Cache-aware Load Balancing for Question Answering *

David Dominguez-Sal
DAMA-UPC
Jordi Girona 1,3
Barcelona, Spain
ddomings@ac.upc.edu

Mihai Surdeanu
Fundació Barcelona Media
Ocata 1
Barcelona, Spain
mihai.surdeanu@barcelonamedia.org

Josep Aguilar-Saborit
IBM Toronto Lab
8200 Warden Ave
Markham, Canada
jaguilar@ca.ibm.com

Josep Lluis Larriba-Pey
DAMA-UPC
Jordi Girona 1,3
Barcelona, Spain
larri@ac.upc.edu

ABSTRACT

The need for high performance and throughput Question Answering (QA) systems demands for their migration to distributed environments. However, even in such cases it is necessary to provide the distributed system with cooperative caches and load balancing facilities in order to achieve the desired goals. Until now, the literature on QA has not considered such a complex system as a whole. Currently, the load balancer regulates the assignment of tasks based only on the CPU and I/O loads without considering the status of the system cache.

This paper investigates the load balancing problem proposing two novel algorithms that take into account the distributed cache status, in addition to the CPU and I/O load in each processing node. We have implemented, and tested the proposed algorithms in a fully fledged distributed QA system. The two algorithms show that the choice of using the status of the cache was determinant in achieving good performance, and high throughput for QA systems.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Performance, Algorithms, Experimentation

*David Dominguez is supported by a grant from the *Generalitat de Catalunya* (2005FI00437). The authors want to thank *Generalitat de Catalunya* for its support through grant number GRE-00352 and Ministerio de Educación y Ciencia of Spain for its support through grant TIN2006-15536-C02-02, and the European Union for its support through the Semedia project (FP6-045032).

CIKM'08, October 26–30, 2008, Napa Valley, California, USA.

1. INTRODUCTION

Traditional information retrieval (IR) engines provide searches based on keywords to retrieve a relevant document. Despite that they are indisputably useful, many searches target only a very short part of a document, like a paragraph or an entity. It is difficult to achieve the desired precision of an answer without a deeper understanding of the document content. Thus, the trend in the foreseeable future is that IR systems are going to increase their computational costs to read and analyze the documents. We take Question Answering (QA) as an example of systems with such additional costs and we target the problem of how to distribute properly the load in a distributed system.

Load balancing algorithms implemented in distributed systems assign the tasks to each node in such a way that all the resources available are used evenly. In order to achieve the best performance, it is necessary to feed the load balancing algorithm with an estimation of the resources needed for each task as close as possible to the real needs of the task. There are different types of load balancing algorithms based on dynamic or static techniques, in other words, algorithms that take or do not take into account the evolution of the environment. Most of those algorithms are based on modeling only the CPU [24], the disk I/O [7], or both [22], but none of them is aware of the cache contents in the system, the CPU load, and the I/O load. If the cost to process a task is incorrectly estimated, the solutions to rebalance the tasks in a distributed system may be cumbersome, leading either to: (a) aborting a task in the overloaded node, and transferring it to a different node [12], or (b) migrating a task preemptively from one computer to another [18]. In both cases, the impact on the system is important and it adds processing overhead and additional network communications.

Information retrieval systems, which are distributed in clusters, often implement data caches that can reduce significantly the computing time of a task. In this scenario, the load balancing algorithm could overestimate the cost of some tasks, leading to undesired imbalances. We build a simple simulation model (described in [10]) to quantify the potential imbalance introduced by cached data. The plot in Figure 1 shows the parametrization of this model for a distributed system that estimates the imbalance whose origin is the cache. We model a non cache-aware load balancing algorithm, that assigns a set of tasks to the nodes in the

network. Some nodes become underloaded because their assigned tasks have their data cached and take a shorter time to be finished, whereas some nodes will be overloaded because their tasks do not have their data cached. Figure 1 plots the percentage of nodes that are idle depending on the system hit rate. We observe that, for large hit rates, a vast majority of the nodes are imbalanced, which shows that placing an effort on the use of information about the caching system may improve the balance of the system.

The main contribution in this paper is the proposal of two dynamic load balancing algorithms that consider all the factors that affect the performance of a QA system: the CPU, the I/O and the cache. We decompose the execution of a query into multiple tasks and we trigger our load balancing algorithms at different stages during the execution of a query to improve the distributed system performance. We target complex systems where all the tasks do not behave homogeneously: each task has a different CPU and I/O usage. Moreover, the final cost to process a task varies according to the current state of the caches in the system. Thus, the load balancing algorithm must be aware of the resources and the cached contents in the cluster, in order to pick the best node to continue the execution of a query.

The first algorithm proposed, *Probability Cost* (PC), estimates the cost of processing a task of the query depending on where the information is located in the distributed cooperative cache of the system, and the current CPU and I/O loads. The second algorithm, *Affinity* (AF), additionally takes into account the frequency of accesses to the documents in the past to exploit the data locality for future queries. Moreover, AF is able to divide unevenly the workload to get a larger benefit of the cache and use the disk and the CPU more efficiently.

As a second contribution of this paper, we apply PC and AF to a fully fledged distributed QA system that we have built [9]. The execution time of our QA system is dominated by two tasks: the retrieval of documents from disk and the processing of those documents with natural language tools. The first task requires a large amount of I/O, whereas the second task consumes many processor cycles. In our distributed system, we have a distributed collaborative cache for the raw disk documents and for the processed documents. Our algorithms decide where each of those two tasks are executed, taking into account how the CPUs and disks are occupied, and where the most relevant data for their execution is stored in the distributed cache.

As a third contribution of this paper, we compare our proposals to the best previous contributions for QA, Weighted Average Load (WAL) [22]. First of all, our algorithms behave better than WAL with a speed up of 1.38. Second, for environments where the cost of the document processing is light, AF behaves better, while in the other cases, it is PC that does so. Third, our load balancing algorithms allow our QA system to obtain a significant throughput with an average of 6.27 queries answered per second, compared to the 4.55 queries for the baseline algorithm.

Paper structure: The paper is organized as follows. In Section 2, we give a brief description of our QA system, and how the distributed architecture is organized. Then, Section 3 describes the load balancing algorithms tested in this paper: first the non cache-aware and continuing with the new load balancing algorithms. Following, we report the experimental evaluation of the load balancing techniques

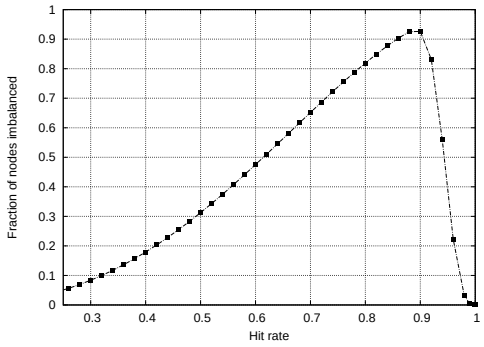

Figure 1: Fraction of nodes imbalanced from a cluster of 1024 nodes.

for a wide variety of configurations in Section 4. Section 5 reviews some of the related work. Finally, we draw some conclusions and expose our ideas for future work.

2. QA ARCHITECTURE

In this paper, we use a fully-fledged factoid QA system, whose implementation details are presented in [9]. We depict the system modules in Figure 2. The implementation of the QA system follows a traditional architecture of a pipeline with several sequential computing blocks: (i) Question Processing (QP), which analyzes the query, understands the question focus, and transforms the natural language question into a computer data structure; (ii) Passage Retrieval (PR), which is an IR system that obtains from disk the set of the most relevant documents for a query; and (iii) Answer Extraction (AE), which applies natural language tools to process the documents read in PR and identifies the most relevant answers for the query. The system is modular and we can vary its configuration to test its performance in different environments.

From a data processing perspective, our QA system implements a two-layered 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 implement a caching layer after each stage. The first layer caches the documents read from disk in PR, and the second caches the document analysis coming from AE. This local cache configuration is analyzed in [9]. This system obtained state-of-the-art performance in an international evaluation [21]

2.1 The Distributed Architecture

In order to build a distributed system, we replicate the local system in each node of the network. QA systems with text collections that are too large to be replicated can partition the collection and assign each partition to a group of nodes [4], in which each group behaves similarly to our architecture. On top of the QA system, we deploy a cooperative cache using an algorithm similar to ICP [23]: a node can query the rest of nodes in the network to retrieve the data associated to a document identifier (it is possible to retrieve the data for PR and for AE: the full raw text of

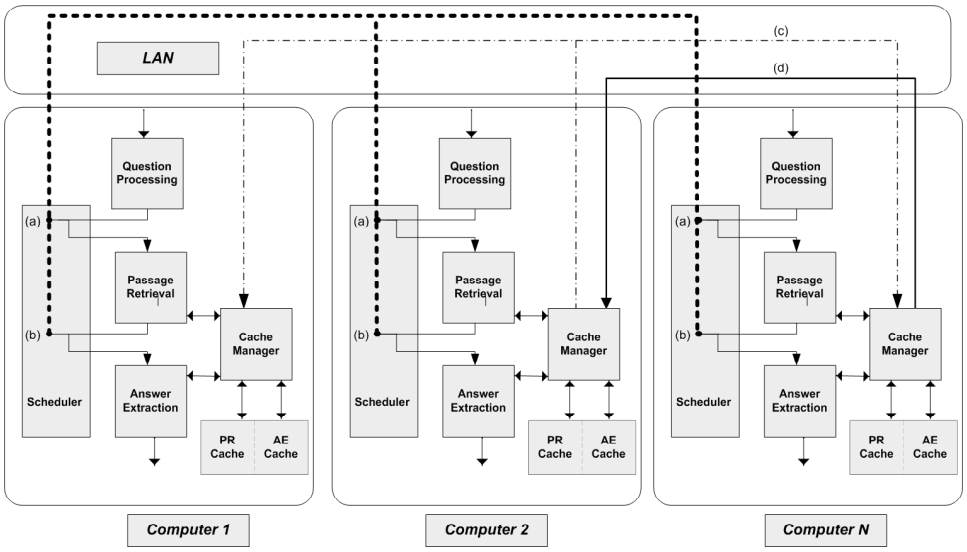

Figure 2: Diagram of the three computing blocks of our QA system: QP, PR and AE. We implement two scheduling points, for PR (a) and AE (b), where the execution can continue locally or be forwarded to a different node. The cache manager stores the documents retrieved from disk in PR, and the processed data generated in AE. (c) Request a document in the cooperative cache. (d) Send the requested data.

the document as well as its natural language analysis); and if any node has the contents available in its cache, it sends the requested data to the querying node (operations (c) and (d) in Figure 2). Once the data is received, it is added to the cache of the requester node. Following this procedure, any node can see the cache contents of the rest of nodes that belong to the distributed QA system.

Each query runs in its own thread, so several queries can be simultaneously executed in a node, even if they are executing the same computing block. In our system, the set of CPUs on a node share a waiting queue for pending tasks. We allow one more task than CPUs in order to avoid having multiple threads competing for the same resources. If a query is going to start the execution of a computing block and there are no resources available, the computing block is queued until another computing block finishes.

The scheduling points: We add two scheduling points to the system, which are depicted in Figure 2. The PR scheduling point (a) is triggered after the query reads the indexes from the document collection and computes the list of the document identifiers that will be read from disk, and before the complete documents are read from disk. The AE scheduling point (b) is situated after the documents are read from the disk and before they are processed by the natural language tools. PR and AE are the most expensive tasks of the QA system with more than 98% of the execution time. Queries that reach a scheduling point know the set of document identifiers involved in the current task of the query, before the expensive computation starts. We use the variable $Data_{(task(q),q)}$ to refer to the size of this set of identifiers. $Data_{(PR,q)}$ is the number of documents that q read in PR. $Data_{(AE,q)}$ is the number of documents that will be analyzed by the natural language tools. Note that, due to a filtering step following PR, the set of documents that are processed in AE is typically about one order of magnitude smaller than the set of documents read from disk in PR.

When a query reaches the scheduling point, the node triggers the load balancing algorithm to decide in which node the query is going to continue its execution. If the load balancing algorithm decides that the query should continue running locally, then the query continues its execution immediately, or it is queued if there are no resources available for that task. If the load balancing algorithm selects a remote node, the query is packed and transferred to the selected node. Each time a query finishes a computing block in the system, all the queued queries are rescheduled by the load balancing algorithm again with the updated stats from the rest of nodes. A query that is waiting in the queue to be executed locally can thus be rescheduled and assigned to a new node because, for example, the remote node has new cached contents or is less loaded. In order to simplify our architecture we limit to one the number of forwards per computing block, i.e. a query assigned to a node for AE can not be forwarded again to compute the AE block. However, it is possible that a query is forwarded in each of the computing blocks: once for PR and once more for AE.

Measuring the system load: Each node i measures its current load in two dimensions: one for the I/O ($Load_{(i)}^{IO}$) and another one for the CPU ($Load_{(i)}^{CPU}$). Each node sends its load measure to the rest of the nodes in the network periodically or if their current value differs more than fraction since its last update. Summarizing, all the nodes compute their local load, and receive recent load stats from all the computing nodes. Additionally, we use this periodic communication to detect when a node is not available, and when a new computing node has joined the network.

Our load balancing algorithms combine the two dimensions of the load measure to select the most suitable server to continue the execution of a query. The CPU load of i is calculated as the aggregated CPU time that is necessary to complete the current computing block of the queries as-

signed to Q_i (this includes the queries that are currently running and the queries waiting in the queues):

$$Load_{(i)}^{CPU} = \sum_{q \in Q_i} \left(C_{(task(q))}^{CPU} \cdot Data_{(task(q),q)} \right),$$

where $C_{task(q)}^{CPU}$ is the average cost, measured in time, that it takes to process a data unit in the current computing block of query q. The system measures the cost to process a computing block dynamically according to the recent history, and for each of the computing blocks. So, the system stores a different cost for each of the three different tasks: $C_{(QP)}^{CPU}$, $C_{(PR)}^{CPU}$ and $C_{(AE)}^{CPU}$. The system records the time spent in each different computing block of the recent queries answered by the node, and sets $C_{(task(q))}^{CPU}$ as the average time spent by the previous queries. Note that, although two queries are in the same computing block, the load contribution from a query that accesses a large number of documents is heavier than for a query that accesses a few documents because the number of units to process is larger ($Data_{(task(q),q)}$). A similar procedure is used to calculate $Load_{(i)}^{I/O}$, and all the associated information related to I/O.

State of the distributed cache: The distributed QA system implements an algorithm to monitor the state of the caches in each node of the network efficiently. Each node mantains a data structure, called Evolutive Summary Counters (ESC, described in [8]), that keeps a record of the recent documents accessed in a node (during both PR and AE). ESC monitors what documents are accessed in each node, and it can be used to monitor the current state of the distributed cache. The data structure can be shared by different cache-aware algorithms, for different purposes, and its computation cost can be amortized by the different algorithms. For example, the same ESC can be used to reduce the number of queries to locate a document in the network, or to improve the cache hit rate with a placement algorithm of the data [8]. In this paper, we focus on the load balancing problem and we use ESC as a tool to provide information of the global cache state to our load balancing algorithms.

An ESC is similar to the summary caches proposed by Fan [11]: both report recent information about the nodes in the network. Both structures use Count Bloom Filters (CBF), that is a variant of Bloom Filters [3] to count the number of elements in a set. Like Bloom Filters, a CBF is very compact because they keep an approximate count that can differ from the real value, with a fraction of error that can be tuned as desired. In both proposals, summary counters and ESC, each count filter is active for a certain period of time in a round robin fashion and, at certain intervals of time, each computing node generates a summary of its local CBFs and sends the summary to the rest of nodes. However, the summary caches report only the current contents in the cache, while the ESC summaries cover all the documents read in the recent history. This difference is important because an ESC contains information about the usage of non frequent documents, which may not be cached, and that we apply to improve the load balancing. All in all, in our QA system, each node receives an ESC summary from each node in the network: the ESC summary received from a node i contains the number of times that document d has been read recently in i, $ESC_i(d)$. Note that the ESC only use the network during periodic updates. However, a node can check at any moment the number of times a document has been accessed in a certain node without any new communication.

Our load balancing algorithms check the ESC summary received from a node to estimate the probability that a certain document is cached in that node. The probabilities are calculated using the location procedure described in [8], which estimates the probability that a document d is cached at a certain node i, $P_{(d \in i)}$. The value of $P_{(d \in i)}$ is computed dynamically according to the number of recent accesses to d in that node[1]. Our cache-aware algorithms use this probability to estimate which nodes contain a document with high probability, and the probability that the document can be found in a certain node of the network.

3. LOAD BALANCING TECHNIQUES

We divide the reported algorithms into two categories: non cache-aware algorithms, and cache-aware algorithms. Each of these techniques is executed every time a query reaches a scheduling point. The load balancing algorithm selects a node, s, that will continue the execution of the query. s is a node that belongs to the set of available computing nodes in the network N.

3.1 Non cache-aware algorithms

Here, we describe three caching algorithms that do not take into account the cache contents. The first two algorithms perform a distribution of the work that is not aware of the tasks executed. The third algorithm is used to schedule jobs that need to use the CPU as well as the disks, as in the QA case. We use these algorithms as a baseline to compare against the cache-aware algorithms.

Round robin (DNS): There is no dynamic load balancing algorithm in the cluster. The client sends the queries to the nodes in the cluster following a round robin policy. Each query is executed in a single node, hence s is always the local host. This technique simulates a DNS-like load balancing scheme. Note that the DNS-based policies in the internet, suffer from imbalances produced by the DNS caches in the network [5], that we omit here.

Random: This method picks the node s at random from the available servers in the system. This method does not take into account the load in each node to take the decision.

Weighted Average Load (WAL): This algorithm, described in [22], assigns the query q to the least loaded node in the system according to the CPU and I/O usage of q. This algorithm is CPU and I/O aware. Once the query reaches a scheduling point, WAL estimates the cost to calculate q in each node of the network, and picks s as the node with the lowest weighted average load:

$$s_{\text{WAL}} = \arg\min_{i \in N} \left(W_{(q)}^{CPU} \cdot Load_{(i)}^{CPU} + W_{(q)}^{I/O} \cdot Load_{(i)}^{I/O} + \zeta_{(i)} \right), \quad (1)$$

where $W_{(q)}^{CPU}$ is the fraction of time that q will spend in the CPU, $W_{(q)}^{CPU} = C_{(task(q))}^{CPU} \cdot \left[C_{(task(q))}^{CPU} + C_{(task(q))}^{I/O} \right]^{-1}$, and

[1]The intuition behind the search algorithm in [8] is that the more frequently a document is accessed, the more probable it is cached in that node. So, each node estimates a different hit probability for each different document access frequency. This estimation is corrected in such a way that the probability of hit is increased for future queries, when a document is found, and the probability is reduced otherwise.

$W^{I/O}_{(q)}$ is the analogous value for the I/O. $\zeta_{(i)}$ is a parameter to reduce the number of forward operations if the amount of imbalance is small: if i is the local node $\zeta_{(i)}$ is 0, and otherwise it is the average time to compute the next computing block of q. Hence, a query is only forwarded when the gain produced by its forwarding is bigger than its own cost in the current node. If several nodes have the same averaged load a random one is chosen among them.

3.2 Cooperative Cache-Aware Algorithms

Cooperative caching is an effective technique to reduce the execution time of document retrieval systems [9]. In a system with a cooperative cache, the queries can retrieve data not only from the local cache in the node, but also to request the data cached in the memory of remote nodes. Thus, the execution time of a query does not only depend on the local cache information but also on the global state of the cluster. Load balancing techniques need to estimate the cost to process a task as accurately as possible in order to make better decisions. Hence, it is beneficial to integrate the cache state into the load balancing algorithms to improve the accuracy of the estimation. Our proposals rely on the information distributed by the ESC-summaries, described in Section 2. Note that the cost to process a cache miss is much higher than the cost to inspect the ESC summaries, so using ESC pays off.

Our algorithms are fully distributed and do not have any centralized process. Thus, even if a subset of nodes in the network crash or have to be added to the system, our system adapts to these dynamic changes.

In this paper, we implement two algorithms:

Probability Cost (PC): This algorithm modifies WAL to include the impact of a cache hit in the query cost. PC changes the formula to compute $C^{CPU}_{(task(q))}$, which we refer now as $C'^{CPU}_{(task(q),i)}$. $C'^{CPU}_{(task(q),i)}$ defines the CPU cost to execute the task q, or in other words, the additional load added to the system if q is executed on node i. The new formula reflects the load reduction produced by the cached data. The new value is the weighted sum of costs to process a document depending on the cache state of the N nodes in the network: a local hit, a remote hit or a cache miss ($C^{CPU}_{HIT(task(q))}$, $C^{CPU}_{RHIT(task(q))}$, $C^{CPU}_{MISS(task(q))}$, respectively):

$$\begin{aligned} C'^{CPU}_{(task(q),i)} = \sum_{d\in task(q)} \Big[& C^{CPU}_{HIT(task(q))} \cdot P_{(d\in i)} + \\ + & C^{CPU}_{RHIT(task(q))} \cdot P_{(d\notin i \wedge d\in N)} + \\ + & C^{CPU}_{MISS(task(q))} \cdot P_{(d\notin N)} \Big]. \end{aligned}$$

The probability to find a document in node i, $P_{(d\in i)}$, is estimated by the location procedure. The probability of a miss, $P_{(d\notin N)}$, can be calculated with the assumption that the cache contents in each node are independent among them: $P_{(d\notin N)} = \prod_{i\in N}(1-P_{(d\in i)})$. The probability of a remote hit, $P_{(d\notin i\wedge d\in N)}$, stands for the documents that are neither local hits nor misses: $P_{(d\notin i\wedge d\in N)} = 1-(P_{(d\in i)} + P_{(d\notin N)})$. Then, we estimate the cost of a hit/remote hit/miss, $C'^{CPU}_{(task(q),i)}$, in each computing block using the costs recorded for the last k queries answered by the node. Finally, we apply an analogous procedure to obtain $C'^{I/O}_{(task(q),i)}$. PC calculates the local load in a node according to the new procedure to calculate $C'^{CPU}_{x,localhost}$ and $C'^{I/O}_{x,localhost}$, and sends its load to the rest of nodes like WAL.

Finally, we modify the server selection formula to indicate wether the query will find the documents locally, or it will retrieve them using the cooperative cache. The new formula depends on the current query as well as the cooperative cache contents because we add to the current load in node i the cost to process q in i. The algorithm selects the least loaded node according to the load in each node and the state of the cache in each node:

$$\begin{aligned} s_{PC} = & \arg\min_{i\in N} \Big[W^{CPU}_{(q)} \cdot (Load^{CPU}_{(i)} + C'^{CPU}_{(task(q),i)}) + \\ + & W^{I/O}_{(q)} \cdot (Load^{I/O}_{(i)} + C'^{I/O}_{(task(q),i)}) + \zeta_{(i)} \Big]. \end{aligned}$$

The modified formula enforces the access to the information locally. $C'^{y}_{(x,i)}$ is lower for the nodes that have the information locally available. Even if remote hits are not much more expensive than local hits, it is faster to access the information locally and reduce the network traffic.

Affinity (AF): This algorithm aims at combining two metrics to improve the performance of the system: the load in each node, and the affinity between the data retrieved by the query and the cache contents of a node. The first metric is used to send the query to the node with more resources available, the second tries to additionally exploit the locality of accesses.

The nodes measure their current load in the same way as in PC, which takes into account the cache hits. However, we introduce a factor in the selection of the most suitable node, $\vartheta(i,q)$, that measures the affinity of a query q with the node i. The modified formula for the node selection is:

$$\begin{aligned} s_{AF} = & \arg\min_{i\in N} \Big[\vartheta^{-1}_{(i,q)} \cdot \Big(W^{CPU}_{(q)} \cdot (Load^{CPU}_{(i)} + C'^{CPU}_{(task(q),i)}) + \\ + & W^{I/O}_{(q)} \cdot (Load^{I/O}_{(i)} + C'^{I/O}_{(task(q),i)}) + \zeta_{(i)} \Big) \Big]. \end{aligned}$$

Two properties are desirable to estimate the affinity of a query with a node: (a) affinity is higher for the nodes where the data is cached, and it is lower for the rest of nodes; (b) it gives more weight in the score to rare documents because it is preferable to replicate popular documents rather than rare ones in the network. Although other formulas may be applied[2] , we calculate ϑ with a popular relevance formula used in IR, the $tf \cdot idf$ [19]. IR focuses on finding the set of documents most relevant to a given input query. Both the input query and the collection documents are modeled as a vector of keywords. In our case, we use $tf \cdot idf$ to find the system nodes that best fit to respond to a given cache request. Thus, in our situation the query is composed of the document identifiers requested from the cache, and each node is modeled as the vector of recent document accesses obtained from its ESC summary. Tf in the node vector is computed as the number of times that the node has read the corresponding document recently. Note that AF does not look up the document content to load balance the system. We estimate $tf \cdot idf$ as follows:

[2]In our experiments $tf \cdot idf$ showed a better behavior than other similar approaches as idf alone.

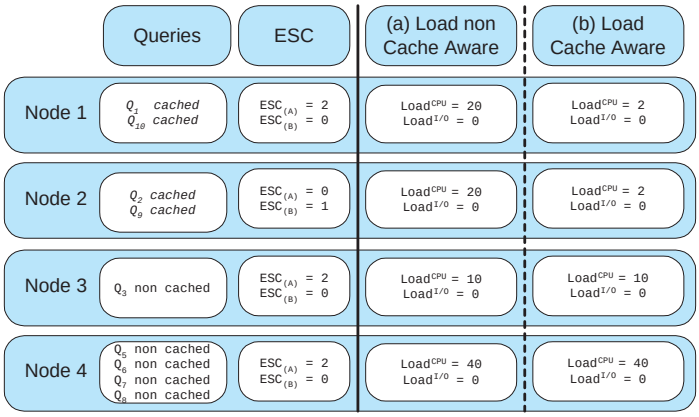

Figure 3: Document A is cached in nodes 1, 3 and 4; document B in node 2. Node 4 is overloaded and is forwarding Q_8, which will access A and B. $C^{CPU}_{(x)} = 10$ for the non cache-aware example, and $C^{CPU}_{HIT(x,i)} = 1$ and $C^{CPU}_{MISS(x,i)} = 10$ for the cache-aware.

$$\vartheta(i,q) = tf_{(ESC_i,q)} \cdot idf_{(q)} = \\ = \sum_{d \in q} ESC_{i(d)} \cdot \log\left(\frac{N}{||\{j \in N | ESC_{j(d)} \neq 0\}||}\right).$$

In information retrieval $tf \cdot idf$ is used because some terms are more relevant for the query than others. Nevertheless, in our case the intuition to use $tf \cdot idf$ is slightly different. On the one hand, the tf determines the number of times that a document has been read. The higher the tf, the larger the probability of the document to be in the cache. In case we had only used tf, the server selection process would be dominated by the popular documents requested by the query, and consequently, infrequent documents would be transferred to the remote nodes with popular documents. Therefore, by only using tf we would replicate rare documents while popular ones would be more centralized. The use of idf aims at solving this problem by giving an estimate of whether a document is rare or popular. Thus, in our $\vartheta_{(i,q)}$ function, rare documents have more weight in the calculation of the affinity score, and so, the chances that they are accessed locally are higher, and consequently fewer replicas are cached. This behavior reduces the global load of the system because the data is available locally, obtaining a more efficient use of the available resources.

3.3 Comparison example

We illustrate the differences between the three main load balancing algorithms proposed (WAL, PC and AF) with an example of a distributed QA system composed of four nodes. Figure 3 shows the system state at a given time: each node has information about the load and the ESC summary of the rest of nodes. In this example, node 4 is overloaded and is going to forward the execution of the query Q_8 to the most suitable node in the network. In order to simplify the example, we assume that (i) the current task of Q_8 does not need I/O ($W^{I/O}_{(Q_8)} = 0$), and (ii) if $ESC_{(x)} \neq 0$, then document x is cached in that node. The picture distinguishes how the cluster computes its current load using a non cache-aware algorithm (WAL) versus a cache-aware (PC and AF). In the former, nodes 1 and 2 report a long execution queue but this is inaccurate because their assigned queries are cached and they will take very short time to be completed. In the latter, nodes 1 and 2 compute a more accurate load because they are cache-aware.

WAL assigns Q_8 to node 3 that it is the "less" loaded node according to its information. However, this is not a good choice because node 3 has to compute query Q_3 that is not cached, and it will take a long time to complete. Although nodes 1 and 2 report longer queues, the queries in these nodes have their data cached in memory and they will finish much earlier than Q_3.

Cache-aware algorithms report a more accurate state of the system load: both PC and AF send a lower load for nodes 1 and 2 because their data is cached. Both nodes, 1 and 2, are missing one of the documents (node 1 is missing A, and node 2 is missing B). Thus, if Q_8 is executed in node 1 document B would be replicated twice in the network (in nodes 1 and 2); if Q_8 is executed in node 2 document A will be replicated in all the nodes of the network. On the one hand, PC sends Q_8 either to the node 1 or 2 because they have the same load, which is the lowest among the available nodes. On the other hand, AF picks node 2 to process Q_8 because document A is very popular and B is not, and consequently, the idf score is much larger for node 2: $\vartheta_{(1,Q_8)} < \vartheta_{(2,Q_8)}$. Although, in this example Q_8 is going to be completed equally as fast either in node 1 or 2, the choice of AF is superior if we look into what happens for the next queries. According to the recent history, future queries are going to request more often document A than document B. For example, suppose the next query that arrives, Q_{11}, only reads document A. In case Q_8 was executed in the node 1 three nodes would have the information locally cached for Q_{11}, but if Q_8 was executed in node 2 all four nodes would have the information locally cached. By the time Q_{11} arrives, the load in each node may have changed drastically, and any of the nodes may be idle. AF ensured that Q_{11} finds all the information cached locally independently of which node is underloaded, PC may need remote accesses if Q_{11} is assigned to node 2 and document A is not already there.

4. EXPERIMENTAL RESULTS

Setup: For our tests we use a fully-fledged QA system running on a cluster of 16 nodes connected with a gigabit Ethernet network. Each node in the system is equipped with an Intel dual core CPU at 2.4GHz and 2GB of RAM. The QA system was explained in Section 2. We use as the textual repository the TREC document collection [16] which has approximately 4GB of text in 1 million documents. The database in our experiments is replicated, and in case a document is not available in cache, each node can load it from its local disk. This strategy simulates a distributed file system tuned for read-only accesses, which is currently not available in our cluster. An additional computer is used as a client that issues each new query to a different computer in a round robin fashion. The question set contains 3000 queries randomly selected following $\text{Zipf}_{\alpha=0.59}$ and $\text{Zipf}_{\alpha=1.0}$ distributions. We choose these distributions as a result of several analyses of query logs from different web engines: the former due to a study from Saraiva et al. [20] where they analyzed

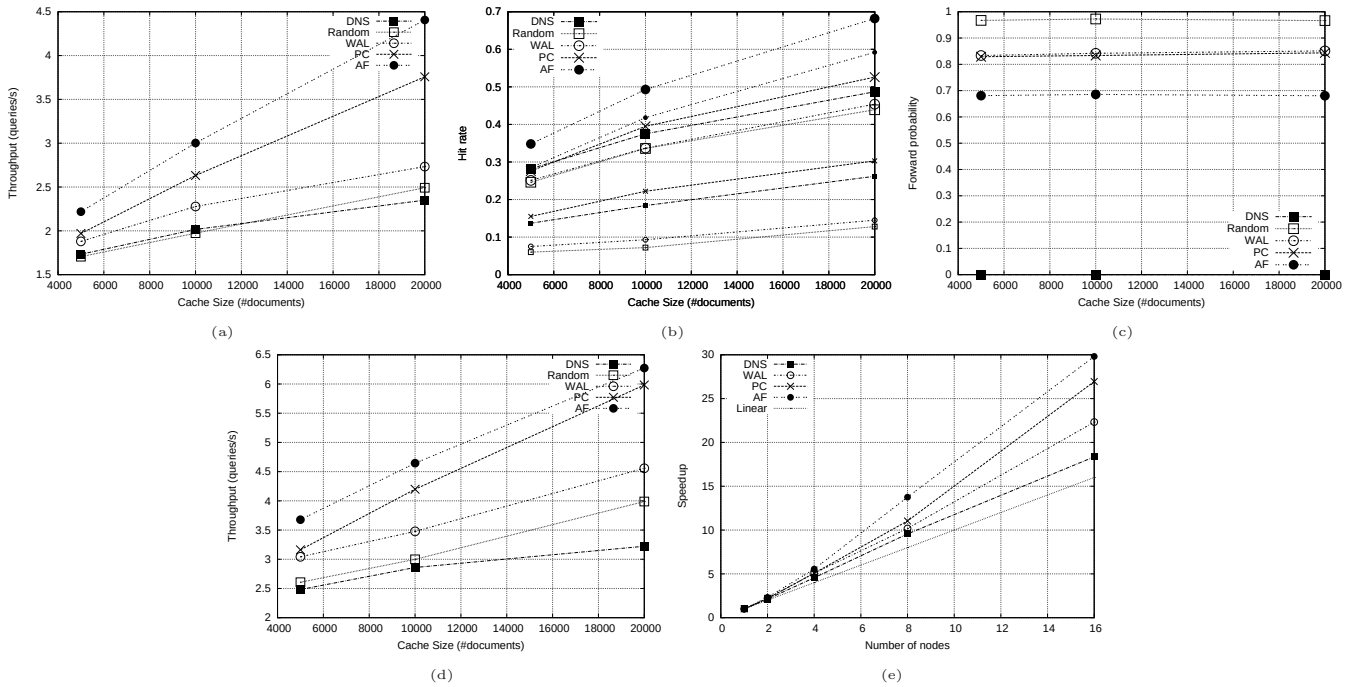

Figure 4: Plots a, b, and c are the results for Zipf$_{\alpha=0.59}$. Plots d, and e are the results for Zipf$_{\alpha=1.0}$. In b, we mark the sum of the remote and the local rate with large marks, and the local hit rate with small marks.

a query log which fitted a Zipf$_{\alpha=0.59}$; and the latter as a sample of more skewed distributions that can be found in other studies such [2, 14]. The questions from the query sets were selected from questions that were part of former TREC-QA evaluations (700 different questions). The client issues the queries to keep the system under a high load, with an average of eight simultaneous queries per node.

The executions shown in the plots of this section include all the times incurred by the different parts of the system and the load balancing algorithms that we explain and test.

4.1 Comparison of load balancing algorithms

In this experiments, we compare the different load balancing algorithms for the two different query distributions explained in the setup.

Distribution Zipf$_{\alpha=0.59}$: Plots a,b,c in Figure 4 show the results for the Zipf$_{\alpha=0.59}$ distribution. In this plot, the horizontal axis is the maximum number of documents that can be stored in the cache. The vertical axis measure the average throughput (a), the hit rate (b) and the probability of forwarding (c).

The first observation is that the simpler policies have poor results: a DNS based approach is not advisable, and Random is not competitive either with policies that are aware of the execution costs. Random, in some cases, has less throughput than DNS because the cost of forwarding a query is not negligible: a transfer forces the system to pack all the documents and data structures related to the query, transfer them through the network, and unpack them in the receiving node. Moreover, Random is not aware of the load in the destination node, so the transfer may increase the system imbalance instead of reducing it. WAL gets a better performance from the system because it combines the usage of the different available resources simultaneously: the accesses to the disks and the CPU time. However, we see that the knowledge of the cache contents is relevant when we use a cooperative cache. Cache-aware algorithms increase the throughput of the system for all the cache sizes tested: it improves the throughput of WAL by 61%, and DNS by 88%.

We depict the hit rate obtained by each algorithm in Figure 4(b). Note that for each algorithm we plot two lines, one with smaller marks and another one with larger marks, for the local hits and the total hits respectively. The difference between the two lines indicates the amount of remote hits in the system. We see the reason why AF gets the best average throughput in Figure 4(a): it keeps a high locality in the accesses to data (its local hit rate is larger than the total hit rate of any of the other algorithms), and it limits the number of replicas of infrequent documents so it can get a better total hit rate. Both of these factors contribute to increase the throughput of the system. Furthermore, Figure 4(b) highlights one of the limitations of the Random and WAL policies: the small local hit rate. Even if a remote hit using the cooperative cache is fast, a local cache access is much faster. Random and WAL do not take into account any locality so the majority of cache hits are remote, and must be retrieved using the network. However, PC is cache-aware and exploits better the cache locality better than non cache-aware algorithms. So, the execution time of PC is faster because the data is more often accessed locally.

The trend for all the load balancing algorithms, as the cache size grows, is to increase the average throughput of the benchmark because the system gets more hits. Nevertheless, the hit rate increase is smaller as the cache grows because we are approaching to the results with an infinite cache.

We also recorded the number of servers that a query visits during its execution. We plot the probability that a query is forwarded when it reaches a scheduling point in Figure 4(c).

We do not observe any influence of the cache size in the number of forward operations, all the algorithms show the same behavior independently of the cache size. We see that among all the algorithms, AF forwards fewer queries than the rest of algorithms. This is because of the locality policy of AF: in the AE scheduling point, the local node has increased the affinity with the current query because it has accessed the documents in PR, and the local node becomes a preferable choice unless it is overloaded. In general, the forwarding rate is high for all the algorithms, which is a sign that the load balancing algorithms contribute to distribute the workload. In a local network, forwarding is not particularly expensive, but if nodes are not in the same local network, the forward rate can be reduced by applying a bigger weight to enforce the processing of the queries in the local network.

Distribution Zipf$_{\alpha=1.0}$: Figure 4(d) shows the results for the execution time but with a more skewed query set. The shape of the results is similar as in Figure 4(a): cache-aware algorithms are significantly better than the rest of algorithms and AF is the best algorithm among all. We observe that the increase in the skewness reduces the execution time of all the algorithms because the caches are more effective for more skewed distributions. We do not include the plot, but the number of nodes visited per query, for the $Zipf_{\alpha=1.0}$ test, is similar to that shown in Figure 4(c).

We have also experimented with the system performance varying the number of nodes in the distributed system. We plotted the experiments as the system speed-up in Figure 4(e). All the tested algorithms behave consistently for the different number of processors and achieve a speed up superlinear, which is a consequence of the use of the cooperative cache. As we add more nodes to the cluster, the total amount of memory dedicated to caching in the cluster grows and consequently the number of cache hits in the cooperative cache increases as well. We see that even if we use no load balancing algorithm (DNS) we obtain a superlinear speedup because of the major efficiency of the cooperative cache.

Although DNS reaches a very good speed up, it creates important imbalances that are stressed when the number of nodes is increased. WAL detects those imbalances and is able to transfer some of the work from the overloaded nodes to the underloaded. However, the performance of WAL can be improved. As predicted by the model, if the load algorithm is not cache-aware it will not take the optimal decision because the information is not complete enough, hence the additional cache information incorporated in AF and PC makes them faster. The imbalance caused by the cache grows with the number of nodes interconnected. For four nodes, the influence of caching in the load balance is almost none: PC and WAL get a similar speedup for four nodes. However, for 16 nodes the execution time of WAL is 20% larger than for PC. Finally, we observe that AF works better than PC even for a 2 node cluster, where it is 5% faster, because AF improves the efficiency of the cooperative cache.

For this query distribution, 16 nodes combined with the biggest cache configuration tested, we reached the highest throughput in the experiments, which is more than 6.25 queries per second (q/s). This corresponds to a speedup of more than 100 over the original system, without a cache system, in a single computer (whose throughput is 0.06 q/s); and a speedup of 38.3 if we consider as baseline the system with cache in one computer (0.16 q/s).

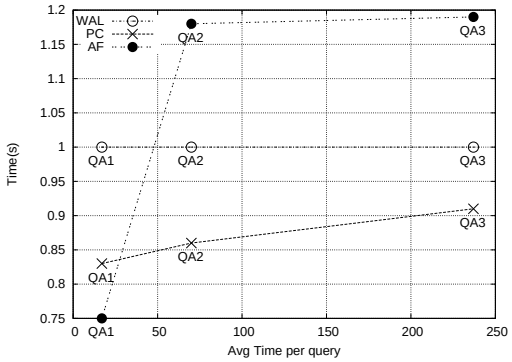

Figure 6: Experiments for different AE modules. Query set follows a Zipf$_{\alpha=1.0}$ distribution

Uneven load vs. performance: In Figure 5 we show the workload (in number of documents that are processed in AE), and the CPU time per each node in the system. We do not show the I/O load because it shows similar patterns. For simplicity, we do not show the random algorithm because, as we showed in the previous section, its performance is very similar to the DNS algorithm. PC and WAL are the algorithms that balance the load more evenly. Here, we confirm that the improvement of PC does not only come from the small increase in the hit rate, but also from the reduction of the idle time of the processors (we measured an average CPU usage of 0.56, 0.63 and 0.65 for WAL, PC and AF respectively).

Through these results, we can reinforce the fact that cache-aware algorithms, although they may introduce a significant imbalance of the workload in the system, they achieve a better overall performance by making a better use of the system resources. This can clearly be depicted in the AF workload and CPU time plots: while nodes in the AF algorithm present uneven peaks of workload compared to WAL and DNS, the CPU time per node is significantly lower because of a better use of the available resources, that is to say, a better load balancing strategy.

4.2 Impact of the Answer Extraction Module

The quality of a QA system depends on all its components, but the dominant factor is the performance of the AE module. A more complex AE analysis of the documents done by the AE module leads to a deeper understanding of the text content and better extracted answers. Although a system slower than the one used in the previous experiments is not usable for interactive searching, in some situations it may be desirable to use a more accurate system but with longer response time.

In this section, we analyze the impact of the increase in the computational complexity of the AE modules. Our objective is to see how effective are the load balancing methods if we have a different QA system, and see how adaptable the load balancing techniques are.

Additional Setup: We change the original natural language processing library (in the AE block) used in the experiments in Section 4.1 from Maximum Entropy (ME) to another library based on Support Vector Machines (SVM). With this change we analyze the tradeoff between speed and accuracy: our SVM classifiers perform better than ME, but

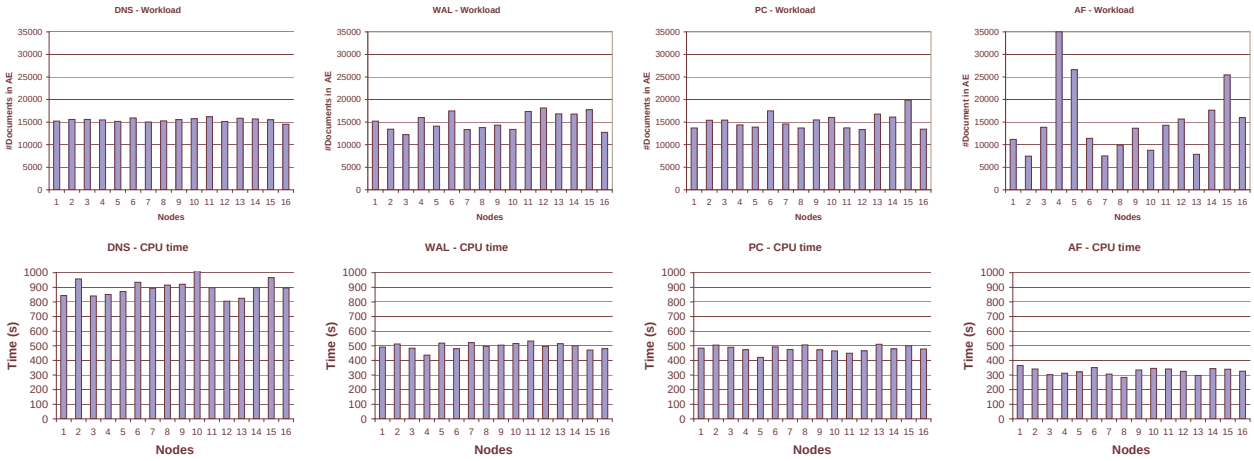

Figure 5: Workload and CPU time consumption for each node in the system. Query set follows a Zipf$_{\alpha=1.0}$

.

they are significantly slower. We build and test three different system configurations:

- QA_1: The base system that we used in the experiments in Section 4. It uses a library based on ME and takes an average time of 17 seconds per query.
- QA_2: The system takes an average time of 70 seconds per query and uses the new SVM library, which implements a medium complexity language model.
- QA_3: The system takes an average time of 237 seconds per query and uses the new SVM library, which implements a high complexity language model.

In case all the data is found in the cooperative cache, either locally or remotely, all the algorithms have the same behavior and take the same execution time because the natural language processing module does not need to be triggered.

Results: We plotted the normalized execution time for each of the QA systems in Figure 6. We use as horizontal axis the average execution time per query (from left to right the points represent QA_1, QA_2 and QA_3). The vertical axis is the normalized execution time. We see in this figures that PC is better than WAL for all the configurations. The addition of cache information reduces the execution time between 17% and 8%, over a non cache-aware algorithm. Actually, QA_3 is much slower than QA_1: the difference between PC and AF for QA_3, measured in seconds, is twice the corresponding difference for QA_1. There are two reasons why the relative execution time of PC and WAL gets closer for heavier workloads as shown in Figure 6: (i) The penalty for each miss is higher in heavier systems, i.e the cost of a miss makes the difference between a local hit and a remote hit negligible. And, (ii) WAL reports the expected load in a node without cache knowledge. In heavy systems the queries whose data is cached stay a tiny amount of time in the node, compared to queries that must be completely processed. Therefore, the noise in the load produced by a cached query disappears relatively faster in heavy systems. Thus, in heavier systems it is less relevant if the algorithm is cache-aware or not.

Even if AF works very well for QA_1, the performance of AF degenerates to disappointing results for QA_2 and QA_3. Although the number of hits is similar independently of the AE module, the very expensive cost of a miss in a heavy system, makes the difference between a local hit and a remote hit insignificant. This situation makes irrelevant the locality of an algorithm, which was the main benefit of AF. We also observe a tradeoff between global hit rates and the load balancing of the algorithms. Figure 5 shows that AF accumulates very large workloads in some nodes because AF believes that they will be executed very fast. However, the ESC values are not updated in real time and the search probabilities are not 100% accurate. Thus, load balancing algorithms have incomplete information and may make a few wrong choices. PC reduces the consequences of incomplete information due to a more even division of the workload. However, AF is more aggressive and creates larger differences in the workload. In QA_1, these imbalances are not very severe but in the heavy case they overload some nodes of the system. In the heavy systems, the AF overloads are not overcomed by the better cache performance, and the system throughput is smaller. Despite the results for heavier systems, AF is a good choice for practical uses of QA because most users are not willing to trade several minutes to get the answers from the system [15] for a small increase in the precision.

5. RELATED WORK

Many applications related to the access of huge data repositories are distributed and need load balancing algorithms [5]. Most of the research on load balancing for general applications has been oriented towards algorithms that balance the load according to a single parameter, which is either the CPU load [24], the number of jobs in the queue of a node [13], the disk I/O [7], etc. However, some other works are closer to our proposals because they combine several of these variables. For example, Surdeanu et al. [22] combine the use of CPU and I/O and propose the WAL formula. Qin et al. [18] add a new parameter to the WAL formula that takes into account the amount of memory needed to execute an application. Andresen et al [1] combine the load in each node with the network cost to forward a task through the network [1], which is relevant for geographically distributed

systems. But to our knowledge, there is no other work that in addition to the CPU and I/O load considers the cache contents in each node and the effects of cooperative caching in the task execution, as we do. Moreover, we use no centralized process that can become a bottleneck in the system.

There are some papers that present algorithms with heuristics to benefit the assignation of tasks that are repeated many times to the same subset of nodes. LARC, developed by Pai et al. in [17], is an algorithm that selects the servers according to a locality policy and the CPU load in each node. However LARC is very different from our proposals: LARC is not I/O aware, it does not consider the impact of cooperative caching, and it uses a centralized process to distribute the tasks. Finally, LARC is not aware if the data is cached in certain node, it only follows a policy that facilitates the caching of a subset of data in a node. A different proposal that takes into account caching and I/O, but not CPU, is WARD by Cherkasova et al. [6]. WARD performs an offline static analysis of the past logs that assigns to each server a subset of the data so that the load will be balanced. However, the analysis is static and the load may differ from the previous log, whereas our proposals, based on ESC, are dynamic and are based on current workload.

In the case of distributed architectures for QA, the only contribution is from Surdeanu et al. [22]. The authors introduced Weighted Average Load (WAL) that takes into account the CPU usage as well as the I/O load in the system. However, WAL does not take into consideration the cache contents because the original system did not use caching. As we have seen above, if cache is enabled our algorithms are faster than WAL for all configurations of our QA system.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown that the overestimation of the query cost originated by the cache hits can produce large imbalances in computationally intensive distributed systems, such as Question Answering.

We propose two algorithms that deal with the imbalance problem and assign tasks considering several factors: the cache contents available in the network, its CPU and its I/O loads. *Probability Cost* showed a significant and consistent performance improvement in all our configurations –for different query sets, number of nodes, cache sizes, query distributions and QA configurations. These results indicate that PC is a good load balancing algorithm for any QA distributed platform. PC performance increases the throughput of the best published non cache-aware algorithm (WAL) for QA from 9% up to 38%, which is proof that the cache contents are relevant for load balancing.

We also introduced the *Affinity* algorithm, which includes a preference for local accesses to data and avoids the replicaton of non popular documents in the cooperative cache. We proved that the throughput of the system increases if the load balancing algorithm considers the cache contents and imbalances the workload to use the caches more efficiently. The results for AF are better than PC for lighter systems where it increases the throughput of WAL by approximately 61%. Although AF works fine only for a limited set of configurations, they are relevant. For example, an interactive QA system that preprocesses most of the natural language analysis, falls into the "light" CPU load category where AF is the best choice. However, the full preprocess of huge, and probably not static, document collections is not always possible as it would require prohibitive computing resources. For such a configuration PC becomes a preferable choice.

Our future work goes towards the design of load balancing algorithms that capture the system stats during its execution, and dynamically pick one or the other load balancing algorithm in accordance to the current system state.

7. REFERENCES

[1] D. Andresen, T. Yang, O. Ibarra, and O. Egecioglu. Adaptive partitioning and scheduling for enhancing www application performance. *JPDC*, 49(1):57–85, 1998.

[2] R. Baeza-Yates. Web usage mining in search engines. In A. Scime, editor, *Web Mining: Applications and Techniques*, pages 307–321. Idea Group, 2005.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.

[4] J. Callan. Distributed information retrieval. *Advances in Information Retrieval*, pages 127–150, 2000.

[5] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. *ACM Comp. Surveys*, 34(2):263–311, 2002.

[6] L. Cherkasova and M. Karlsson. Scalable web server cluster design with workload-aware requestdistribution strategy WARD. *WECWIS 2001*, pages 212–221, 2001.

[7] Y. Cho, M. Winslett, S. Kuo, J. Lee, and Y. Chen. Parallel I/O for scientific applications on heterogeneous clusters: a resource-utilization approach. In *ICS*, pages 253–259. ACM, 1999.

[8] D. Dominguez-Sal, J. Aguilar-Saborit, M. Surdeanu, and J. Larriba-Pey. On the use of evolutive summary counters in distributed retrieval systems. *Technical report.UPC-DAC-RR-DAMA-2008-1*, 2008.

[9] D. Dominguez-Sal, J. Larriba-Pey, and M. Surdeanu. A multi-layer collaborative cache for question answering. In *Euro-Par*, pages 295–306, 2007.

[10] D. Dominguez-Sal, M. Surdeanu, J. Aguilar-Saborit, and J. Larriba-Pey. Load balancing for question answering. *Technical Report UPC-DAC-RR-DAMA-2008-2*, 2008.

[11] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE Trans on Networking*, 8(3):281–293, 2000.

[12] M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC resource management systems: queuing vs planning. *JSSPP*, 2003.

[13] C. Hui and S. Chanson. Improved strategies for dynamic load balancing. *IEEE Concurrency*, 7(3):58–67, 1999.

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

[15] F. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[16] NIST. TREC question answering track. *http://trec.nist.gov/*, 1999-2007.

[17] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. *ACM SIGPLAN Notices*, 33(11):205–216, 1998.

[18] X. Qin, H. Jiang, Y. Zhu, and D. Swanson. Improving the performance of I/O-intensive applications on clusters of workstations. *CC*, 9(3):297–311, 2006.

[19] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *IPM*, 24(5):513–523, 1988.

[20] P. Saraiva, E. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. *ACM SIGIR*, pages 51–58, 2001.

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

[22] M. Surdeanu, D. Moldovan, and S. Harabagiu. Performance analysis of a distributed question/answering system. *TPDS*, 13(6):579–596, 2002.

[23] D. Wessels and K. Claffy. Internet cache protocol: protocol specification, version 2. *RFC 2186*, 1997.

[24] M. Willebeek-LeMair and A. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *TPDS*, 4(9):979–993, 1993.