinct queries out of 1.5 millions, but only 6,640k distinct URLs occur among the 15 millions documents returned as top-10 results for these queries. Note that if all the results returned for the distinct queries would be completely disjoint, the distinct URLs should be 9,120k, about 27% more.

This high sharing in the URLs returned to WSE users surely justifies the adoption of caching techniques. Moreover, the sharing of URLs also among the results of different queries motivates the adoption of caching at the document level for speeding-up the generation of snippets, in addition to the query results cache commonly exploited in WSEs. Unfortunately, caching documents is very demanding in term of amount of fast memory needed to store even the most frequently accessed document.

However, other researchers already investigated the exploitation of lossless or lossy compression techniques for reducing memory demand, and proved that effective snippets can be generated also from document surrogates that retain less than half of the original document content [18, 17]. In this paper we proceed further in the same direction, and propose an effective, usage-based technique for generating and managing in a cache much more concise document surrogates.

Having this goal in mind, next question we must answer is the following: how many different snippets have to be generated for a single document? The more the number of different snippets generated by a single document, richer the content and larger the document surrogate from which these snippets can be generated.

Let us denote by δ_u the number of different snippets associated with URL u. Figure 2 shows that also δ_u follows a power-law distribution: a few URLs have several different snippets being generated and returned to the user. Indeed, about 99.96% of URLs have less than 10 snippets associated with them, and about 92.5% of URLs have just a single snippet associated with them. This means that the large majority of documents satisfies a single information need, and therefore just one snippet is usefully retrieved for them. On the other side of the coin, we have that about 8% of documents retrieved originate more than one snippet. These documents are potentially retrieved by different queries addressing different portions of the same document. To generate efficiently high-quality snippets for these documents may thus require to cache a richer surrogate.

Our thesis is that in most cases, even when the same URL is retrieved by different queries, the snippets generated are very similar. This happens, for example, in the case of query specializations and reformulations, synonyms, spell corrections, query-independent snippets such as those returned for navigational queries. To further investigate this point, in Figure 3 we show, for the 1,000 most frequently retrieved URLs, the number of distinct queries that retrieved them, and the corresponding number of distinct snippets generated. It is apparent that while many distinct queries retrieve the same document, only a small number of snippets, typically less than 10, is generated. This proves that when the same URL answers distinct queries, most of these different queries share exactly the same snippet.

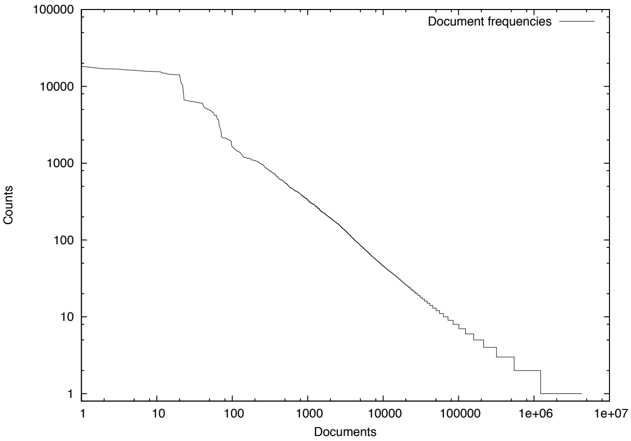

Figure 1: Popularity distribution of documents retrieved among the top-10 results (log-log scale).

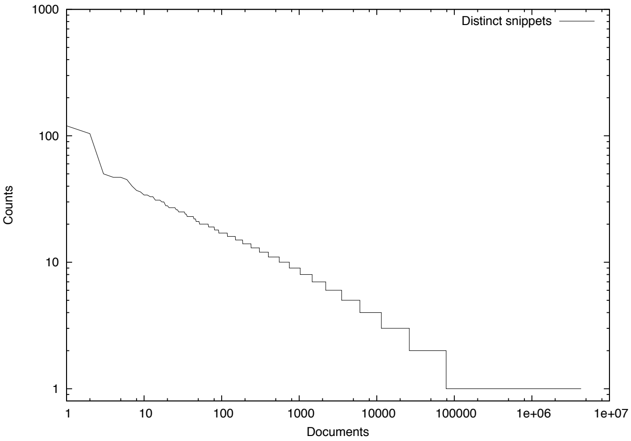

Figure 2: Number of distinct snippets per document distribution (log-log scale)

Our proposal tries to exploit this interesting fact by introducing a novel snippeting strategy that allows to devise concise and effective document surrogates based on the past queries submitted to the WSE, and to exploit similarities among queries retrieving the same URL.

3. ARCHITECTURAL FRAMEWORK

The WSE subsystem responsible for query answering, as sketched in Figure 4, is usually made up of three cooperating components [3]:

- The WSE Front-End (FE) is in charge of managing the interactions with users: it receives a stream of queries, and returns the result pages built by exploiting the two other components.

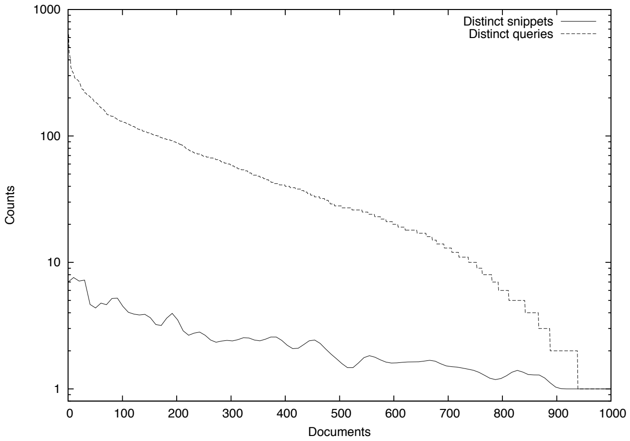

Figure 3: For the top-1,000 popular documents: number of distinct queries retrieving each document and number of distinct snippets associated with that document.

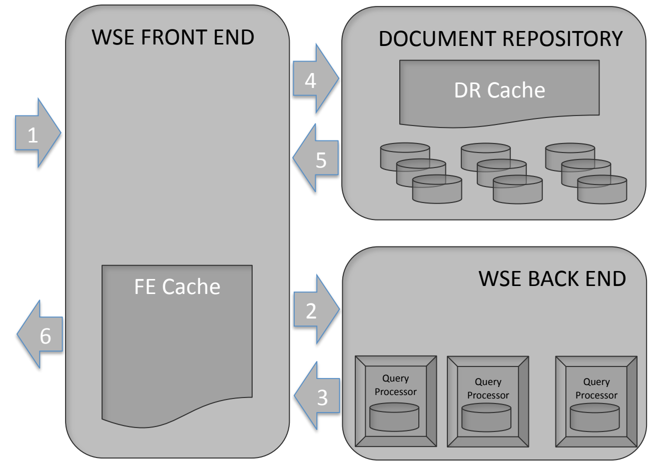

Figure 4: Sketch of the architecture of our testbed.

- The WSE Back-End (BE), for each query received from FE, extracts the top-k most relevant documents in its possibly distributed indexes, and returns to FE the corresponding document identifiers (docIDs).

- The WSE Document Repository (DR) is assigned the task of producing the result page by enriching and making user-understandable the ranked lists of docIDs returned by FE. More specifically, for each docID in the result set, DR retrieve the corresponding URL, and generates the snippet that summarizes, as shortly and clearly as possible, the query-biased content of the associated document.

In Figure 4, we illustrate the data flow. The user query q submitted to the WSE is received by FE (step 1). This forwards the query to BE (step 2) which is in charge of finding, via the distributed index, the most relevant documents to the query, and returns to FE the corresponding document identifiers (step 3). FE sends the query and the docID list to DR (step 4) which returns the corresponding query-biased snippets (step 5). Finally, the SERP is built by FE and returned to the user (step 6).

In order to improve throughput and save redundant computations, all the three components may exploit caching. FE caches SERPs of frequent queries, BE may cache portions of the postings lists of terms frequently occurring in user queries, and DR may cache frequently requested documents. In this paper we focus on issues deriving from the use of caching on DR. For its evaluation we also consider the presence of a query result cache on FE, but the caching of postings lists, exploited by the various Query Processor in BE, is out of the scope of this work.

FE Cache.

FE usually hosts a large cache storing results of recently submitted queries. This cache was proved to be very effective and capable of answering more than one third of all the queries submitted to commercial WSEs without involving the BE [11, 8, 5, 2, 6]. The cache is accessed by supplying the current query as the key. Whenever a hit occurs, the FE may avoid to forward the query to BE, since the most relevant document are already present in the cache. The steps 2 and 3 are skipped, thus reducing the load at the BE. Depending on the organization of the FE cache, the DR may or may not invoked for generating the snippets. We distinguish between two possible caches types hosted on FE:

- $\mathsf{RCache}_{\mathsf{DocID}}$ stores in each entry the docIDs of the most relevant documents to a given query;

- $\mathsf{RCache}_{\mathsf{SERP}}$ uses larger entries to store all the information contained in a SERP, that is both the URLs and query-biased snippets.

In case of a miss in a $\mathsf{RCache}_{\mathsf{DocID}}$ cache, the most relevant documents are requested from BE (steps 2,3), and inserted into the cache with key q for future reference. Since snippets and URLs are not stored, both in case of a hit and in case of a miss, these must be requested from DR (steps 4,5).

If FE adopts the $\mathsf{RCache}_{\mathsf{SERP}}$, when a hit occurs, FE can retrieve the SERP for the query, and promptly return it to the user (step 6). The DR is not invoked, since the cache entries also stores URLs and snippets. In case of a miss, both the BE and DR must be invoked for producing the result page, which is also cached into $\mathsf{RCache}_{\mathsf{SERP}}$.

The type of result cache hosted on FE clearly affects the stream of requests processed by DR.
If FE hosts an $\mathsf{RCache}_{\mathsf{DocID}}$ cache, DR must process all the incoming WSE queries. It is worth noting that, from the point of view of the DR workload, this case is exactly the same as an architecture where no result caches are present. On the other hand, the presence of an $\mathsf{RCache}_{\mathsf{SERP}}$ cache on FE strongly reduces the volume of requests issued to DR, since only the queries resulting in misses on $\mathsf{RCache}_{\mathsf{SERP}}$ generate requests for URLs and snippet extractions.

DR Cache.

Given a query q and a document identifier $docID$, DR retrieves the URL of the corresponding document and generates a query-based snippet.

The DR cache, which aims at reducing the number of disk accesses to the documents, is accessed by $docID$, while query q is used to select the best sentences from the document associated with $docID$. In literature, there are two main cache organizations:

- $\mathsf{DRCache_{doc}}$: each cache entry contains an integral copy of a document;
- $\mathsf{DRCache_{surr}}$: each entry stores a surrogate of a document, which includes its most relevant sentences precomputed offline;

The $\mathsf{DRCache_{doc}}$ works as a sort of OS buffer cache, speeding up the access to the most popular documents present on disk. Once the full document is retrieved a snippetting algorithm may easily generate a proper summary for the given query. However, the document size is significantly larger than the required snippet. Indeed, as shown in the previous section, for most documents only a single snippet is generated, and for the most popular documents, less than 10 snippets are sufficient to answer any query. Therefore, most of the cache memory is actually wasted.

$\mathsf{DRCache_{surr}}$ tries to increase the number of entries cached in a given amount of memory by exploiting document surrogates which are shorter of the original document still retaining most of the informative content. Document surrogates are generated offline for each document, and are accessed and stored in $\mathsf{DRCache_{surr}}$ in place of the original documents. The size of the surrogates induces a trade-off between cache hit-ratio and snippet quality. On the one hand, having small surrogates allow to cache more of them, thus allowing to increase the hit ratio. On the other, surrogates have to be sufficiently large in order to produce high quality snippets for all the possible queries retrieving a document.

This paper proposes a novel DR cache, namely $\mathsf{DRCache_{Ssnip}}$, whose entries are particular document surrogates, called *supernippet*, whose construction and management are based on the past usage of the WSE by the user community. The idea behind $\mathsf{DRCache_{Ssnip}}$ is to exploit the past queries submitted to the WSE in order to produce very concise document summaries, containing only those document sentences being actually useful in generating snippets. Also, the size of each summary may be very small when only a few sentences were needed to generate the snippets for the past queries, or it may be larger when there are many useful sentences, and thus several topics of interest for users, in each document. We prove that this cache organization allows for high hit ratio and high-quality query-biased snippets.

4. DR CACHE MANAGEMENT

As discussed in Section 3 and illustrated in Figure 5, the DR can host different types of caches, either $\mathsf{DRCache_{doc}}$, $\mathsf{DRCache_{surr}}$, or $\mathsf{DRCache_{Ssnip}}$. To discuss the various cache organizations, it is worth introducing some notation, summarized in Table 2.

Let D be the set of all the crawled documents stored in the repository. Each $d \in D$ is composed of a set of sentences, i.e. $d = \{s_1, s_2, \cdots, s_n\}$.

We denote by S_d, $S_{d,q}$, and $SS_{d,Q}$, the surrogate, snippet, and supersnippet of a document $d \in D$. The generation of these summaries from d actually consists in selecting a suitable subset of its passages/sentences according to some relevance criterium. Note that the notation used recalls that a surrogate S_d is a query-independent excerpt of d, while a snippet $S_{d,q}$ depends on the query q. Finally, the content of a supersnippet $SS_{d,Q}$, $SS_{d,Q} \subseteq d$, depends on the set Q of past queries submitted to the WSE which retrieved document d.

Symbol	Meaning
D	The collection of the crawled documents
d	A generic document ($d \in D$) associated with a distinct *DocID* and a distinct URL, composed of a set of sentences s_i, i.e., $d = \{s_1, s_2, \cdots, s_n\}$
s	A sentence composed of several terms
t	A term included in a sentence s
S_d	A surrogate that summarizes document d, and includes its most important sentences
$S_{d,q}$	The query-biased snippet of document d, which includes the sentences of d that are the most relevant to q
$SS_{d,Q}$	The supersnippet of document d, computed over the set Q of past queries
$I(s)$	The informative content of a sentence s
$R(s,q)$	The relevance of sentence s to query q

Table 2: Table of Notation

The lookup keys of all the three caches $\mathsf{DRCache_{doc}}$, $\mathsf{DRCache_{surr}}$, and $\mathsf{DRCache_{Ssnip}}$ are the unique identifiers of the various documents (docIDs). Given a key docID associated to document d, then the possible contents of the entries of caches $\mathsf{DRCache_{doc}}$, $\mathsf{DRCache_{surr}}$, and $\mathsf{DRCache_{Ssnip}}$ are, respectively, d, S_d, and $SS_{d,Q}$.

About the memory hierarchy of DR, below the cache level, which is located in *main memory*, we have the *disk*-stored document repository, which keeps the collection of all the indexed documents D and the associated URLs. Both the integral documents [1] and the URLS are directly accessible by supplying the corresponding docIDs.

Figure 5 shows these disk-based levels of memory hierarchy in DR.

4.1 Surrogates, snippets, and supersnippets

In this section we discuss how the best sentences to include into S_d, $S_{d,q}$, and $SS_{d,Q}$ can be devised. For the sake of simplicity hereinafter we will assume surrogates, snippets, and supersnippets composed by sets of sentences. The definitions and metrics discussed can be however trivially adapted to passages or any other piece of text of fixed or variable size.

Surrogates.

Several proposals appeared in the literature regarding the generation of concise surrogates retaining most of the informative content of a document. In Section 7 we survey some of these methods. They are generally based on a function to evaluate the informative content $I(s)$ of each sentence $s \in d$, and on a constraint on the size of the surrogate, e.g. the surrogate has to contain a fraction x, $0 < x < 1$, of all the sentences of the original document. Given $I(s)$ and x, the best surrogate S_d for d is the one that maximizes:

[1] In case DR hosts a $\mathsf{DRCache_{surr}}$ cache, below the cache level we have surrogates S_d of all the documents, which are statically generated from each $d \in D$.

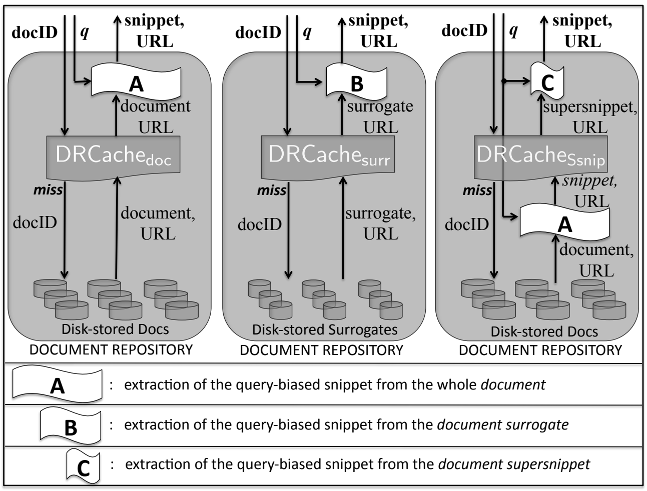

Figure 5: Three possible DR organizations, hosting either DRCache$_{doc}$, DRCache$_{surr}$, or DRCache$_{Ssnip}$. Note the activities labelled A, B, and C, whose size models the associated computational load, which in turn depends on the size of the input: either documents d, surrogates S_d, or supersnippets $SS_{d,Q}$.

$$S_d = \arg\max_{\sigma \subseteq d} \sum_{s \in \sigma} I(s), \quad \text{where } |\sigma| \leq \lfloor x \cdot |d| \rfloor$$

Query-biased Snippets.

There exists a vast literature on the methods to generate query biased snippets, for which we invite interested readers to refer to Section 7.

Query biased snippets $S_{d,q}$ are generated for all the top-k documents retrieved by a WSE for query q, by means of a function $R(s, q)$, which measure the relevance to q of the various sentences $s \in d$. Hence, given function $R(s, q)$ and the maximum size sz of a snippet, expressed in terms of the number of sentences, the best snippet $S_{d,q}$ for a query q and a document d is the one that maximizes:

$$S_{d,q} = \arg\max_{\sigma \subseteq d} \sum_{s \in \sigma} R(s, q), \quad \text{where } |\sigma| \leq sz \qquad (1)$$

In our experiments, a query-based snippet, extracted from either a document d or a surrogates/supersnippet of d, is simply obtained by applying Equation (1) with a relevance score computed as follows:

$$R(s, q) = \frac{|\overline{s} \cap \overline{q}|^2}{|\overline{q}|}$$

where $\overline{s}$ and $\overline{q}$ are the bags of terms occurring in s and q. This corresponds to ranking all the sentences and selecting the top-k ranked sentences. In our experiments, we set $k = 3$.

Supersnippets.

Given a set Q of queries submitted in the past to the WSE, e.g., the set of queries recorded in a query log, let us consider the set Q_d, $Q_d \subseteq Q$ of past queries having document d in the result set returned to the user. Set Q_d contains all the past queries for which document d was considered relevant by the WSE ranking function. By using a snippeting technique as the one sketched above, the set $U_d = \bigcup_{q \in Q_d} S_{d,q}$ can be easily built. While a document d can contain several different sentences, the above set only contains those sentences of d that are likely to be of some interest to past WSE users. U_d can be thus considered a surrogate of d, whose construction, and thus the relative criterium for sentence selection, is *usage* based. By analyzing a query log spanning a long time period, we have shown that the number of different snippets for a given document is in most cases very small. Thus, we can expect also the size U_d to be small. Moreover, some redundancies can be found in U_d because some sentences are very similar to others.

Given a similarity score $sim(s, s')$ between sentences s and s' of a document, we define the *supersnippet* $SS_{d,Q}$ as the set of *medoids* [7] obtained by a k-medoids clustering, run on the set of sentences contained in U_d that minimizes inter-cluster similarities and maximizes intra-cluster similarities. The number of clusters, i.e. k, itself is the optimal one and it is not a-priori fixed.

A careful reader may note some resemblance of the term “supersnippet” to “superstring”. In the “superstring problem”, given a input set of symbol strings, the goal is to find the shortest string that includes as sub-strings all the others. Our supersnippet is made up of several strings, but, in a similar way as superstring, we aim to reduce its size by eliminating redundant sentences.

The above definition of a supersnippet cannot be used in practice. The main reason is that we need an efficient algorithm for constructing them. Therefore, we adopt the following greedy strategy. We read queries in the same order as they arrive. For each query we retrieve the corresponding SERP. For each snippet in the result set, we consider each sentence contained within. In the corresponding supersnippet, we add a sentence only if its similarity with all the previously added ones is below a certain threshold. In our experiments, we used the Jaccard index measured on sentence terms as similarity metric, and set the threshold to 0.8, i.e. less than 80% of the terms in the candidate sentence appear in the sentences already added to the supersnippet.

Furthermore, in our experiments we bound the maximum number of sentences forming a supersnippet. Several approaches can be adopted to limit to sz sentences the maximum size of a supersnippet. A first *static* approach consists in sorting all the sentences of $SS_{d,Q}$ by frequency of occurrence within snippets $S_{d,q}, \forall q \in Q_d$. The first sz sentences can in this case be considered as the most relevant for document d on the basis of their past usage.
Another possibility is to consider a supernippet as a fixed size *bin*. This bin is filled with new sentences occurring in snippets retrieved for the queries in Q_d. When the bin is completely filled, and another sentence has to be inserted, a replacement policy can be used to evict from the supersnippet a sentence to make place for the new one.

In this paper we followed this second possibility and adopted a simple LRU policy to always evict the *least recently used* sentence first. We preferred to adopt a dynamic supersnippet management policy to avoid the aging of statically-generated supersnippets as time progresses and interests of users change. We leave as a future work the analysis of the effect of aging on static supersnippets, as well as the study of the performance of alternative sentence replacement policies. The pseudocode reported in Algorithm 1 sketches the simple management of $\mathsf{DRCache}_{\mathsf{Ssnip}}$. Function *Lookup*$(q, DocID)$ is called to lookup for a supersnippet of d in the cache, and implements the LRU management of $\mathsf{DRCache}_{\mathsf{Ssnip}}$ entries. *Lookup*$(q, DocID)$ calls function *UpdateSS*$(SS_{d,Q}, DocID)$ to update as above described the sentences stored within $SS_{d,Q}$.

Unlike other approaches, where the document surrogates are statically determined, our supersnippets are thus dynamically modified on the basis of their usage. Their contents change over time, but depend on the query-biased snippets extracted in the past from the original documents to answer to user queries. As a consequence of this approach, in our supersnippet cache hosted in the DR, a miss can occur also when the docID key is matched, but the query-biased snippet returned for it results to be of *low quality*. Even in this case a miss penalty must be paid, and a query-biased snippet must be extracted from the integral document stored on disk. This new snippet is then used to update the cached supersnippet on a LRU basis.

Query-biased Snippets: Documents vs. Surrogates.

In Figure 5 we show that when DR hosts either $\mathsf{DRCache}_{\mathsf{surr}}$ or $\mathsf{DRCache}_{\mathsf{Ssnip}}$, a query-biased snippet is extracted from a cache entry that contains a surrogate of d. More specifically, according to Equation (1), we extract the query-biased snipped $S_{d',q}$ from d', where $d' = S_d$ or $d' = SS_{d,Q}$. While S_d is a proper surrogate of d, the supersnippet $SS_{d,Q}$ is a special surrogate whose construction is query-log based.

4.2 Cache management policies

In this section we discuss the three possible organizations of DR illustrated in Figure 5, and the relative cache management issues. The first two caches $\mathsf{DRCache}_{\mathsf{doc}}$ and $\mathsf{DRCache}_{\mathsf{surr}}$ are pretty standard. The docID is used as look-up key. If no cache entry corresponding to the key exists, a miss occurs. Then the lower memory level (disk-based) is accessed to retrieve the associated URL, along with either the document or the surrogate. These caches can be *static*, i.e. the cache content does not change during the cache utilization (although it can be updated by preparing off-line a new image of the cache), or *dynamic*. In the last case, many replacement policies can be used in order to maximize the hit ratio, trying to take the largest advantage possible from information about recency and frequency of references.

Note that for both caches, once retrieved the matching cache entry, the query-biased snippet must be extracted from the content of this entry. The two activities, labeled as **A** and **B**, correspond to snippet extractions. The cost of extracting the snippet from the document surrogate (**B**) is smaller than from the whole document (**A**). In general, both the *hit cost* and the *miss penalty* of $\mathsf{DRCache}_{\mathsf{surr}}$ are smaller than the ones of $\mathsf{DRCache}_{\mathsf{doc}}$.

$\mathsf{DRCache}_{\mathsf{Ssnip}}$ is more complex to manage. As for the other two caches, docIDs are used as look-up keys. If no cache entry corresponding to the key exists, a miss occurs. In this case the miss penalty is more expensive, since the query biased snippet must be first extracted from the document (see activity labeled as **A**), and the copied to a cache entry.

$\mathsf{DRCache}_{\mathsf{Ssnip}}$ needs to exploit a special cache policy, i.e. a special rule, to determine whether a request can be satisfied using the cached supersnippet. More specifically, even when the look-up of a docID succeeds, the query-biased extraction (see activity labeled as **C**) can fail, because the snippet results of *low quality* with respect to the query. In this case we have a *miss of quality*. This leads to accessing to the disk-stored repository to retrieve the whole document. The query biased snippet is thus extracted from the document (see activity labeled as **A**). This query-biased snippet is finally used to update the corresponding cache entry, which contains the document supersnippet.

4.3 Contributions

Our work contains several original contributions. To the best of our knowledge, this is the first proposal of a method to produce supersnippets, i.e., short document surrogates that are generated by exploiting the knowledge collected from the query-biased snippets returned in the past by a WSE. Moreover, we define an abstract WSE architecture model, in which we host different types of DR cache along with a classical query results cache in the FE. This model allows us to study and experiment different cache organizations, and to propose a novel method that guarantees the best snippet generation performance compared with several baselines. The hit rates measured for our *supersnippet cache* result in fact to be remarkably higher than those obtainable with other DR cache organizations. In the experimental evaluation, we considered also the presence of a query results cache that filters out most frequent queries, without asking neither the WSE BE nor the DR. Even in this case the hit rates measured for our supersnippet cache results to be very high. Another important contribution is the methodology to validate our proposal by evaluating also the quality of the snippets extracted by our supersnippets. In fact we employed a real-life query log and the pages of results returned for the logged queries by an actual commercial WSE. The analyses conducted on such query log also motivates the DR caching techniques proposed in this paper.

5. CACHE EFFECTIVENESS

The component **C**, described in the previous section and illustrate in Figure 5, plays an important role for $\mathsf{DRCache}_{\mathsf{Ssnip}}$. For each incoming query, it must be possible to decide whether or not a cached supersnippet contains the sentences needed to produce an high quality snippet. If this is not the case, a miss of quality occurs, and the disk-based document repository is accessed.

To this end, a quality function must be defined, which, given the query q and the supersnippet $SS_{d,q}$ is able to measure the goodness of $SS_{d,q}$. In the following we propose and analyze a simple quality metrics, and we compare the snippets produced by $\mathsf{DRCache}_{\mathsf{Ssnip}}$ with those returned by a commercial WSE. Indeed, we compare the quality of the snippets generated by our caching algorithm, with the quality of the results returned by the Yahoo! WSE. Note that the Yahoo! WSE is also used to simulate the underlying BE and DR, thanks to the property of $\mathsf{DRCache}_{\mathsf{Ssnip}}$ of being independent from the snippet generation engine.

The proposed metrics takes into account the number of terms in common between the query q and the generated snippet $S_{d,q}$, and it is illustrated in the following section. This metrics can be used directly by $\mathsf{DRCache}_{\mathsf{Ssnip}}$ to detect quality misses.

Algorithm 1 DRCache$_{\mathsf{Ssnip}}$ caching algorithm.

```
function Lookup($q$, $DocID$)
    if $DocID \in \mathcal{C}$ then                              ▷ cache hit
        $SS_{d,Q} \leftarrow \mathcal{C}(DocID)$                 ▷ retrieve the supersnippet
        $S_{d,q} \leftarrow$ Snippet($q, SS_{d,Q}$)              ▷ generate a snippet
        if $\neg$HighQuality($S_{d,q}, q$) then                  ▷ quality miss
            $S_{d,q} \leftarrow$UpdateSS($SS_{d,Q}$, $DocID$, $q$)
        end if
        $\mathcal{C}$.MoveToFront($SS_{d,Q}$)                    ▷ LRU update
    else                                                         ▷ cache miss
        $SS_{d,Q} \leftarrow \emptyset$
        $S_{d,q} \leftarrow$UpdateSS($SS_{d,Q}$, $DocID$, $q$)
        $\mathcal{C}$.PopBack()
        $\mathcal{C}$.PushFront($SS_{d,Q}$)                      ▷ add a new supersnippet
    end if
    return $S_{d,q}$                                             ▷ Return the snippet
end function

                     ▷ Update the supersnippet with a new snippet for $q$
function UpdateSS($SS_{d,Q}$, $DocID$,$q$)
    $d \leftarrow$ GetDoc($DocID$)                               ▷ access repository
    $S_{d,q} \leftarrow$ Snippet($q, d$)
    for $s \in S_{d,q}$ do                                       ▷ update the supersnippet
        $s' \leftarrow \arg\max_{t \in SS_{d,Q}} sim(s,t)$
        if $sim(s,s') \geq \tau$ then
            $SS_{d,Q}$.MoveToFront($s'$)                         ▷ LRU update
        else
            $SS_{d,Q}$.PopBack()
            $SS_{d,Q}$.PushFront($s'$)                           ▷ add a new sentence
        end if
    end for
    return $S_{d,q}$
end function
```

5.1 Snippet quality measure

An objective metrics measuring the quality of the snippet $S_{d,q}$ for the query q can take into account the number of terms q occurring in $S_{d,q}$: the more query terms included in the sentences of a snippet, the greater the snippet goodness. More specifically, we adopted the scoring function proposed in [16]:

$$score(S_{d,q},\ q) = \frac{|\overline{S}_{d,q} \cap \overline{q}|^2}{|\overline{q}|} \qquad (2)$$

where $\overline{S}_{d,q}$ and $\overline{q}$ correspond to the sets of terms appearing, respectively, in the sentences of $S_{d,q}$ and in the query q.

We claim that a snippet $S_{d,q}$ is of *high quality* if $q \subseteq S_{d,q}$ for queries with one or two terms, or if $S_{d,q}$ contains at least two query terms for longer queries. It can easily shown that our good quality criteria is met whenever $score(S_{d,q}, q) \geq 1$.

Proposition 1. *Let $\overline{q}$ and $\overline{S}_{d,q}$ be the sets of terms of q and $S_{d,q}$, respectively. If $score(S_{d,q},\ q) \geq 1$ then either $|\overline{q}| \leq 2$ and $\overline{q} \subseteq \overline{S}_{d,q}$, or $|\overline{q}| > 2$ and $|\overline{S}_{d,q} \cap \overline{q}| \geq 2$.*

Proof. If $|\overline{q}| = 1$ then $score(S_{d,q},\ q) \geq 1$ trivially implies that $\overline{q} \subseteq \overline{S}_{d,q}$.

If $|\overline{q}| > 1$ then $score(S_{d,q}, q) \geq 1$ means $|\overline{S}_{d,q} \cap \overline{q}|^2/|\overline{q}| \geq 1$. By a simple arithmetical argument we have that $|\overline{S}_{d,q} \cap \overline{q}|^2 \geq |\overline{q}| \geq 2 \Rightarrow |\overline{S}_{d,q} \cap \overline{q}| \geq \sqrt{2} \approx 1.4$. Since $|\overline{S}_{d,q} \cap \overline{q}|$ must be an integer value, then we can conclude that $|\overline{S}_{d,q} \cap \overline{q}| \geq 2$. This also is equivalent to $\overline{q} \subseteq \overline{S}_{d,q}$ for the case $|\overline{q}| = 2$. □

This rough but fast-to-compute score measure is exploited in the implementation of DRCache$_{\mathsf{Ssnip}}$ to evaluate on-the-fly the goodness of the snippets extracted from the document supersnippets stored in the cache. If the score is lower than 1, the cache policy generates a *quality miss.*

We applied this quality score to the snippets generated by the Yahoo! WSE for the top 10 document returned to the *D1* query log. It is worth noting that the fraction of these snippets resulting of good quality – i.e., with score greater or equal to 1 – is high, but remarkably lower than 100%: the high quality Yahoo! snippets resulting generated from the *D1* query log are about 81% only. In fact, the snippets returned by Yahoo! do not always contain the terms of the query. This is not always due to badly answered queries. For example, this may happen when a query matches some metadata (e.g., the URL name) and not the snippet's sentence, or when only a few terms of a long query are actually included in the document retrieved. Even misspells or variations in navigational queries aiming at finding well-known portal websites are typical example of this phenomenon. A deeper study of this issue is however out of the scope of this work.

To compare with DRCache$_{\mathsf{Ssnip}}$, we set up the following experiment. First we created the set of supersnippets for the documents being returned by the system. Each query in the *D1/training* query log was submitted to the Yahoo! WSE, and we collected the returned documents and their snippets. For each document, the corresponding supersnippet was generated by picking the most representative sentences on the basis of the returned snippets as describe in the previous section. We run different experiments by allowing a maximum number of 5 and 10 sentences in each supersnippets. Note that the resulting supersnippets have varying size, which depends on the number of distinct queries related with a document, and with the number of its sentences included in the snippets related with those queries. Therefore, many supersnippets have a number of sentences smaller than the allowed maximum.

Second, the *D1/test* was used to assess the quality of the generated supersnippets. Similarly as above, for each query in *D1/test* we collected the corresponding results, by ignoring those documents for which a supersnippet was not generated. Note that in case of previously unseen documents, the DRCache$_{\mathsf{Ssnip}}$ caching algorithm would retrieve the correct snippet from the document repository, therefore it makes sense to discard those documents for the evaluation of the supersnippets goodness. For each collected document we computed the score of the snippets generated from the corresponding supersnippet, and, in particular, we measured the fraction of them being of high quality.

In Figure 6 we report the fraction of high quality snippets generated by the Yahoo! WSE and by the exploiting supersnippets of maximum size 5 and 10 sentences. Values are averaged over buckets of 100,000 consecutive (according to the query log order) snippets.

The percentage of good snippets generated by Yahoo! is only slightly higher[2]. Of course, the quality measured for the supersnippeting technique increases with the number of sentences included. The score of $score(S_{d,q}, q)$ averaged over all the snippets in *D1/test* is 1.42, 1.39, and 1.38 respectively for the Yahoo! ground truth, and our technique exploiting

[2] The peak measured for the snippets in the fifth bucket plotted in Figure 6 is due to the presence in the dataset of some bursty queries caused by specific events regarding public persons.

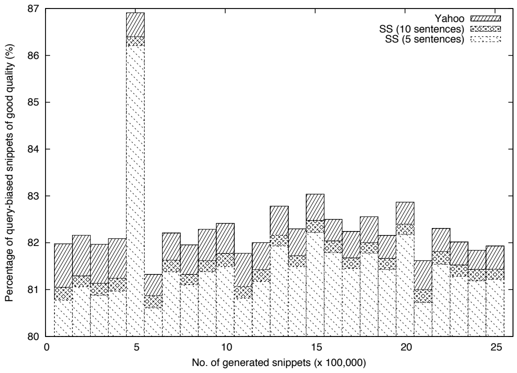

Figure 6: Average fraction of high quality snippets returned by Yahoo! and DRCache$_{\text{Ssnip}}$ measured for consecutive groups of queries in *D1/test.*

supersnippets limited to 10 and 5 sentences. In conclusion, the proposed supersnippeting technique produces snippets of high quality, very close to those generated by the Yahoo! search engine. Indeed, with only 5 sentences at most for each supersnippet, a fraction of 81.5% of the snippets from the whole *D1/test* are of high quality, with a very small loss (0.8%) compared with Yahoo!, where this fraction rises up to 82.3%.

6. HIT RATIO ANALYSIS

In the previous section we have analyzed the quality of snippets generated using our novelapproach. In this section we evaluate the effectiveness of the various DRCache strategies (i.e. DRCache$_{\text{doc}}$, DRCache$_{\text{surr}}$, and DRCache$_{\text{Ssnip}}$) when used in combination with a FECache. Four different DRCache sizes have been experimented: 256, 512, 1024, and 2048MByte. Our experiments are run on the D2 dataset. We warm up the various caches by executing the stream of queries from `D2/training` and we evaluate the various hit ratios using `D2/test`. It is worth recalling that, as discussed in the previous section, a hit in DRCache is when the snippet returned by the cache has $score(S_{d,q},\ q) \geq 1$. On the other hand, if the score is less than one, we say DRCache incurs in a *quality miss.*

The first set of experiments consists in evaluating the hit ratio of DRCache when used in combination of a RCache$_{\text{DocID}}$. Since RCache$_{\text{DocID}}$ only stores the document identifiers of each results list, all the queries generate requests to DR, and the size (better to say, the hit ratio) of the FECache does not affect the performance of DRCache. Table 3 shows hit ratio results for five different DRCache organizations. DRCache$_{\text{doc}}$, DRCache$_{\text{surr}}$ (*doc*, and *surr* in the table), and DRCache$_{\text{Ssnip}}$. The latter is tested by using three different *maximum* supersnippet sizes (i.e. the *maximum* number of sentences forming the supersnippet), namely 5, 10, and 15 (i.e. ss_5, ss_{10}, and ss_{15}) sentences. The tests regarding DRCache$_{\text{doc}}$ were done by considering all the documents as having a size equal to the average size of the documents retrieved by Yahoo! for all the queries in dataset D1. The size of the surrogates used for testing DRCache$_{\text{surr}}$ were instead fixed to be half of the corresponding document [17]. Both DRCache$_{\text{doc}}$ and DRCache$_{\text{surr}}$ are managed using a LRU policy.

The first observation is that in all the cases, hit ratios increase linearly when cache size increases. Also in this case, the linear trend in the hit ratio growth is confirmed. Indeed, the growth cannot be indefinite as the maximum hit ratio attainable is bounded from above by the number of distinct snippets (considering their quality, too) that can be generated by SERPs for queries in the stream.

In terms of hit ratio, DRCache$_{\text{Ssnip}}$ outperforms DRCache$_{\text{doc}}$, and DRCache$_{\text{surr}}$ independently of the size of the cache. On the other hand, we can note that the maximal size of supersnippets does not affect sensitively DRCache$_{\text{Ssnip}}$ hit ratio.

DRCache	Size (in MB)	Hit Ratio DRCache
doc	256M	0.38
	512M	0.41
	1024M	0.44
	2048M	0.49
surr	256M	0.41
	512M	0.453
	1024M	0.49
	2048M	0.53
ss_5	256M	0.42
	512M	0.463
	1024M	0.51
	2048M	0.554
ss_{10}	256M	0.416
	512M	0.462
	1024M	0.51
	2048M	0.55
ss_{15}	256M	0.413
	512M	0.461
	1024M	0.51
	2048M	0.55

Table 3: Hit ratios of DRCache (with DRCache$_{\text{doc}}$, DRCache$_{\text{surr}}$, and DRCache$_{\text{Ssnip}}$ strategies) and RCache$_{\text{DocID}}$.

An important observation, is that we have a miss in DRCache$_{\text{Ssnip}}$ when either the requested document is not present in cache or when the quality score is too small (i.e. less than one). Not necessarily this happens when a "good" snippet cannot be retrieved from DRCache$_{\text{Ssnip}}$. In real cases, for instance, this might be due to misspelled queries that if automatically corrected before being submitted to the underlying back end, would result in an acceptable score. As an example, consider the query "*gmes*", clearly this query can be easily corrected into "*games*". In fact, the snippet generated for the first result returned by Yahoo! for the query "*gmes*" is:

```
" Games.com has a new free online game everyday.
Play puzzle games, hidden object games, arcade
games and more!  Games.com has exclusive poker,
casino and card games."
```

Since the original query term "*gmes*" is not included into the snippet, its quality score is 0, and even if we generate exactly the same snippet as the one generated by Yahoo!, we would count it as a quality miss. To overcome this issue, when we have quality score less than one, we consider the snippet returned by Yahoo! for the same query-document pair. If even the score of the Yahoo! snippet is low as our, a hit could be counted. Table 4 shows the hit ratio results for the same tests of the previous table when quality misses are counted as just described. In all the considered cases hit ratio values raise by about 8% – 9%. Furthermore, this can be probably considered as a better estimation of real hit ratios that can be attained by our DRCache$_{\mathsf{Ssnip}}$ in a real-world system.

Therefore, we can conclude that in the case of a FE running a RCache$_{\mathsf{DocID}}$, the presence of a DRCache decreases considerably the load on the document server, and that DRCache$_{\mathsf{Ssnip}}$ is the best cache organization among the ones tested.

DRCache	Size (in MB)	Hit Ratio DRCache
ss_5	256M	0.5
	512M	0.55
	1024M	0.59
	2048M	0.63
ss_{10}	256M	0.497
	512M	0.548
	1024M	0.59
	2048M	0.625
ss_{15}	256M	0.493
	512M	0.546
	1024M	0.59
	2048M	0.62

Table 4: Hit ratios of DRCache$_{\mathsf{Ssnip}}$ (RCache$_{\mathsf{DocID}}$ case) when quality-misses are counted by comparing quality of snippets with Yahoo! ones.

The next experiment consists in evaluating the effect of a RCache$_{\mathsf{SERP}}$ as a filter for requests arriving to DRCache. A RCache$_{\mathsf{SERP}}$ cache, in fact, stores the whole SERP including the generated snippets. In case of a RCache$_{\mathsf{SERP}}$ cache hit, then, we do not have to request those snippets to the DRCache, thus reducing considerably the load on the DRCache. On the other hand, since frequently requested (i.e. popular) snippets are already built and stored into the RCache$_{\mathsf{SERP}}$ cache, we expect DRCache's hit ratio to be lower than the one obtained when used in combination with RCache$_{\mathsf{DocID}}$.

Tables 5 and 6 show hit ratios for various configurations of DRCache. As in the above experiment, Table 6 reports hit ratios when quality-misses are counted by using the comparison between snippets generated from the supersnippets and Yahoo! ones.

As expected, in all the cases hit ratios are lower than in the previous cases. The differences in the hit ratios obtained by DRCache$_{\mathsf{Ssnip}}$ with respect to DRCache$_{\mathsf{doc}}$, and DRCache$_{\mathsf{surr}}$ instead increase remarkably. This prove that our supersnippets are very flexible and can provide effective snippets also for less popular queries. Moreover, if we sum up the hit ratios occurring on both FECache and DRCache$_{\mathsf{Ssnip}}$, we obtain a impressive *cumulative* hit ratio of about 62%. Note that this is an upper bound to the real cumulative hit ratio. Indeed, FECache stores duplicate snippets (due to possible shared snippets among SERPs), therefore the actual cumulative hit ratio may be slightly lower. We can observe that also in this set of experiments, the maximum number of sentences stored in the supersnippet does not influence heavily the DRCache$_{\mathsf{Ssnip}}$ hit ratio.

7. RELATED WORK

Our proposal for an efficient and effective caching system for document snippets is related to two different research topics. The first is concerned with text summarization, i.e. the production of possibly concise document surrogates, which contain sets of sentences/words that preserve to some extent the meaning of the original documents from which they are generated. This document summaries can be independent of the submitted queries, or can be query-biase. Another related topic deals with the opportunities for exploiting caching at various levels to improve throughput, also exploiting WSE log analysis.

FECache #Entries (Hit Ratio)	DRCache	Size (in MB)	Hit Ratio DRCache
128K (0.36)	*doc*	256M	0.087
		512M	0.11
		1024M	0.14
		2048M	0.19
	surr	256M	0.11
		512M	0.148
		1024M	0.19
		2048M	0.25
	ss_5	256M	0.15
		512M	0.217
		1024M	0.29
		2048M	0.36
	ss_{10}	256M	0.141
		512M	0.21
		1024M	0.287
		2048M	0.35
	ss_{15}	256M	0.14
		512M	0.207
		1024M	0.287
		2048M	0.35
256K (0.38)	*doc*	256M	0.083
		512M	0.1
		1024M	0.134
		2048M	0.17
	surr	256M	0.104
		512M	0.135
		1024M	0.177
		2048M	0.23
	ss_5	256M	0.139
		512M	0.194
		1024M	0.266
		2048M	0.34
	ss_{10}	256M	0.13
		512M	0.189
		1024M	0.266
		2048M	0.334
	ss_{15}	256M	0.12
		512M	0.18
		1024M	0.266
		2048M	0.33

Table 5: Hit ratios of DRCache (with DRCache$_{\mathsf{doc}}$, DRCache$_{\mathsf{surr}}$, and DRCache$_{\mathsf{Ssnip}}$ strategies) and RCache$_{\mathsf{SERP}}$.

Text summarization techniques. The techniques to generate good surrogates S_d of a document d (see Table 2) are those for summarizing text, and are based on the function $I(s)$ that estimates the document's information contained in each sentence $s \in d$.

There are several ways to compute $I(s)$:

- **Luhn Method**: One of the first approach for text summarization was proposed by Luhn [10]. The method exploits the term frequencies tf in order to assign a weight to each term $t \in s$. A term t is considered as significant if tf overcome a threshold T, whose value depends on the number of sentences of d.

  According to Lu and Callan [9], a sentence s is assigned a score $I(s)$ that depends on the so-called *clusters*, which are particular sequences of terms included

FECache #Entries (Hit Ratio)	DRCache	Size (in MB)	Hit Ratio DRCache
128K (0.36)	ss_5	256M	0.21
		512M	0.29
		1024M	0.36
		2048M	0.42
	ss_{10}	256M	0.2
		512M	0.28
		1024M	0.356
		2048M	0.419
	ss_{15}	256M	0.19
		512M	0.27
		1024M	0.355
		2048M	0.419
256K (0.38)	ss_5	256M	0.19
		512M	0.258
		1024M	0.336
		2048M	0.4
	ss_{10}	256M	0.18
		512M	0.251
		1024M	0.334
		2048M	0.398
	ss_{15}	256M	0.18
		512M	0.249
		1024M	0.332
		2048M	0.395

Table 6: Hit ratios of DRCache$_{\mathsf{Ssnip}}$ (RCache$_{\mathsf{SERP}}$ case) when quality-misses are counted by comparing quality of snippets with Yahoo! ones.

in s. A cluster starts and ends with a significant term, and does not include long subsequences of insignificant terms. A cluster is scored on the basis of the ratio between the significant and the total number of terms included in the cluster. Finally, the importance of sentence s, i.e. its informative content $I(s)$, is given by the maximum score of all the cluster in s.

- **TF-IDF Method**: Tsegay *et al.* [17] adopted a classical TF-IDF (Term Frequency Inverse Document Frequency) method to score the various sentences s in d. We can consider this method as an extension of the Luhn one, which only exploits TF. Unfortunately, this technique needs a corpus for computing IDF score.

  The formula adopted is the following:

  $$I(s) = \sum_{t \in s-stopwords} (\log\ tf + 1) \times \log \frac{N}{df}$$

  where tf is the raw count of a term t in the document d where $s \in d$, $N = |D|$ is the number of documents in the collection, while df is the count of documents that contain term t.

- **Position in text**: Edmundson [4] proposed another weight-based method that combines the frequency with other heuristics like the position (usually, first sentences are a natural summary of a document), or the formatting (the titles contain meaningful sentences).

Different approaches based on machine learning and many other techniques have been proposed [14]. Recently an effective method, based on a specific function learned from a training data, and defined in terms of multiple features extracted from s, have been devised [12].

The *query biased snippet* $S_{d,q}$ are a particular form of text summarization, since the selected sentence of d not only depend on the sentence relevance, but also on a query q. Tombros and Sanderson [16], each sentence has the following score:

$$s\_rel(s,q) = \frac{|s \cap q|^2}{|q|}$$

where $|s \cap q|$ is the number of query terms in the sentence s and $|q|$ the number of query terms: the more query terms a sentence contains, the higher is his score. The relevance of a sentence becomes a function $R(s,q)$ biased on query q:

$$R(s,q) = k_1\ s\_rel(s,q) + k_2\ I(s)$$

where k_1, k_2 are arbitrary weigh parameters.

WSE caching techniques. Query logs record historical usage information, and are a precious mine of information, from which we can extract knowledge to be exploited for a lot of different purposes [15]. The analysis of common usage patterns to optimize system performance by means of caching techniques is one of the most important uses of such source of information.

The good efficiency of caching techniques in WSE implementation is motivated by the inverse power law distribution of query topics searched for [19, 15]. This high level of sharing justifies the adoption of a caching system for Web search engines, and several studies analyzed the design and the management of such server-side caches, and reported about their performance [11, 8, 5, 2].

The SERPs returned for frequently submitted queries are cached on the WSE FE (see Figure 4 to improve responsiveness and throughput, while index entries of terms commonly occurring in user queries are cached on BE to make faster query processing. Even partial results computed for popular sub-queries can be cached to further enhance performance.

Fagni *et al.* showed that combining static and dynamic caching policies together with an adaptive prefetching policy achieves even a higher hit ratio [5]. In their experiments, they observe that devoting a large fraction of entries to static caching along with prefetching obtains the best hit ratio. They also showed the impact of having a static portion of the cache on a multithreaded caching system. Through a simulation of the caching operations they showed that, due to the lower contention, the throughput of the caching system can be doubled by statically fixing a half of the cache entries. This behavior was confirmed also in [2] were the impact of different approaches, such as static vs. dynamic caching, and caching query results vs. caching posting lists was studied.

Caching techniques can also be exploited by BE, by keeping in cache the uncompressed postings lists (or portions of them) associated with terms that are frequently and/or recently used. For a survey about these techniques, and the tradeoff between caching policies adopted by the FE and/or BE components of a WSE [2].

With regard to DR caching, which was investigated in depth in this paper, we can mention the idea of the Snippet Engine [17, 18], in which the original DR documents are replaced with their surrogates. This improves DR caching, since a surrogate takes up less space than the original one (size can range from 20% to 60% of the original document). Moreover, producing the snippets from a surrogate is faster than from the original document, because we have less sentences to compare with the query.

Turpin *et al.* [18] additionally compressed the surrogate with a semi-static compression model, thus obtaining significant improvement in the performances. Moreover, Tsegay *et al.* [17] proved that query-biased snippets built from surrogate are, in most cases, identical to those built from the whole document. They also provided an approach called *simple go-back* (SGB): if a surrogate does not contain all the terms of the query, SGB builds the snippet from the original disk-stored document.

8. CONCLUSIONS AND FUTURE WORK

We have presented a novel technique for scaling up search engine performance by means of a novel caching strategy specifically designed for document snippets. Design choices of our novel DR cache are motivated by the analysis of a real-world query log that allowed us to better understand the characteristics and the popularity distribution of URLs, documents and snippets returned by a WSE. Our $\mathsf{DRCache}_{\mathsf{Ssnip}}$ stores in its entries the supersnippets that are generated by exploiting the knowledge collected from the query-biased snippets returned in the past by a WSE. $\mathsf{DRCache}_{\mathsf{Ssnip}}$ enables the construction of effective snippets (having an average quality very close to that measured on our ground truth) for already processed query/docID pairs and, more importantly, the "*in-cache*" generation of snippets also for query/docID pairs not previously seen. A deep experimentation was conducted using a large real-world query log by varying the size and the organization of the caches present in an abstract WSE architecture model. The hit ratios measured for $\mathsf{DRCache}_{\mathsf{Ssnip}}$ result to be remarkably higher than those obtainable with other DR cache organizations. In particular a hit-ratio of 62% was measured using a $\mathsf{DRCache}_{\mathsf{Ssnip}}$ cache of 2,048MB and a docID result cache in th FE. In the experimental evaluation, we considered also the presence of a SERP cache on the FE that filters out most frequent queries, without asking neither the WSE BE nor the DR. Even in this case the hit rates measured for our supersnippet-based cache results to be very high (up to 42%), with a very large difference over the other cache organizations tested.

To the best of our knowledge, this is the first work discussing a cache designed to relieve the load from document repository by exploiting the knowledge about the past queries submitted. For this reason, several research directions remain open. One of the most important open questions is related to how the document cache and the result cache interact each other. What is the best combination of them. What is the best placement option we can choose: is it better to keep them separated on different machines (thus allowing the exploitation of more aggregate memory), or to reduce the exploitation of network is it better to keep them on the same machine? Another interesting question regards more strictly our supersnippet organization: preliminary tests showed that dynamic supersnippets as the ones discussed in this paper outperform static ones. An open question is the evaluation of the cost/performance ratio between these two different organizations, and the analysis of possible aging effects over statically built supersnippets. One may also investigate the effect of combining text compression and supersnippetting to allow the cache to store more surrogates. We plan to address the previous questions in our future work.

Acknowledgements This work was supported by the EU-FP7-250527 (Assets) project.

9. REFERENCES

[1] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 6–20. IEEE, 2007.

[2] R. Baeza-Yates, A. Gionis, F.P. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *ACM Transactions on the Web (TWEB)*, 2(4):1–28, 2008.

[3] L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22 – 28, mar. 2003.

[4] H P Edmundson. New Methods in Automatic Extracting. *Computing*, 16(2):264–285, 1969.

[5] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[6] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proc. of the 18th Int. Conference on World Wide Web*, pages 431–440. ACM, 2009.

[7] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data An Introduction to Cluster Analysis.* Wiley Interscience, New York, 1990.

[8] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th international conference on World Wide Web*, pages 19–28. ACM, 2003.

[9] J. Lu and J. Callan. Pruning long documents for distributed information retrieval. In *Proceedings of the eleventh international conference on information and knowledge management*, pages 332–339. ACM, 2002.

[10] H.P. Luhn. The automatic creation of literature abstracts. *IBM J. of research and development*, 2(2):159–165, 1958.

[11] E.P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[12] D. Metzler and T. Kanungo. Machine learned sentence selection strategies for query-biased summarization. *Learning to Rank for Information Retrieval*, 40, 2008.

[13] D.A. Patterson and J.L. Hennessy. *Computer organization and design: the hardware/software interface.* Morgan Kaufmann Pub, 2009.

[14] D.R. Radev, E. Hovy, and K. McKeown. Introduction to the special issue on summarization. *Computational Linguistics*, 28(4):399–408, 2002.

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

[16] A. Tombros and M. Sanderson. . In *Proc. of the 21st Annual Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval.* ACM, 1998.

[17] Y. Tsegay, S. Puglisi, A. Turpin, and J. Zobel. Document Compaction for Efficient Query Biased Snippet Generation. *Advances in Information Retrieval*, pages 509–520, 2009.

[18] A. Turpin, Y. Tsegay, D. Hawking, and H. E. Williams. Fast generation of result snippets in web search. *In: ACM SIGIR*, 39(5):127–134, 2007.

[19] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings of IEEE INFOCOM 2002, The 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002.