Using Evolutive Summary Counters for Efficient Cooperative Caching in Search Engines (Supplementary material)

David Dominguez-Sal, Josep Aguilar-Saborit, Mihai Surdeanu, Josep Lluis Larriba-Pey

This document is organized as follows. Appendix A describes a model that calculates the cost to disseminate the Evolutive Summary Counters described in Section IV-A. Appendices B and C detail the pseudocode of ESC-placement and ESC-search described in Section IV-B and IV-C, respectively. Appendix D gives an example of the ESC-search algorithm. In Appendix E, we compare the network bandwidth spent by ESC-search (including the diffusion of the ESC-summaries) and a broadcast protocol. Then, we describe the QA system implemented in our experiments in Appendix F and the parameterization of the distributed cache in G. Finally, we report additional experiments performed for ESC-placement and ESC-search in in Appendix H and I.

Appendix A
ESC scalability

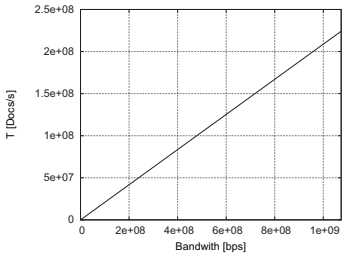


Fig. 1. ESC-summary communication cost.

The main overhead of the ESC architecture described in this paper is the diffusion of the ESC-summaries through the network. The ESC record the documents accessed by the system, and hence the size of the ESC depends directly on the total number of different documents accessed by a node (D) and the average frequency of access to the documents (fr).

The size of a ESC-summary can be computed as the size of the counters, multiplied by the number of distinct entries. Since the counters are codified in binary and adapt the bits per entry to the value stored, the size of the counters is $log_2(fr+1)$. The size of a bloom filter ($BFSize$) is a function of the probability of false positive (FP), which can be derived from [1]:

$$|BF| = \frac{D \cdot ln\,(FP)}{ln(2) \cdot ln(1/2)}.$$

In a distributed system of N computing nodes, the total bits transmitted is the size of the ESC-summaries multiplied by the number of nodes:

$$\begin{aligned} NetBw &= |ESC - Summary| \cdot N \\ &= \frac{D \cdot ln\,(FP)}{ln(2) \cdot ln(1/2)} \cdot log_2(fr+1) \cdot N \end{aligned} \quad (1)$$

The previous formula enables to quantify the network bandwidth required by the ESC in a real application. We can simplify the previous formula taking into account that a FP equal to 0.1 is accurate enough [2]. Additionally, since we are interested in the worst case consumption, we impose the restriction that all documents accessed are different ($fr = 1$) to obtain the following relation:

$$NetBw = 4.79 \cdot (D \cdot N) = 4.79 \cdot T, \quad (2)$$

where T is the global throughput of the distributed system, measured as the number of different documents accessed. T depicts a linear relation with the network bandwidth as illustrated in Figure 1. We observe that the network bandwidth does not depend directly on the number of computers of the distributed system, but it has a linear relation with the overall throughput of the distributed system. For example, the ESC-summaries for a typical QA system (which accesses less than 1k documents per query on average [3]) running in our gigabit Ethernet setup would support more than 2.2k queries per second (2.2 million documents accessed) with a network usage of less than 1% in of the bandwidth usage.

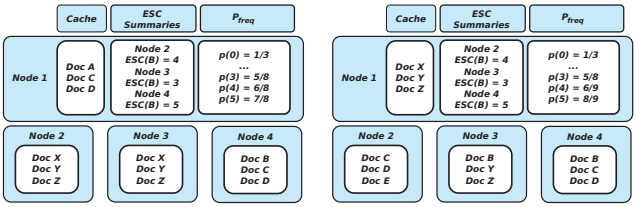
For those environments with very limited network bandwidth, where the broadcasting diffusion mechanism for the ESC-summaries might not be sufficient because of the communication costs, we consider that it is possible to scale the summary communication protocol with the aid of proxies. In this setup, the nodes are divided into teams, and each team designates a node which will act as the team proxy. Only the proxies exchange summaries between teams and the operations in the cooperative cache are tunneled through the proxy of its corresponding team. Besides, since the natural architecture of many applications, such as search engines, typically partition data collections into specialized subcollections [4], [5], the partitioning into teams is easily derived from the subcollection categorization.

Appendix B
ESC-placement pseudo-code

The pseudocode of ESC-placement is listed in Algorithm 1, where the input parameters of the algorithm are a data structure

```
Input: Map<IpAddress, ESCSummary> escMap,
       List<IpAddress> nodesAvailable, Document
       cacheVictim
Output: IpAddress
if (cacheVictim.forwardCounter > MAXIMUM_FORWARDS)
then
    // Do not forward if the cache victim
       exceeded the number of allowed
       forwards
    return NULL;
end
IPAddress mostAccessedAddr := NULL;
int mostAccessedESCValue := -1;
foreach (IpAddress currentAddr : nodesAvailable) do
    // Iterate and find the most accessed
       node
    ESCSummary currentESCSummary :=
    escMap.get(currentAddr);
    int currentESCValue :=
    currentESCSummary.getCount(cacheVictim.id);
    if (currentESCValue > mostAccessedFrequency) then
        mostAccessedAddr := currentAddr;
        mostAccessedESCValue := currentESCValue;
    end
end
return mostAccessed;
```

Algorithm 1: Pseudocode of ESC-placement



(a) (b)

Fig. 2. Example of ESC-Search.

that stores the ESC-summary of each node in the network, the list of all the nodes in the network, and the cache victim (which is least recently used). The algorithm iterates over all the available nodes and finds the node with the largest number of recent accesses in their corresponding ESC-summary.

APPENDIX C
ESC-SEARCH PSEUDOCODE

We show the pseudocode for the ESC-search procedure in Algorithm 2. The input is a data structure that stores the ESC-summary of each node in the network, the list of all the nodes in the network, and the document identifier that has been missed in the local cache. Initially, the nodes are sorted by their individual probability of having the missing document available. Then, the main loop of the algorithm adds more nodes to query until the estimated probability of avoidable miss is below the threshold.

APPENDIX D
ESC-SEARCH EXAMPLE

In this example, we show the steps that ESC-search (configured with $\epsilon = 0.10$) performs to locate documents in a network with four computers. The initial state of the system is depicted in Figure 2(a), in which $N1$ is computing a query that reads "Doc B", but it is not cached locally. The node tries to locate the document in the cooperative cache. First, $N1$ sorts the node list according to the P_{freq}: L=$\{N4, N2, N3\}$, where $N4$ is the node with the highest probability of storing a copy of "Doc B". Then, $N1$ calculates the probability of an avoidable miss when no node is queried:

$$P_{AvMiss}(\{\}, \text{“Doc B”}) = \left[1 - \prod_{i \in N4,N2,N3} \left(1 - P_{freq(ESCS(i,d))}\right)\right] = \left[1 - \frac{1}{8} \cdot \frac{2}{8} \cdot \frac{3}{8}\right] = 0.99.$$

Since $P_{AvMiss}(\{\}, \text{“Doc B”}) > 0.10$ (in other words, it is likely that the document is available in the cooperative cache), $N1$ estimates the probabilities of an avoidable miss if more nodes were queried. The first node that is included is the one which is more likely to store the document, which is $N4$ because "Doc B" has been recently accessed 5 times and its estimated miss probability is the largest:

$$P_{AvMiss}(\{N4\}, \text{“Doc B”}) = \left[\prod_{i \in N4} \left(1 - P_{freq(ESCS(i,d))}\right)\right] \cdot \left[1 - \prod_{i \in N2,N3} \left(1 - P_{freq(ESCS(i,d))}\right)\right] = \left[\frac{1}{8}\right] \cdot \left[1 - \frac{2}{8} \cdot \frac{3}{8}\right] = 0.11.$$

However, the probability is still above 0.10 and it is necessary to extend the search to more nodes:

$$P_{AvMiss}(\{N4, N2\}, \text{“Doc B”}) = \left[\frac{1}{8} \cdot \frac{2}{8}\right] \cdot \left[1 - \frac{3}{8}\right] = 0.02$$

```
Input: Map<IpAddress, ESCSummary> escMap,
       List<IpAddress> nodesAvailable, int documentId
Output: List<IpAddress>
// Sort the list of nodes by the
   probability of having the document
   cached in each node
List<IpAddress> L :=
nodesAvailable.sortByProbability(escMap, documentId);
int s := 0;
double probAvoidableMiss := probAvoidableMiss(escMap,
nodesToQuery);
while (probAvoidableMiss > ϵ ) AND s < L.size() ) do
    // Update the avoidable miss probability
       until we reduce it below ϵ
    // probAvoidableMiss computes P_AvMiss(s, L)
    probAvoidableMiss := probAvoidableMiss(escMap, s, L);
    s++;
end
// List of nodes that ESC-search selects to
   query is the sublist of s nodes.
List<IpAddress> nodesToQuery := L.subList(0, s);
return nodesToQuery;
```

Algorithm 2: Pseudocode of the ESC-search node selection with the dynamic policy.

The algorithm finishes here because it estimates that the probability of an avoidable miss is 0.02 if nodes 2 and 4 are queried, which is sufficient to satisfy our probability miss requirements. Note that up to this point there has been no communication among nodes because the ESC-summaries from the other nodes are already stored in $N1$. In order to retrieve "Doc B", $N1$ sends a request message to nodes 2 and 4 and discovers that the document is only available in $N4$, which transfers it to $N1$. Finally, $N1$ updates the values of P_{freq} according to the final result: (a) it decreases $P_{freq(4)}$ to $\frac{6}{9}$ because the document was not found in $N2$ (where "Doc B" was accessed four times; and (b) it increases $P_{freq(5)}$ to $\frac{8}{9}$ because the document was found in $N4$ (where "Doc B" was accessed five times).

Later, $N1$ computes another query that requests "Doc B", which again is not available locally as depicted in Figure 2(b). The steps are similar to the one described for the previous request but the algorithm finishes after only two steps:

$$P_{AvMiss}(\{\}, \text{"Doc B"}) = \left[1 - \frac{1}{9} \cdot \frac{3}{9} \cdot \frac{3}{8}\right] = 0.99$$

$$P_{AvMiss}(\{Node\ 4\}, \text{"Doc B"}) = \left[\frac{1}{9}\right] \cdot \left[1 - \frac{3}{9} \cdot \frac{3}{8}\right] = 0.09$$

Therefore, ESC-search would only query $N4$ but not $N2$. We observe ESC-search is an algorithm that adapts to the previous experience of the system. Given that in the previous search the document was only available in one node, it updated the corresponding $p_{freq(x)}$ and now it is able to reduce the number of severs queried.

APPENDIX E
SEARCH COMMUNICATION OVERHEAD

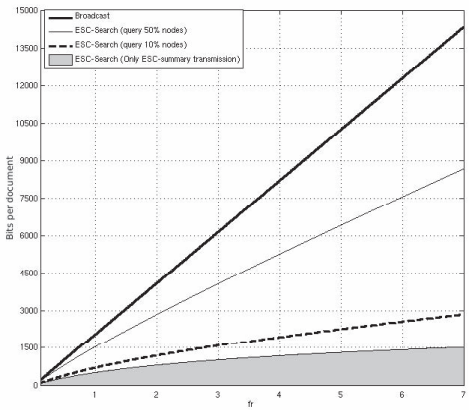


Fig. 3. Parametrization of the search model to calculate the bits sent through network per document. Parameters: N=16, k=5, D=1, FP=0.05, S_k=16bytes.

In this appendix, we model the amount of communication introduced by the ESC-summaries and the ESC-search protocol. We compare the overhead of our proposals to a system that broadcasts all the cooperative cache requests, such as the ICP protocol.

For the broadcast policy, we model the number of bits broadcasted by a node using the following analytical model:

$$BitSend_{Broadcast} = N \cdot (D \cdot fr \cdot S_k)\,,$$

where N is the number of nodes in the system; D is the number of different documents accessed in a window of time; fr is the average number of times that each of those D documents is requested in the same window of time; and S_k is the size of the identifier key for each document.

We model the number of bits sent by the ESC-search policy as:

$$BitSend_{ESC} = h{\cdot}(D \cdot fr \cdot S_k){+}k{\cdot}(|ESC - Summary| \cdot N)\,.$$

The first term in the summation stands for the number of bits involved in the requests. Hence, we substitute N in the broadcast formula by h here, because the request is sent to a subset of nodes. The rest of this first term is equal to the broadcast formula. The second term stands for the number of bits sent due to the broadcast of the ESC-summaries. Here, k is the number of CBF in ESC, and $ESC - Summary$ is the size of the ESC-summaries described in Appendix A.

Parameter D affects equally the two algorithms, thus, in our approach it is not relevant for the comparison. For ESC-search, fr determines the size of the counters of the CBF. In our implementation, we use the Dynamic Count Filters explained in [6]. In DCF, the counters grow logarithmically and dynamically with the size of the value they store.

In Figure 3, we plot our network communication model for different setups. We plot the number of bits sent, as a function of the average number of times that each document is requested in the network, fr. The setups shown are for the broadcast policy, and for an ESC-search configuration which queries between 10% and 50% of the nodes (this includes the requests as well as the broadcast of the ESC-summaries). The solid area of the plot represents the broadcast of the ESC-summaries, which is the same for the two ESC-search scenarios.

Figure 3 confirms that if we increase the average number of requests for the same document, the search based on ESC-summaries is more beneficial than the broadcast policy, even for a considerable number of nodes queried (50%). The number of bits sent due to the requests grows linearly with fr for both scenarios. However, the slope for ESC-search (h) is smaller than for the broadcast policy (N). The number of additional bits sent because of the ESC-summary transmissions does not grow significantly, because the counters grow logarithmically with fr.

The results can be better understood by combining the plots in Figures 4 and 3. According to the experimental results from Figure 4 (where fr can be estimated to be close to 2 for Zipf$_{\alpha=1.0}$, the location recall is over 97.5% when we query 10% of the available nodes. In Figure 3, we see that for this configuration ESC-search achieves a four-fold reduction of the transmissions. If we wish to achieve even higher accuracy (according to Figure 4 more than a 99.7% location recall) it is possible to query 50% of the nodes and still save a 40% of the bandwidth (see Figure 3).

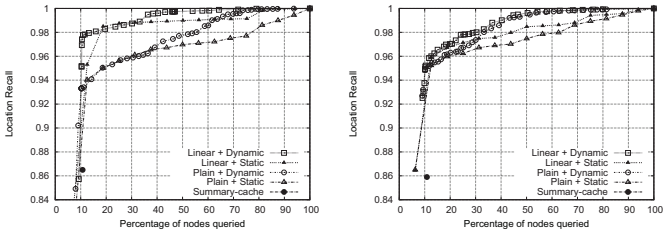


Fig. 4. Location recall for the ESC-search algorithms. The query set sent to the system follows (a) Zipf$_{\alpha=0.59}$ and (b) Zipf$_{\alpha=1.0}$.

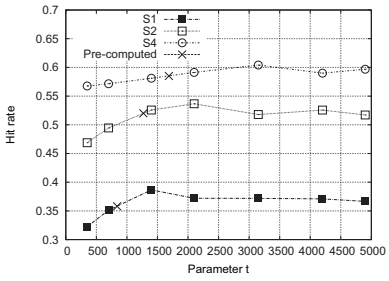


Fig. 5. Hit rate for various values of t and different cache sizes. 'S1' corresponds to a local cache size with enough room to store approximately 8.5% of the documents requested to that node. 'S2' is two times bigger than 'S1', and 'S4' is four times bigger than 'S1'. The question distribution sent follows a Zipf$_{\alpha=0.59}$ and k is fixed to 7.

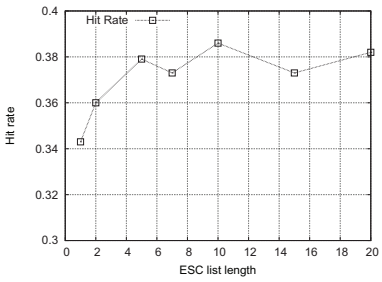


Fig. 6. Hit rate for different values of k with fixed t to 1500. The question distribution sent follows a Zipf$_{\alpha=0.59}$

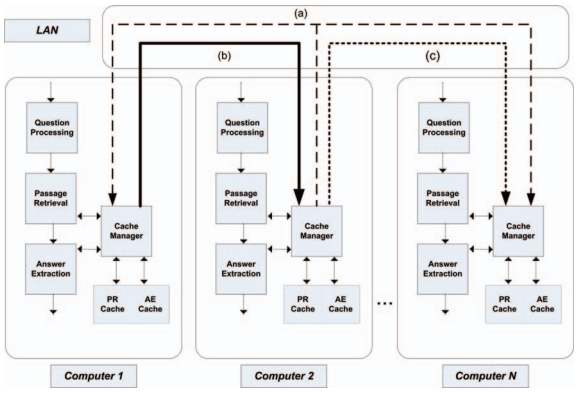


Fig. 7. QA system composed of three sequential blocks QP, PR and AE. PR and AE can request and store data into the cache using the cache manager.

Furthermore, current network technologies only achieve the highest bandwidths with big network packets, and the ESC-summary is a compact structure that can be transferred in a sequence of packets that get the best performance from the network. ESC-search is also more adaptable when there is temporary congestion in the network because the refresh summary rate and the number of count filters (τ and k, respectively) can be adapted. Considering the effect of the parameter changes in ESC (see Appendix G), ESC-search can temporarily delay the refresh of the summaries and/or reduce the number of requests until the peak is over without a large performance drop.

Last but not least, the size of the keys (S_k) also affects the amount of data sent through the network. The size of the request message grows proportionally to S_k. However, since ESC-summaries hash the keys to a count filter, the key size does not affect the size of ESC-summaries. Distributed systems which can handle huge key sizes, such as the 64KB keys supported by Bigtable [7], will have even larger traffic reduction thanks to the decrease in the number of transmissions.

APPENDIX F
QUESTION ANSWERING SYSTEM IMPLEMENTATION

The QA system implemented in this paper (depicted in Figure 7) uses a traditional architecture consisting of three computing blocks linked sequentially [8]: *Question Processing* (QP), *Passage Retrieval* (PR), and *Answer Extraction* (AE). We describe the components next.

Question Processing: the QP component detects the type of the expected answer for each input question. We model this problem as a supervised Machine Learning (ML) classification problem: (a) from each question we extract a rich set of features that include lexical, syntactic, and semantic information, and (b) using a corpus of over 5,000 questions annotated with their expected answer type, we train a multi-class Maximum Entropy classifier to map each question into a known answer type taxonomy [9].

Passage Retrieval: our PR algorithm uses a two-step query relaxation approach: (a) in the first step all non-stop question words are sorted in descending order of their priority using only syntactic information, e.g., a noun is more important than a verb, and (b) in the second step, the set of keywords used for retrieval and their proximity is dynamically adjusted until the number of retrieved passages is sufficient, e.g., if the number of passages is zero or too small we increase the allowable keyword distance or drop the least-significant keyword from the query. The actual information retrieval (IR) step of the query relaxation algorithm is implemented using a Boolean IR system [10] that fetches only passages that contain *all* keywords in the current query. Once the query relaxation step completes, the extracted passages are ranked in descending order of a relevance score computed based on the density and frequency of the keywords in the passage texts [3]. PR concludes with the elimination of the passages whose relevance is below a certain threshold. Thus, the system avoids applying the next CPU-intensive module for passages

that are unlikely to contain a good answer.

Answer Extraction: the AE component extracts candidate answers using an in-house *Named Entity Recognizer* [11], and ranks them based on the lexico-syntactic properties of the context where they appear. We consider as candidate answers all entities of the same type as the answer type detected by the QP component. Candidate answers are ranked using a formula inspired from [12], which calculates for each candidate answer a combination of different heuristics that measure the keywords order and density, such as the distance between keywords or the number of keywords that are found in the same order in the query and in the text. The candidate answers with the highest score are returned to the user as the list of answers.

It is beyond the purpose of this paper to survey qualitative aspects of QA. It is however important to mention that most QA research was motivated by the evaluations organized by the Text REtreival Conference (TREC) [13]. The local QA system used in this paper replicates some of the most successful architectures that were proposed in this context [3], [9], [14]–[16]. We evaluated the quality of the answers from this QA system in a recent international evaluation with results in the state-of-the-art. More details on the system architecture and the evaluation framework are available in [17].

QA systems are excellent solutions for very specific information needs, but the additional functionality provided, i.e., query relaxation and answer extraction, is not computationally cheap: using the set of questions from previous QA evaluations we measured an average response time of 75 seconds/question. The fully fledged factoid QA system[1] used in this paper distributed its execution time as follows: 1.2% QP, 28.8% PR, and 70.0% AE. This brief analysis indicates that QA has two components with large resource requirements: PR, which needs significant I/O to extract passage content from the underlying document collection, and AE, which is CPU-intensive in the extraction and ranking of candidate answers.

From a data processing perspective, our QA system implements a two layer architecture: first, we extract the relevant content from documents that are lexically close to the input question, and second, we semantically analyze this content to extract and rank short textual answers to this question, e.g., named entities such as person, organization, or location names. Because both these blocks are resource intensive, the former in disk accesses and the latter in CPU usage, we allocate a local pool of memory to each stage that is managed with an LRU policy. For both caches, the cache unit is a full document, which is never partitioned in our testbed. Each memory pool stores the data for a fixed number of documents, and each document is retrieved from the caches by its corresponding document identifier. Cached entries in PR store the raw document, and cached entries in AE store the raw document plus its natural language analysis. Since the data for PR is a subset of the data for AE, the information associated to a document is only in one of the cache pools. Once a document is accessed in PR or AE, it is promoted to the corresponding pool for PR or AE, respectively, removing the LRU entry of the pool to make room (see [18] for further details). In addition to the local LRU policy, the cache manager implements the cooperative cache operations described in Section IV of the paper (request/response and forward) for each of the two caches. Therefore, the cached data can be transferred from one node to another, and there may exist multiple copies of a document in a given time. In our experiments, we account for the total hit rate of both caches.

[1]A factoid question answering system is able to respond queries whose answer corresponds to a concrete and objective answer such as a person name (Eg: Who discovered the Penicillin?), a date (Eg: When was the Penicillin discovered?) or a location (Eg: Where was born the Penicillin discoverer?). Other types of questions which are not factoid are for example the generation of lists of results (Eg: List me medicaments that contain Penicillin.) or descriptions (Eg: Why is the Penicillin effective against bacteria?).

APPENDIX G
ESC PARAMETRIZATION

This appendix estimates the parameters necessary to configure ESC. Our objective is to find the adequate values for τ and k, where τ is the refresh time between two CBF updates, and k is the number of CBF in the list. Our parametrization is based on the total monitoring time of the cache window: $t = \tau \cdot k$, which is a derived parameter from the other two.

Discussion of t: This parameter sets the time history that the ESC is monitoring. If this value is too large, entries that have not recently been accessed (in other words, that currently are not in the distributed cache) will be taken into account. If we set t too small, the algorithm will not be aware of the documents that are still in the caches of the remote nodes. In Figure 5, we plot the hit rates for three different cache sizes, and for several values of t fixing k to 7. The selection of t is sensitive to the size of the cache: for each cache size we see that the best value of t changes, and increases with the cache size. The reason is that with bigger caches, the entries can be kept more time in memory and consequently ESC must reflect a longer history.

We set t to the average time that it takes a new entry to be replaced from the cache. This way, ESC monitors the documents which have been accessed in the last $\tau \cdot k$ time units, and that will probably be in the cache of the node. We can simulate the time it takes to obtain v different values (where v is the number of entries that can fit in cache) from a random variable which follows the same distribution as the query requests. We propose to set t to:

$$t = \frac{DiffDocsCacheSize_{(v,\delta)}}{Throughput}. \quad (3)$$

Where $DiffDocsCacheSize_{(v,\delta)}$ is the total number of document requests following a distribution δ to fill a cache of size v, and $Throughput$ is the average number of documents accessed by the system per sec.

We have evaluated Equation (3) for the experiments reported in Figure 5 and marked the values of t with a cross on each of the three plots. The theoretical results are close to the optimal values. Moreover, the experiments were done with an empty cache at the beginning of the execution, so, the experimental optimal hit rate would move even closer to the theoretical prediction if the caches had been warmed up beforehand.

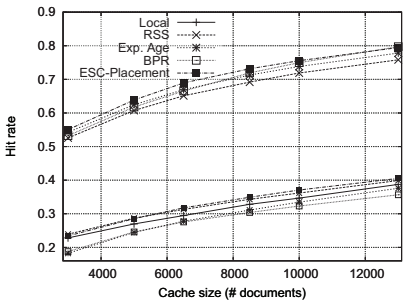


Fig. 8. Hit rate of the cooperative caching algorithms for different cache sizes

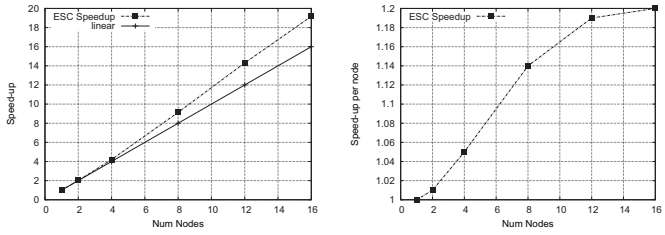
Discussion of k: This parameter determines the number of CBFs that an ESC contains. In other words, for a given t, if we set a large k, then few documents will be monitored during the slice of time assigned to a CBF ($\tau = \frac{t}{k}$), leading to a more accurate summary of that time slot. However, a larger k means that summaries will be generated more frequently, yielding a larger network traffic. It seems that a tradeoff between both situations would be the best solution. In Figure 6, we show the results for an experiment where we plot the hit rate with a fixed t and a variable value of k. We observe that values of k larger or equal to 5 obtain hit rates close to the asymptotic hit rate.

APPENDIX H
ADDITIONAL EXPERIMENTS WITH ESC-PLACEMENT

Experiment 1 (Cache size): In this experiment, we use a query distribution Zipf$_{\alpha=1.0}$, and we test several cache sizes. Figure 8 shows the hit rate of the cooperative caching algorithms. The largest cache size tested, which corresponds to a cache size of 13000 documents, has enough room to store approximately 12% of the requested documents in a node. We plot two lines for each cooperative caching algorithm: we use large points for the global hit rate and smaller ones for the local hits. The figure shows that the improvement of all the cooperative caching algorithms grows logarithmically towards the hit rate obtained with an infinite cache, 0.92. The local cache policy is far below the cooperative caching algorithms for all the tested configurations. Besides, the local cache hit rate improves at a slower rate than the cooperative caching global hit rate.

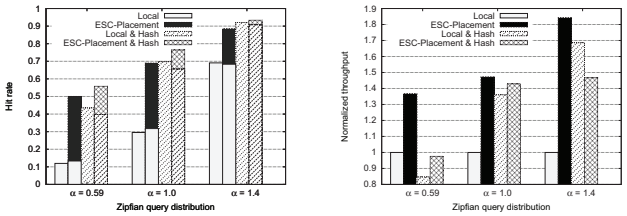
Similarly to the previous experiments, ESC is the best algorithm for all the configurations, and BPR is very close for the configurations with the largest cache size. However, as in the previous algorithm the BPR locality is smaller because BPR is not aware of the contents of the nodes. ESC is the algorithm with the best cache locality, with a relative improvement of up to 30% over other algorithms such as EA or BPR. We observe that for small caches it is more difficult to keep the balance between a good local hit rate and the cooperative cache, and here we observe that the ESC-summaries contain essential information to improve the data placement.

Experiment 2 (Speedup): Figure 9(a) shows the speedup of the QA system as the number of processors grows from one to



(a) (b)

Fig. 9. Speedup of the ESC-placement, compared to a linear speedup (a). Speedup per node added (b). The question distribution follows a Zipf$_{\alpha=0.59}$.



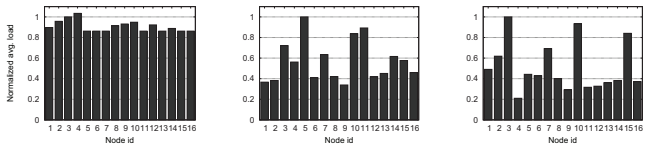
(a) Hit rate (b) Normalized throughput

Fig. 10. Performance comparison with hash based policies.

sixteen computing nodes. The speedup is measured relatively to the system with a local cache in a single computer. The performance gain is above linear (which is shown as a solid line as reference) and increases with the addition of new computers. Note that the speedup increase is above linear due to the Zipf question distribution, which favors repeated questions, and due to our distributed caching algorithm, which efficiently manages these questions. In our experiments, the round robin algorithm balances the load in the cluster reasonably well because the number of queries is not too small, and all the queries were generated following the same probability distribution. So, even with just a local cache, a result not too far from a linear speedup would be expected. The right plot of Figure 9(b) shows the speedup per node for the left plot of the same figure. The benefit rates per added node are: 1.01 for two nodes, 1.14 for eight, 1.19 for twelve, and 1.2 for sixteen nodes. The reason for this increase is that a larger number of computers implies that more cooperative memory is available. Consequently, an effective cooperative caching algorithm can store more documents and increase the hit rates. In the case of 16 nodes, the progression of the benefit ratio (1.19 compared to 1.2) is smaller because we are getting closer asymptotically to the maximum hit rate achievable with an infinite cache.

Experiment 3 (Query locality): Round robin policies guarantee that all nodes compute the same number of queries, which yields a similar load in all nodes. However, round robin does not enforce the query locality because it selects the destination node for the queries independently of the query content. A cache policy that groups all the similar queries (which access to overlapping document subsets) in a single node will intuitively improve the hit rate of the system.

In this experiment, we implement a hash based distribution policy that aims at a better query locality. It computes a hash over the input text of the query, and then applies a modulo operation on the number of nodes, where each different output



(a) (b) (c)

Fig. 11. Detail of the load in each individual node of the cluster for Zipf$_{\alpha=1.0}$=: (a) ESC-placement + Round Robin, (b) ESC-placement + hash, (c) Local + hash.

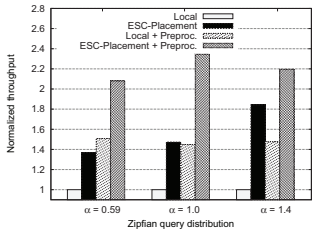


Fig. 12. Normalized throughput for system that has a preprocessed version of the NLP analysis of a QA system.

is assigned to one node. This hash policy aims at a better locality in contrast to the query distribution performed by round robin, which aims at a balanced workload.

Figure 10(b) depicts the hit rate of the hash based policy versus the round robin, and confirms that the hash based policies achieve better hit rates than their corresponding round robin alternatives. The reason is that the initial hash distribution reduces by the number of nodes (16 in this case) the number of different queries that one node can receive, and thus, the cache is more effective because of this more reduced query set. We note that the hash policies are more effective, with respect to the hit rate of their corresponding non-hashed alternative, for the skewed distributions ($\alpha = 1.0$ and 1.4) than for $\alpha = 0.59$. Although both local and ESC-placement improve their hit rate over round robin policies we observe that ESC-placement with hash is still significantly better than the hash policy plus a local cache.

However, the throughput of the hash-based policies is not as good as in our previous experiments with round robin because the hash policies do not take into account the load balance in the cluster. In Figure 10(b), we compare the throughput of the round robin configuration against the hash based policies. We observe that for none of the policies with a hash, we obtain a better throughput than for the previous experiments with ESC-placement.

In Figure 11, we plot the effective usage of each node in the network, which indicates the percentage of time that each node is doing useful computation, i.e. the percentage of time that the node is not idle. The closer to one are the bars, the more balanced is the cluster. Here, we observe a clear indicator of the unbalances in the system with hash policies in Figures 11(b-c), where we find a few nodes above the average load in the cluster. On the other hand, the round robin policy (Figure 11(a)) presents a very balanced workload, though its hit rate, as we previously mentioned, is more limited. Therefore, we find that for an optimal throughput there is a compromise between the data locality and the load balancing of the system. For the interested reader in this tradeoff, we refer to [19] where we propose and analyze load balancing strategies that take into account the cache state of a distributed system in order to maximize the throughput of the system.

Experiment 4 (Collection preprocessing): In our previous experiments, the QA system analyzed the documents with NLP tools during the computation of the AE block on the fly. Another alternative approach, which saves CPU time during the query computation, is to preprocess the whole collection of documents with NLP tools and store the processed documents on disk. Then, the AE module does not need to compute the document analysis, but loads it from secondary storage, unless it is cached in the main memory. In this experiment, we compare our cooperative caching proposals for such an architecture.

In Figure 12, we compare the throughput of our previous system to one that keeps a copy of all the documents preprocessed in disk. We observe that for our hardware setup the throughput increases, if we are able to preprocess the data rather than to analyze it on the fly as we did in the previous experiments. Anyway, we observe that ESC-placement is still faster than the local policy for preprocessed collections because ESC-placement is able to retrieve documents from remote memory, which reduces the traffic from disk. This result indicates that ESC-placement is also adequate in environments where we are able to preprocess the collection beforehand or even implement caches on disk that can be much larger than in memory caches.

The configuration in this experiment obtained the largest throughput among our tests. Nevertheless, we note that although preprocessing might be valid for some scenarios, it may not always be feasible or advisable for QA. Some of the main reasons are the following. (a) Document collections are huge and are always growing. For example, Google recently quantified the number of different web sites crawled as more than one trillion, which makes it prohibitive to preprocess all of them with NLP tools. (b) Many queries appear only once in a real workload of a search engine because of the long tail of the Zipf distribution, and thus, it might not be worth to preprocesses the whole dataset given the amount of unique queries and the cost to process each document (that is significantly more expensive than building current IR inverted indexes). For instance, Baeza et al. found that 50% of the queries received in the Yahoo UK web search engine during a whole year are unique [20]. (c) If the processed data is stored on the disks, then the bottleneck of the system is the I/O from disks in order to supply data for both PR and AE computing blocks. Although in our configuration, (which is able to compute two threads simultaneously) it is faster to preprocess documents and store them on disk, this might not hold in architectures with a larger degree of parallelism. Besides, the current trend in the last years is that the number of CPUs is growing at faster rate than the disks are. For example, a general purpose processor, such as UltraSparc T2, can run 64 threads simultaneously [21], and a GPU, such as Nvidia GeForce GTX 295, has 480 CUDA cores [22]. This trend suggests that reanalyzing documents might be faster (except

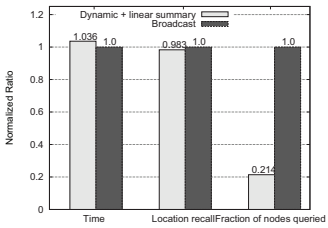


Fig. 13. Execution comparison between broadcast and ESC-search algorithm. The time, the hit rate and the number of queries are normalized with respect to the broadcast search protocol. ϵ is set to 0.05

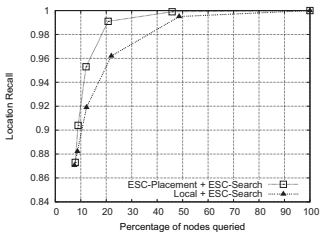


Fig. 14. Comparison between ESC-search and summary caches with ESC-placement activated (a) and no forwarding algorithm (b).

for the very complex processes) than reading preprocessed data (thus, more bytes) from a disk cache.

Appendix I
Additional ESC-search experiments

Experiment 5 (Broadcast comparison): In this experiment, we perform a detailed comparison of the ESC-search algorithm against the broadcast protocol. We note that if the network is not overloaded, the use of broadcasting achieves better performance than any other algorithm because its location recall is 1. Figure 13 plots the execution time of the search algorithm compared to the broadcast baseline. Even if all nodes are interconnected with a fast network that supports broadcast, which is our hardware setup, we see that the execution time of the best ESC-search configuration only increases the broadcast approach by about 3%. However, the probabilistic algorithm only queries approximately a fifth of the total number of nodes. The ESC-search is more scalable because it reduces the stress on the network, and so it is possible to add many more computers than with the broadcast search implementation. Finally, the limited number of remote requests of ESC-search make it a viable candidate if a few computing nodes do not share the same LAN.

Additionally, the configuration of the probabilistic algorithm is easier and more adaptable to the query distribution. In the static approach, the selection of h is a blind choice made by the administrator of the system, and may become a bad choice if the query distribution changes. On the other hand, the probabilistic approach is based on the usage frequencies, which can be extracted on the fly for any query distribution. The selection of ϵ for the probabilistic approach is straightforward and independent from the query log, because it is based on the probability of an avoidable miss.

Experiment 6 (Influence of ESC-placement in ESC-search): In this experiment we compare the location recall of ESC-search when it is combined with ESC-placement versus a system where ESC-search is combined with a placement strategy without forwarding. ESC-search and ESC-placement rely on the same information, provided by the ESC-summaries, hence we study here the correlation between the placement and the location decisions based on ESC.

We observe in Figure 14 that both scenarios share a similar pattern: a steep increase of the slope that converges up to almost a perfect recall when more than one third of the nodes are queried. The location recall for a small number messages is also large enough for most applications: it is over 99% once the system queries approximately 20% of the nodes on average. We also observe that the recall when ESC-placement is activated is better than when no forwarding policy is applied. This happens because ESC-search together with ESC-placement, versus a system where ESC-search is combined with a placement strategy without forwarding, implement complementary policies on top of the same data structures: (a) ESC-placement prefers nodes with more local accesses to the corresponding document, and (b) ESC-search queries first the same nodes with the largest number of accesses to the given document. In other words, ESC-placement tends to forward cache entries to the nodes that will be inspected first by ESC-search.

References

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

[2] D. Dominguez-Sal, "Analysis and optimization of question answering systems," Ph.D. dissertation, Universitat Politecnica de Catalunya, 2010.

[3] D. Moldovan, M. Pasca, S. Harabagiu, and M. Surdeanu, "Performance issues and error analysis in an open-domain question answering system," *ACM Trans. Inf. Syst.*, vol. 21, no. 2, pp. 133–154, 2003.

[4] D. Puppin, F. Silvestri, and D. Laforenza, "Query-driven document partitioning and collection selection," in *Infoscale*, 2006, p. 34.

[5] J. Dean, "Keynote talk: Challenges in building large-scale information retrieval systems," in *WSDM*, 2009.

[6] J. Aguilar-Saborit, P. Trancoso, V. Muntés, and J. Larriba-Pey, "Dynamic count filters," *SIGMOD Record*, vol. 35, no. 1, pp. 26–32, 2006.

[7] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *Trans. Comp. Syst.*, vol. 26, no. 2, 2008.

[8] D. Roussinov, W. Fan, and J. Robles-Flores, "Beyond keywords: automated question answering on the web," *Commun. ACM*, vol. 51, no. 9, pp. 60–65, 2008.

[9] X. Li and D. Roth, "Learning question classifiers: the role of semantic information," *Natural Lang. Eng.*, vol. 12, no. 03, pp. 229–249, 2005.

[10] Lucene, *http://lucene.apache.org/*, Sept. 2008.

[11] M. Surdeanu, J. Turmo, and E. Comelles, "Named Entity Recognition from Spontaneous Open-Domain Speech," in *Interspeech*, 2005, pp. 3433–3436.

[12] M. Pasca, *Open-Domain Question Answering from Large Text Collections*. CSLI Publications Stanford, Calif, 2003.

[13] NIST, "TREC Question Answering Track," *http://trec.nist.gov/data/qamain.html*, 1999-2007.

[14] E. Nyberg, R. Frederking, T. Mitamura, M. Bilotti, K. Hannan, L. Hiyakumoto, J. Ko, F. Lin, L. Lita, and V. Pedro, "Javelin I and II systems at TREC 2005," in *TREC*, 2005.

[15] J. Chu-Carroll, K. Czuba, P. Duboue, and J. Prager, "IBM's Piquant II in TREC2005," in *TREC*, 2005.

[16] D. Moldovan, C. Clark, S. Harabagiu, and D. Hodges, "Cogex: a semantically and contextually enriched logic prover for question answering," *J. Applied Logic*, vol. 5, no. 1, pp. 49–69, 2007.

[17] M. Surdeanu, D. Dominguez-Sal, and P. Comas, "Design and performance analysis of a factoid question answering system for spontaneous speech transcriptions," *Interspeech*, 2006.

[18] D. Dominguez-Sal, J. Larriba-Pey, and M. Surdeanu, "A multi-layer collaborative cache for question answering," in *Euro-Par*, ser. LNCS, vol. 4641. Springer, 2007, pp. 295–306.
[19] D. Dominguez-Sal, M. Surdeanu, J. Aguilar-Saborit, and J. Larriba-Pey, "Cache-aware load balancing for question answering," in *CIKM*, 2008, pp. 1271–1280.
[20] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *SIGIR*. ACM, 2007, pp. 183–190.
[21] S. Microsystems, "UltraSPARC T2 and T2 Plus Processors," *http://www.sun.com/processors/UltraSPARC-T2/specs.xml*, Retrieved on Nov. 2009.
[22] Nvidia, "Geforce GTX 295 Specifications," *http://www.nvidia.com/object/product_geforce_gtx_295_us.html*, Retrieved on Nov. 2009.