Using Evolutive Summary Counters for Efficient Cooperative Caching in Search Engines

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

***Abstract*—We propose and analyze a distributed cooperative caching strategy based on the Evolutive Summary Counters (ESC), a new data structure that stores an approximated record of the data accesses in each computing node of a search engine. The ESC capture the frequency of accesses to the elements of a data collection, and the evolution of the access patterns for each node in a network of computers. The ESC can be efficiently summarized into what we call ESC-summaries to obtain approximate statistics of the document entries accessed by each computing node.**

We use the ESC-summaries to introduce two algorithms that manage our distributed caching strategy, one for the distribution of the cache contents, ESC-placement, and another one for the search of documents in the distributed cache, ESC-search. While the former improves the hit rate of the system and keeps a large ratio of data accesses local, the latter reduces the network traffic by restricting the number of nodes queried to find a document. We show that our cooperative caching approach outperforms state of the art models in both hit rate, throughput, and location recall for multiple scenarios, i.e., different query distributions and systems with varying degrees of complexity.

***Index Terms*—H.3.4-b Distributed systems, Distributed caching, Resource intensive applications, Count Filter**

I. Introduction

Emerging search engines are moving several steps beyond the naive bag-of-words approximation predominant in today's information retrieval models. For example, multimedia search requires a deep analysis of the multimedia content (e.g. music, images, movies), instead of just matching query keywords to the caption of the multimedia object. Or, question answering systems perform deep syntactic and semantic analysis of the document texts in order to provide short, exact answers to natural language questions.

This paper focuses on such next generation engines, which have several things in common: (a) they are all significantly more computationally expensive than current search engines, and (b) they all synthesize the collection of documents to a set of data objects that are interpretable by the machine, e.g., textual passages with deep natural language analysis [1] or multimedia objects with visual/audio features [2]. Since the generation of these data structures is costly, they are good candidates for in-memory caching when their use is frequent. Furthermore, changes to the datasets used by these search engines are generally limited. They typically consist of new document additions, e.g., when new entries of a blog are published, but these changes do not modify previous versions of a document. In other words, once a document is added to the collection, it can be considered read only.

In this paper, we describe a novel cooperative caching strategy for such distributed search engines that is fully implemented on commodity hardware. Our approach manages cache contents according to recent usage information. The foundation of our architecture is a set of local caches (one per system node) that are linked together by a cooperative protocol that provides system wide transparency. Our cooperative caching strategy relies on a new data structure, which we call *Evolutive Summary Counters* (ESC). The ESC keep a record of the recent data accesses of a node. This information is stored in count bloom filters similarly to other proposals such as [3], [4], however, and as we detail further in this document, our proposal applies these compact data structures to record the frequency of access for a given time window, and at the same time maintain its recent history. This information turns to be very valuable in order to improve the placement and location of data in a cooperative cache, because it combines recency and frequency of historical data accesses.

We use the ESC information to deal with two important issues in the cooperative cache management: the *placement* and the *search* of the cached information. The placement algorithm controls where information is cached in the network and is driven by two principles: (a) information should be cached in the nodes where it is most often accessed and (b) information frequently accessed should be cached at least in one node. We propose *ESC-placement*, which is an algorithm that achieves good cache locality by sending documents to nodes that accessed them frequently in a recent time frame, and by avoiding the replication of documents infrequently accessed. As a second issue for the cooperative cache management, we study data search, i.e., how to locate the node that is currently caching a certain data unit. We propose *ESC-search*, which is a search algorithm that estimates the probability of finding a document in a node dynamically, reducing the number of nodes queried. All in all, a single data structure – ESC– provides good performance for both the placement and search of documents in a distributed search engine system.

We summarize the contributions of this paper as: (i) we propose the ESC as a compact data structure to record the recent history of data accesses, (ii) we propose ESC-placement that is an algorithm to distribute the cache contents in a cluster of computers efficiently, (iii) we propose ESC-search, which

The members of DAMA-UPC thank the Ministry of Science and Innovation of Spain and Generalitat de Catalunya, for grant numbers TIN2009-14560-C03-03 and GRC-1087 respectively.

D. Dominguez-Sal and J.L. Larriba-Pey are with DAMA-UPC, Jordi Girona 1-3, Barcelona, Spain. E-mail: {ddomings,larri}@ac.upc.edu

J. Aguilar Saborit was at IBM when this work was performed. E-mail: jaguilar@microsoft.com

M. Surdeanu is with the Stanford Natural Language Processing Group, E-mail: mihais@stanford.edu

is a location procedure to know with high probability if a document is available and where in the cooperative cache; and (iv) we compare our proposals to the most recent and significant proposals using a fully-fledged semantic search engine based on question answering technology.

The paper is structured as follows. In Section II, we review the related work. Then, Section III describes the generic architecture that we target with our cooperative cache proposals. In Section IV, we describe the ESC data structure and the ESC-placement and ESC-search algorithms. We report our experiments with it in Section V. Finally, Section VI concludes the paper.

II. Related work

Depending on the application area, the adequacy and the cooperative caching considerations differ. For example, Wolman et al. discuss analytically the application of cooperative caching for web data in WANs [5]. One of their conclusions is that any simple cooperative caching algorithm is close enough to an oracle cache policy for caching web data. However, this setup is not generally valid for all the applications, and cooperative caching has been applied in multiple areas successfully [6]–[12]. In this work, we focus on the placement and the search of the partial computations of a complex search engine, stored in the main memory of the different nodes that are interconnected by a LAN. This scenario also differs from the assumptions by Wolman because the cost to process the documents to solve a complex query is larger than the time to read an HTML document and the communication costs are lower in a LAN than in a WAN.

In one of the first studies for data placement in cooperative caches, Dahlin et al. introduced n-chance forwarding, which is an strategy that selects the target node for forwarding at random, so no statistics from other nodes are used [13]. Later proposals focused on how to improve this algorithm, taking advantage of statistics of the cache. Feeley et al. proposed GMS, which implements a centralized process where all the nodes send statistics about the age of their cache contents to a coordinating node [8]. Ramaswamy et al. proposed Expiration Age in [10], where nodes exchange its recent cache eviction rate and decide to replicate the data on nodes with small contention. Dominguez-Sal et al. proposed Broadcast Petition Recently in [14], which uses a time variable to limit duplicates in the network but does not use global system information such as the ESC. It is also known which is the optimal placement if all the data access statistics are available, as shown by Koropolu [15]. This algorithm, however, is static and does not adapt if the distribution changes. Finally, some placement solutions are based on hash functions [16], but the main drawback of these approaches is that the application of hashes does not encourage the locality to access the data.

There has been also significant effort on the location of documents in a distributed environment. Sarkar and Hartman implement placement and search in a distributed cache using inaccurate information, or hints, about the state of a client-server system [17]. The hints are distributed among the clients, but, unlike our search mechanism, it relies ultimately on the information in the main central server to locate the data. Rhea and Kubiatowicz proposed attenuated bloom filters that summarize the information of distant nodes for routing data [4]. Nevertheless, they do not count frequencies and attenuated bloom filters mix the data location of several nodes in a bloom filter, which reduces its precision. Fan et al. use a bloom filter-based data structure to perform the search [3]. However, they do not provide a placement scheme, nor a history of content accesses, whereas our approach provides this with a single data structure. In our experimental section, we confirm that the combination of frequency and recency improves the location recall over approaches that only store a bit indicating if the cached entry was available at a certain moment of time, like summary caches. Additionally, ESC are a general data structure that may be used in different contexts. For example, Dominguez-Sal et al. discuss load balancing algorithms for distributed systems that are aware of the cache contents in order to improve the throughput and the data locality [18]. Nevertheless, in [18], it is not consider the placement and search problems, which we do in the current paper.

Besides, many algorithms have appeared from the peer to peer community to find information in large networks using distributed hash tables (DHT): Chord [19], Pastry [20], etc. Although DHTs were designed to be used in WANs, some prototypes implement them in networks with small latencies. For example, Shark implements a cooperative cache for distributed file systems [21], and Squirrel is a web server cache based on cooperation too [22]. But, one limitation of DHTs is that it needs to contact several nodes sequentially during the location procedure, which introduces latency in the search. Our proposal overcomes this problem because we are able to contact a small subset of nodes in parallel.

Finally, the aspects related to the management of the local memory pool dedicated to caching are orthogonal to the cooperative caching strategies proposed in this paper. For instance, if the document size exhibits a large variability, the largest documents may be partitioned into sections, paragraphs, or pages with a fixed number of characters. Also, we used an LRU strategy in our system as the local policy, but our distributed proposals can be abstracted away from the local policy. Thus, we believe more advanced local cache policies such as the proposed by Jiang et al. [9], can also benefit from our placement and search algorithms.

III. System architecture

We focus on search engines that are composed of a set of computing blocks, which we treat as black boxes, and the output of the system is obtained as a sequence of computations of these blocks. Some of the computing blocks might be computationally expensive, and thus, their outputs are ideal candidates for caching. Each computationally intensive block has a pool of memory to store the data related to a document (identified uniquely in the document collection) following a local cache policy. We select LRU as the local cache policy because its wide usage in many applications and its proven adequacy to the workload of search engines [23].

In order to build the distributed system, we consider that all nodes have an instance of the search engine, and can

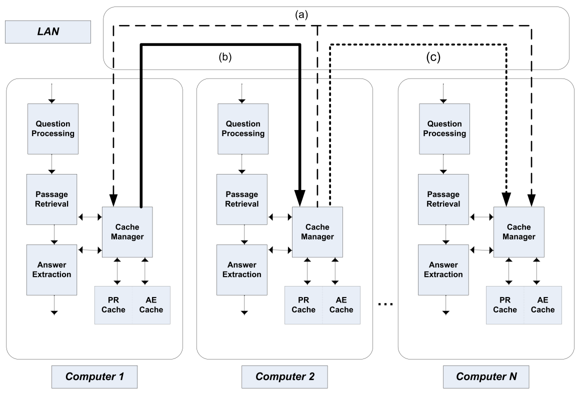


Fig. 1. QA system composed of three sequential blocks QP, PR and AE. PR and AE can request and store data into the cache using the cache manager.

access the document collection and its associated indexes. This design follows previous proposals for complex search engines [24], where each node is able to compute queries autonomously. We illustrate an adaptation of the previously described architecture to a question answering (QA) system with three computing blocks (question processing, passage retrieval, and answer extraction) and its corresponding local caching pools in Figure 1.

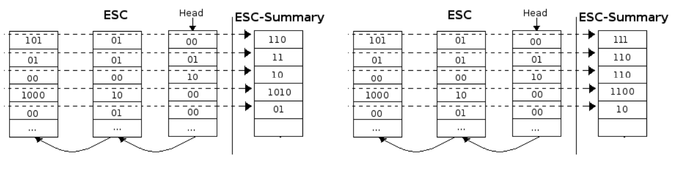
On top of the search engine, we implement a distributed cooperative cache with no centralized processes, which intuitively works like a peer to peer network (with no central node) where all nodes can directly contact the rest of nodes of the network. The communication between nodes is facilitated by the following operations:

- *Request/Response*: these operations obtain a cached entry from a remote node. Once a node has a local miss, it requests the document through a multicast operation (operation (a) in Figure 1). The request includes the document identifier and a parameter to identify the computing block that requests the data. The receivers of the request respond with the data if the entry is available in their caches (operation (b)).
- *Forward*: this operation transfers the least recently used cache entry from a layer to the same layer of another node in the network (operation (c)).

The access to the collection content can be optimized by using shared disks [25], [26], a distributed file system optimized for read operations [27], [28] or simply replicating the collection if the dataset fits in a computer. This architecture can be easily extended to very-large-scale settings. For example, one can envision a setup where a very large collection is partitioned and each different partition is replicated among a group of nodes, where each group implements our architecture.

IV. Cooperative caching algorithms

This section describes the new distributed cooperative caching algorithms proposed in this paper. We first describe the data structure that we propose in this paper and is shared by all our cooperative caching algorithms, the Evolutive Summary Counters (ESC). Then, we explain how to apply the ESC-summaries to the placement (ESC-placement) and the location of data (ESC-search).



(a) Plain Summary (b) Linear Summary

Fig. 2. Summarization example of an ESC (counters are in binary).

A. *Evolutive Summary Counters*

The Evolutive Summary Counters (ESC) are a data structure deployed in each computing node, which records a window of the recent history of the local accesses. The ESC is composed by a linked list of k Count Bloom Filters (CBF) [29]. During a certain period of time, τ, the CBF at the head of the list is active, i.e., it counts the occurrences of the elements in a streamed data set. After τ time units, a sliding operation is applied, so that a new CBF is used during the next τ time units. On reuse, the CBF at the back of the list is reset (all counters are set to 0) and it becomes the head of the list, that is, the active CBF.

Each access to a document is recorded into the active CBF locally. Thus, each computing node keeps an ESC with k CBF, which monitor the last $\tau \cdot k$ time units. After τ time units, when a sliding operation is triggered, each node computes a summary of the ESC that is broadcasted to the nodes in the network, the ESC-summary. Therefore, each node receives the ESC-summaries from the rest of nodes, and the distributed algorithms check the summaries to obtain a description of the system state in order to dynamically update its decisions.

The ESC-summary is computed as the aggregation of the k count filters of the ESC (see [30] for a detailed description of how to aggregate structures based on bloom filters). We evaluate two possible summary implementations:

- *Plain summary:* A simple addition of all the CBFs.
- *Linear summary:* A weighted addition where the most recent CBF is multiplied by k, the second most recent CBF is multiplied by $k-1$, and so on, up to the k-th CBF, which is multiplied by 1.

We depict a diagram of an ESC in Figure 2 with $k = 3$. The figure compares how we combine the three CBFs of the ESC in a plain and a linear summary. The counters in the ESC-summary take the weighted sum, by rows, of the CBFs. The ESC-summary generated is also a CBF.

The operations related to the ESC have low complexity. The recording of an access to a document in the ESC has constant complexity. The summary and slide process take linear time with respect to the CBF size, and the look-up of the frequency in a ESC-summary has also constant complexity. In the supplementary material, we show that the network overhead introduced by the ESC is small enough to permit systems with throughput of thousands of queries per second.

B. *ESC-Placement*

Our proposal performs the distributed placement when a local cache is full and a document is evicted from memory: the document is forwarded to another node if the algorithm decides that this document is valuable enough to be kept in some node of the network, otherwise it is simply discarded. Our objective is to keep the documents in the node where they are accessed, and to encourage the availability of frequent documents in some node of the network.

The target node for forwarding the evicted entry is decided according to the ESC-summaries that have been sent by the remote nodes. The algorithm selects the node whose ESC-summary contains the highest value for the entry being processed and transfers the entry to that node. If the entry is already present in the receiving node, our algorithm marks the entry as the most recently used entry for the local LRU policy. Also, if more than one node has the same largest value, a random selection among these nodes is performed. The pseudocode of ESC-placement is summarized in the supplementary material.

We have an additional counter for each entry in the cache, which accounts for the number of forwarding actions since the last access for each document. Each time a document is forwarded, its respective counter is incremented. If a document is accessed while staying in a cache, its counter is reset. The objective of this counter is to limit the number of forwarding actions to avoid excessive network traffic. We limit this number to 2 in accordance to the results obtained in [13].

ESC-placement automatically evicts from the cache entries scarcely accessed, i.e., documents not read after several forwarding operations. If an entry is evicted from the local cache we are not deleting it but trying to reduce the number of copies: the destination node is the one with the highest probability of holding a copy. If the destination node holds a copy of the entry it is not necessary to remove any other entry, and hence the forwarding procedure is finished. If the destination node does not hold a copy, ESC-placement repeats the forwarding procedure until (a) it finds a document that can be discarded because it is not frequently accessed (with a forwarding counter exceeding the threshold), or (b) a document with multiple copies in the network.

C. ESC-Search

The search algorithm is in charge of locating the node where a document is stored. The objective is to reduce the number of *avoidable misses* in the system: an avoidable miss is a remote cache miss for a document that is cached somewhere in the network, but it is not found because the system did not query the proper nodes. Although the broadcast of the requests reduces the number of avoidable misses to zero, the number of messages may become a bottleneck.

The ESC-search procedure is called whenever an entry d is not found in the local cache. First, ESC-search builds a list (L=$n_1, ..., n_N$) with all the nodes in the system. The nodes in L are sorted in decreasing order of access frequency to d, according to the ESC-summaries locally available. Then, ESC-search selects the first h nodes from L, and requests the entry from all of them in parallel (note that the selection process is computed without network communication). We compare two methods to select h:

Static: h is set to a constant value which is decided at the initialization of the system.

Dynamic: h is calculated every time a document is searched in order to keep the probability of an avoidable miss below a defined threshold. The dynamic ESC-search policy divides the nodes into two lists: the candidate nodes to be queried and the rest. Initially, the candidate node list is empty, and ESC-search estimates the probability of an avoidable miss if no node is queried ($P_{AvMiss}(0,d)$). Then, following a greedy procedure, ESC-search includes the node which is most likely to contain d in the candidate node list until the probability of an avoidable miss is below a threshold ϵ. ESC-search estimates the probability of an avoidable miss with the following formula, which assumes that the probability of finding a document in one node is independent among nodes:

$$\begin{aligned} P_{AvMiss}(s,d) &= prob(d\ is\ not\ in\ nodes\ (n_1..n_s]) \cdot \\ &\quad\ prob(d\ is\ in\ any\ of\ the\ nodes\ (n_s..n_N]) \\ &= \left[\prod_{i=1}^{i=s}\left(1-P_{freq(ESCS(i,d))}\right)\right] \cdot \\ &\quad\ \left[1-\prod_{i=s+1}^{i=n}\left(1-P_{freq(ESCS(i,d))}\right)\right], \end{aligned}$$

where $ESCS(i,d)$, is the value in the ESC-summary received from the i^{th} node for document d; s is the number of nodes that are queried to locate document d; and $P_{freq(x)}$ is the probability that a document with frequency x is available in the memory of the remote node. h is selected as the smallest number of nodes such that $P_{AvMiss}(s,d) < \epsilon$. In the worst case (which is unlikely), the algorithm will query all nodes, but will query one or even none if the document is very unlikely to be in cache. We show the pseudocode for this search procedure with an example in the supplementary material.

In order to evaluate $P_{AvMiss}(s,d)$, each node keeps an array with a local estimation of $P_{freq(x)}$, for each possible value of the counters in the ESC-summaries. This probability is updated dynamically according to the results of the search history. For example, in a system that had requested 5 times documents with frequency 2 and they were only found once, then the current value of $P_{freq(2)}$ would be $\frac{1}{5}$. After each search, this value is updated. For instance, if the document was found in the node whose ESC-summary indicated a frequency of two then $P_{freq(2)}$ becomes $\frac{2}{6}$.

The algorithm is able to adapt to a query distribution because $P_{freq(x)}$ is computed on the fly. The computational cost of this procedure is $O(N)$ in the worst case because the algorithm checks the summaries from all nodes for a given document. Nevertheless, $P_{AvMiss}(s,d)$ is a monotonic decreasing function, so as soon as it reaches a value below ϵ, this becomes the final result, and there is no need to further compute it for a larger set of nodes.

In the supplementary material, we model the overhead of the communication generated by the search algorithm. This analysis shows that the additional cost to transmit the ESC-summaries yields a significant reduction of nodes queried and the overhead of the algorithm is thus significantly smaller than that of the broadcast protocol.

V. Experiments

A. Experimental setup

In this section, we take question answering (QA) as an example of a complex information retrieval system, which demands special attention from caching because of its processing complexity. QA systems are search engines that process natural language queries and search for named entities such as person or location names. Since our QA system is modular, it allows us to reconfigure its processing pipeline to test our proposals for a wide variety of configurations with different CPU and I/O requirements. We report the most relevant results in this section, and we include a broader evaluation of our proposals in the supplementary material. The QA system implemented in this paper (Figure 1) uses a traditional architecture consisting of three blocks linked sequentially [1]: (a) *Question Processing* (QP) that parses and analyzes the natural language query given by the user, (b) *Passage Retrieval* (PR) that retrieves from the document collection the most relevant documents from the collection, and (c) *Answer Extraction* (AE) that analyzes the retrieved documents in PR with natural language tools, and returns the most adequate answers to the user. We detail the internal implementation of the system in the supplementary material.

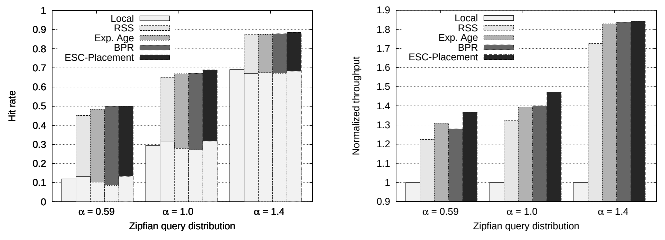
For our test we run the fully fledged QA system on a cluster of 16 nodes connected with a gigabit Ethernet. Each node in the system is equipped with an Intel dual core CPU at 2.4Ghz and 2GB of RAM. An additional computer, in the same local area network, is used as a client that issues queries to the rest of nodes following a round robin policy.

The question sets in our experiments consist of five thousand queries following typical search engine distributions. The queries were selected from the question sets that were part of former TREC-QA evaluations (700 different queries). We generate our query sets following typical distributions from real web search engines [23], [31], [32]: Zipf distributions with varying parameter α in the range [0.59,1.4], and typically below 1.0. In a Zipf distribution, the probability to access document i from the collection is proportional to $i^{-\alpha}$. Therefore, the larger α the more skewed is the distribution.

We set round robin as a baseline to compare all the algorithms under study with a neutral load balancing policy. Our load balancing policy resembles multiple clients sending queries to a server applying round robin DNS [33]. We keep the system under heavy load with an average of six queries per node, issuing a new query every time a query is answered. Since the objective of this work is to analyze placement and search using ESC, we do not emphasize the parametrization of ESC. In the experiments we set k and τ to the best possible values among a set of experiments that we have performed. In the supplementary material, we detail the experiments performed to estimate these parameters empirically.

B. Placement analysis

We perform an analysis of the placement method that we introduced in Section IV-B. We compared our proposal to the following algorithms. (a) A local policy where we disable all the remote operations of the cache manager. (b) Broadcast Petition Recently (BPR), which replicates only the elements that have been accessed multiple times in a fixed window of time [14]. (c) Expiration Age (EA) calculates an estimator, called *expiration age*, which is the average time that cache victims have spent in the cache since their last hit. When a node retrieves a document from another computer in the network, they exchange their expiration ages and a copy is only replicated if the requester has a larger expiration age [10]. (d) Random server selection (RSS) picks the forwarding node at random, with the restriction that each entry can only be forwarded a limited number of times since its last access. This policy is similar to the n-chance protocol described in [13].



(a) Hit rate (b) Throughput

Fig. 3. ESC-placement performance for different query distributions.

For a fair comparison of the placement algorithms explained in this section, we implement the same search algorithm for all the placement methods used. We use a simple protocol, which sends a broadcast message with the requested document identifiers, which is similar to ICP [34]. If any remote node has the needed contents, it sends them back to the requester. The broadcast guarantees that a document will be found if it is available in any node of the network. We analyze the more complex search strategies proposed above in Section IV-C.

The *plain* and the *linear* summarization performed similarly in the placement experiments. In this section, we only report the results that come from the *linear* summary implementation.

Experiment 1 (Query distribution): In this experiment, we test the performance of the question answering system for a variety of query distributions. According to the previous studies mentioned, the query logs follow Zipf distributions ranging from $\alpha = 0.59$ to $\alpha = 1.4$, with α typically between 0.59 and 1.0 . So, we generated three query sets following different Zipf distributions with varying parametrization: Zipf$_{\alpha=0.59}$, Zipf$_{\alpha=1.0}$, and Zipf$_{\alpha=1.4}$.

Figure 3(a) shows the hit ratio for the different algorithms tested. The bars are grouped by distribution: the most skewed distribution corresponds to the rightmost group of bars. For the cooperative caching algorithms, we indicate the local and remote hit rates. The local hit rate is the lowest part of the bar painted with light color. The remote hit rate corresponds to the darker part of each bar. The full bar accounts for the addition of the local plus the remote hit rate.

We observe that the query distribution affects significantly the total hit rate for all the algorithms, as well as the proportion of remote hits. For the less skewed distribution the total hit rate is smaller because the system accesses a wider diversity of documents. Moreover, we observe that the local cache policy has poor results compared to the cooperative policies because the available memory in one node is small with respect to the number of documents accessed. Only if the total available memory dedicated to caching in the network is combined using the cooperative caching algorithms, the hit rate is over 40%. We see that the use of the cooperative caching algorithms yields a three-fold improvement on the number of total hit rates, hence most hits in a cooperative caching algorithm come from the access to remote caches. When looking at more skewed distributions, the ratio between local and remote hits changes. In the case of the Zipf$_{\alpha=1.4}$ distribution, we observe that most data can be retrieved from the local cache, though the cooperative cache still contributes to improving the hit rate.

In Figure 3(b), we show the throughput of the different algorithms normalized to the local policy. The hit rate has a direct influence on the performance: cooperative caching algorithms are much faster than local caching. This huge difference is a strong recommendation for the use of cooperative caching algorithms in QA.

The difference among the cooperative caching algorithms is smaller than between the cooperative cache and the local policy, because all the cooperative caching algorithms take advantage of the cache in the remote nodes. We see that ESC-placement is the algorithm with the best global hit rate, and accordingly, with the best performance. It achieves a throughput above 1.84 times a locally managed system for Zipf$_{\alpha=1.4}$ distributions, and it is significantly better than other alternatives for the rest of distributions. We also observe that the local hit rate of ESC is above other cooperative caching algorithms, such as EA or BPR, because ESC does not only encourage global hits but also cares about locality. As already mentioned, ESC limits the number of copies available in the cooperative cache: it keeps multiple copies of the documents that are very frequently accessed, and reduces the number of copies if they are not so frequent. Therefore, if a document is very frequent there will be several copies of it in different nodes, improving the probability of a local hit.

We also notice that BPR is a very good algorithm from the hit rate perspective: it obtains the second best hit rate, close to ESC-placement. However, the BPR policy performs more remote accesses than other algorithms such as EA or ESC-placement, because BPR gives very low priority to the replication of documents. Due to the additional network traffic of remote hits, we see in Figure 3(b) that BPR throughput is significantly smaller than for ESC-placement, specially for small skews. In our tests, EA obtained better performance than BPR for the less skewed configuration because its local hit rate is better than for BPR, but it has a smaller throughput than ESC because the global hit rate is not as good. Although the lowest values of α are more common, we notice that for the most skewed distributions, Zipf$_{\alpha=1.4}$, all the algorithms have a very good performance. For large values of alpha, we detect tiny differences in the hit rate of cooperative caching algorithms because all of them are very close to the optimal, which is 0.95[1] for this distribution. We observe that RSS achieves a good balance between local and remote hits. However, its performance is not as good as for other algorithms because RSS contacts many nodes, and hence the forwarding policy is slower because of the additional network connections. All in all, ESC-placement obtains the best throughput of all placement algorithms analyzed due to the good hit rate obtained with minimal network overhead. This is evident especially for real-world setups, where the query distribution is not too skewed and the local cache is not sufficient.

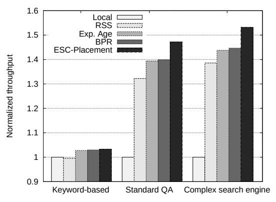
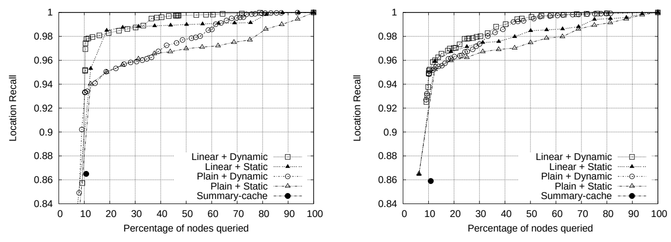


Fig. 4. Normalized throughput of search engines with different degree of complexity (Zipf$_{\alpha=1.0}$)

Experiment 2: (Impact of components with different computational costs): Depending on the search task and the quality of the output, the modules that process the results of a search engine have a different degree of complexity. For example, a traditional keyword based search engine needs fewer computational resources than a question answering system such as our testing system. However, some users may be willing to accept longer computational times to retrieve information for complex queries, or might want to include analysis of multimedia data associated to text, which is typically more computationally expensive than text processing.

In this experiment, we target the performance of the cooperative cache for such systems with varying computational complexity, i.e., we test the impact of our cooperative cache for different types of QA systems. In order to simulate the complexity of different search engines, we use two different alternatives for our answer extraction component. The first, which we call Standard QA, uses a simple maximum entropy with a limited feature set for candidate answer extraction. This component was used in all previous experiments as well. The second approach –Complex QA– performs candidate answer extraction using support vector machines and a richer feature set. The processing overhead of the latter component is approximately three times larger than our initial system [35]. This system is not adequate for interactive question answering, but simulates a system in which the user might sacrifice response time for a better answer quality. As a third testing system, we remove the AE module entirely. This system generates as output a collection of text snippets, which is very similar to the functionality of current keyword-based search engines. The execution time of the light system is approximately one fourth of our initial QA system.

[1]We computed this optimal hit rate running a modified version of our QA system that does not remove any cache entry once inserted (it does not store the full document but only a dummy entry for each identifier), and running on a single computer with a cold cache. We observed that the optimal hit rates were 0.86, 0.92, 0.95 for α=0.59, 1.0 and 1.4, respectively.



(a) Zipf$_{\alpha=0.59}$ (b) Zipf$_{\alpha=1.0}$

Fig. 5. Location recall for the ESC-search algorithms.

Figure 4 shows the normalized throughput for all the systems described previously for Zipf$_{\alpha}$=1.0. The standard QA system corresponds to the system we have used in the previous experiments. We see that in a light system the benefit of cooperative caching is not as large as the improvements found in previous experiments, because the network communication time is closer to the cache miss penalty.

ESC-placement performs the best out of the algorithms tested, because it obtains the highest hit rate with a low network overhead. Since each cache hit bypasses the computational overhead corresponding to that cache block, computationally-intensive systems benefit the most from cooperative cache policies such as ESC. The experiment shown in Figure 4 highlights this observation. Furthermore, the difference between ESC-placement and the local cache increases as the complexity of the underlying QA system increases, which indicates that the benefit of using ESC-placement increases with the complexity of the retrieval engine.

C. Search Analysis

In this section, we analyze the cache search algorithms proposed in Section IV-C. We use the same setup as in Experiment 1, with the addition of the ESC-placement algorithm described in Section IV-B, which was shown to perform the best in the previous section. The network bandwidth is 1 Gbps, and does not constitute a bottleneck for the number of nodes tested. In the experiments, we compare the two possible summary methods, i.e., *plain* and *linear* (see Section IV-A) because they behave significantly different for search. We also combine the two search variants described in Section IV-C, *static* and *dynamic*.

Experiment 3 (Location recall analysis): Intuitively, a good search algorithm must obtain a compromise between the number of nodes requested, and the *location recall* of the system, where the location recall of a search algorithm is defined as the fraction of documents found remotely divided by the documents found if all the nodes were queried. In this first experiment we analyze the different configurations for ESC-search and compare them to the summary caches, proposed by Cao et al [3].

Figure 5 shows the location recall for the two different summary procedures as a function of the number of nodes queried (plain and linear ESC-summary, see Section IV-A) combined with the static and the dynamic location algorithms. The horizontal axis is the percentage of nodes requested: for example, in our cluster, 25% means that, on average, four nodes are requested out of the 16 available. For static policies, each point in the plot corresponds to a configuration that fixes the number of nodes to a constant. For dynamic policies, each point in the plot corresponds to the average number of nodes queried when ϵ is set to a certain value.

In contrast to the results for placement, the search is very sensitive to the summary method: the linear summary achieves results comparable to querying all the nodes, when accessing only 10% of these nodes on average. On the other hand, the plain summary needs to query significantly more nodes to avoid a big number of avoidable misses. This shows that temporal information is crucial for efficient search in a distributed cache. From the point of view of the selection of the proper number of nodes to query, h, the dynamic approach converges to the maximum hit rate faster than the static. The reason is that the adaptability of the dynamic algorithm helps the system query only a few nodes when a document is popular and find rare documents when more nodes are queried. The trends of the algorithms are similar for both distributions investigated. We attribute the slightly slower location recall for the Zipf$_{\alpha=1.0}$ query set to the long tail of rare elements in the distribution, which are the hardest documents to find because they are not replicated in the network and only remain in the global cache for a short time. In our experiments, the combination linear+dynamic achieves the best location recall for almost all percentages of nodes queried. For both distributions, linear+dynamic achieves 99% location recall with only 40% nodes queried. The plot shows that, if the network becomes the bottleneck of the system, we can increase ϵ and then ESC-search queries only 1.75 nodes on average (approximately 10% of the nodes) because linear+dynamic obtains a location recall only 2.1% lower than broadcasting. This is remarkable, because the traffic in the network may be a significant bottleneck in some cases, and this location recall is usually enough for practical purposes.

We also compare our proposal to summary caches, configured with the same parametrization as our ESC-summary. Note that even though the summary cache sends smaller updates because it does not count frequencies but existence, the ESC-search uses the same information as ESC-placement. Hence, a system that implements ESC-placement can implement ESC-search with no additional summary communication.

Figure 5 shows that ESC-search achieves better recall than summary caches. In our configuration, summary cache queries approximately 1.5 nodes on average to reach a location recall of about 0.86 for both query distributions. ESC-search improves the score to 0.95 querying the same number of nodes. ESC-search improves the recall of summary caches because: (i) it is able to detect the nodes that potentially store a document even though in the last update the data was not cached; (ii) if a document is very popular then ESC-search queries a reduced set of nodes.

VI. Conclusions

Next generation search engines will not only have to handle large amounts of data but also invest more computational effort in understanding the underlying content, e.g., multimedia analysis of embedded video and sound, semantic understanding of natural language. This paper introduced a new cooperative caching policy that focuses on this scenario. We used a state-of-the-art distributed question answering as our use case.

The core of our distributed cache environment is the ESC, which is an efficient data structure that captures the frequency of accesses to the documents from big text collections and generate summaries efficiently. To our knowledge, ESC is the first proposal for a data structure that captures the recent access frequency to data collections in distributed systems.

For computational-intensive systems such as QA, our placement approach provides significantly better throughput for a broad variety of scenarios, including different query sets, query distributions and system configurations. Overall, ESC-placement provides a speedup up to 1.81, compared to locally managed caches. Our tests have also shown that ESC is valuable for variants of our basic architecture: the system improves the speedup for systems that compute more complex analysis than ours, and for systems that are able to store preprocessed data on disk, and thus are I/O bound.

Additionally, our search strategy achieves better location recall than other state-of-the-art approaches such as summary cache. Our proposal, ESC-search, obtained a location recall of more than 0.95 compared to 0.86 achieved by summary caches with a similar amount of communication. Compared to a broadcast protocol, the average number of nodes queried by ESC-search was reduced by 90% with respect to the broadcast protocol, while still finding the documents with a probability higher than 97.5%. Furthermore, ESC-search is a flexible algorithm, which is able to adjust the location recall for each document request individually, if desired. For example, in a system where some documents require more computation than others, it is possible to adjust ESC-search dynamically to virtually achieve a perfect recall for the most expensive computing cache entries.

Our tests have been performed on a cluster of computers running a question answering system. However, we believe that the algorithms described in this paper can be applied generally to other configurations and different applications. For example, we can envision their application to image search engines that in the first step filter out a set of candidate images and videos, and then use image analysis to obtain high quality answers, which is a similar architecture to ours. In our paper, we have modified our testing system to resemble these different scenarios, and we found that our ESC based proposals can have an even more important impact on the performance of such complex search engines than on simpler ones.

References

[1] 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.

[2] J. Sivic and A. Zisserman, "Video google: A text retrieval approach to object matching in videos," in *ICCV*, 2003, pp. 1470–1477.

[3] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[4] S. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *INFOCOM*, 2002.

[5] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy, "On the scale and performance of cooperative web proxy caching," in *SOSP*, 1999, pp. 16–31.

[6] T. Anderson, D. Culler, and D. Patterson, "A case for now (networks of workstations)," *IEEE Micro*, vol. 15, no. 1, pp. 54–64, 1995.

[7] M. Raunak, "A survey of cooperative caching," *Technical report*, 1999. [Online]. Available: http://citeseer.ist.psu.edu/raunak99survey.html

[8] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath, "Implementing global memory management in a workstation cluster," in *SOSP*, 1995, pp. 201–212.

[9] S. Jiang, F. Petrini, X. Ding, and X. Zhang, "A locality-aware cooperative cache management protocol to improve network file system performance," in *ICDCS*, 2006, p. 42.

[10] L. Ramaswamy and L. Liu, "An expiration age-based document placement scheme for cooperative web caching," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 5, pp. 585–600, 2004.

[11] K. Lillis and E. Pitoura, "Cooperative xpath caching," in *SIGMOD Conference*, 2008, pp. 327–338.

[12] Y. Du, S. Gupta, and G. Varsamopoulos, "Improving on-demand data access efficiency in manets with cooperative caching," *Ad Hoc Networks*, vol. 7, no. 3, pp. 579–598, 2009.

[13] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative caching: using remote client memory to improve file system performance," in *OSDI*, 1994, pp. 267–280.

[14] 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.

[15] M. Korupolu and M. Dahlin, "Coordinated placement and replacement for large-scale distributed caches," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 6, pp. 1317–1329, 2002.

[16] T. Cortes, S. Girona, and J. Labarta, "Design issues of a cooperative cache with no coherence problems," in *IOPADS*, 1997, pp. 37–46.

[17] P. Sarkar and J. Hartman, "Efficient cooperative caching using hints," in *OSDI*, 1996, pp. 35–46.

[18] 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.

[19] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[20] A. Rowstron and P. Druschel, "Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware*, ser. LNCS, vol. 2218. Springer, 2001, pp. 329–350.

[21] S. Annapureddy, M. Freedman, and D. Mazières, "Shark: Scaling file servers via cooperative caching," in *NSDI*, 2005.

[22] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: a decentralized peer-to-peer web cache," in *PODC*, 2002, pp. 213–222.

[23] E. Markatos, "On caching search engine query results." *Computer Communications*, vol. 24, no. 2, pp. 137–143, 2001.

[24] M. Surdeanu, D. Moldovan, and S. Harabagiu, "Performance analysis of a distributed question/answering system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 6, pp. 579–596, 2002.

[25] T. Lahiri, V. Srihari, W. Chan, N. MacNaughton, and S. Chandrasekaran, "Cache fusion: extending shared-disk clusters with shared caches," in *VLDB*, 2001, pp. 683–686.

[26] E. Rahm, "Parallel query processing in shared disk database systems," *SIGMOD Record*, vol. 22, no. 4, pp. 32–37, 1993.

[27] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 51–81, 1988.

[28] D. Borthakur, "The hadoop distributed file system: architecture and design," *http://lucene.apache.org/hadoop/hdfs_design.html*, Sept. 2008.

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

[30] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, 2003.

[31] P. C. Saraiva, E. S. de Moura, R. Fonseca, W. M. Jr., B. Ribeiro-Neto, and N. Ziviani, "Rank-preserving two-level caching for scalable search engines," in *SIGIR*. ACM, 2001, pp. 51–58.

[32] R. Baeza-Yates, "Web usage mining in search engines," in *Web mining: applications and techniques*, A. Scime, Ed. Idea, 2005, pp. 307–321.

[33] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu, "The state of the art in locally distributed web-server systems," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 263–311, 2002.

[34] K. C. D. Wessels, "Internet Cache Protocol: Protocol Specification, version 2," *RFC 2186*, 1997.

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

David Dominguez-Sal is currently working as a postdoc researcher in DAMA-UPC. He obtained his computer engineering degree at the Facultat d'Informatica de Barcelona in 2004 and his PhD in Computer Architecture from Universitat Politècnica de Catalunya in 2010. His main research interests are question answering, distributed systems and graph data management.

Josep Aguilar-Saborit graduated at the Facultat d'Informatica de Barcelona in 2002 and obtained his PhD from Universitat Politècnica de Catalunya in 2006. He is at present working at Microsoft Corp. in the SQL Server Parallel DataWarehouse Edition. His research interests are in the area of Database system design, performance and implementation among others.

Mihai Surdeanu is a Senior Research Associate in the Computer Science Department at Stanford University. Dr. Surdeanu also serves as the Chief Technical Officer of Lex Machina (lexmachina.com), a company that focuses on information extraction and risk analysis in the legal domain. Mihai Surdeanu earned a PhD degree in Computer Science from Southern Methodist University, Dallas, TX, in 2001. From early 2000 to 2004 he was employed by Language Computer Corporation as a Research Scientist. Since August 2001 he also served as the company's Vice-President of Engineering. In 2004 he moved to Barcelona Spain, where he held Senior Research Scientist positions at Universitat Politècnica de Catalunya, Barcelona Media Foundation and Yahoo! Research Barcelona (the last two concurrent). He joined the Stanford natural language processing group in 2008. Mihai Surdeanu's research interests are in natural language processing (NLP) and distributed processing applied to computationally-intensive NLP applications such as question answering. His recent work applies machine learning techniques to NLP problems such as information extraction, semantic role labeling and question answering.

Josep Lluis Larriba Pey is associate professor with the Computer Architecture Department at Universitat Politècnica de Catalunya and director of the DAMA-UPC research group www.dama.upc.edu. He has been coordinating different research and technology development projects with different companies and administrations, in projects related to high performance data management for large data sets and, in particular, for data integration and graph management.