ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

Analysis and Optimization of Question Answering Systems

David Dominguez Sal

Acknowledgements

This thesis would have not been possible without the intervention of many people, who have inspired many the words in this book. Some have consciously contributed and others may not have been aware. Although the text must follow an order, their importance is not related to it: I believe if any of them were missing, this work would have not been possible. So thanks for being there.

First of all, I would like to thank Josep Lluis Larriba or simply Larri. He has been my director and my friend to look for advice. We have spent many ours discussing and throwing many ideas to the whiteboard, some of which were good and others not so. Thanks for hearing all of them.

I also want to thank Mihai Surdeanu for his help in the development of the thesis. He introduced me to the question answering topic, and the foundations of this work would have not been formulated without him. Although he is now far from here, he has been close either in Barcelona or in the other side of the world. I only miss we cannot discuss our latest ideas around a coffee. I wish him all the luck with his new research and his new little big adventure, Hugo, who will provide as much happy hours as his research.

My family has also played a key role in this small achievement, all my previous ones and the ones coming. Thanks to Jose Luis and Esther who have guided me since I was a child, have loved me all these years, and are the ultimately cause that I am writing this. My sister Marta has also been at my side many years: playing when we were children and still willing to play now. She has helped a lot all these years and this thesis could not be opened without her help. Very special thanks to her for designing the covers of the thesis. I want to also thank my grandparents, Antonieta and Josep, who have taken care of me and have taught me many things about life from their experience which are more valuable than any research.

Marta Perez has also very important in this thesis. She has taught me a lot about how to analyze data and experimental analysis, and the deepest results in the thesis come from her advices. Thanks to Josep Aguilar for his wise advise along the thesis. His experience showed me that hard work eventually become a thesis.

The path is more important than the destination, and in this path I have

met many friends even before I knew I wanted to write the thesis. I want to thank to all the band from Sant Adria (Dan, Vivo, Gonzalez, Marcos, Manu, Fabian, Otero, Tamara, Ana, Monica, Patri...) who meet me as a child, have grown with me and I hope that they continue to get older at my side.

When I started the Phd, a few of my friends of the university began with me. We used to say that we were a lost generation, but I believe we are not lost anymore. Thanks to Edgar, Pere, Juan, Francesc, Leo, Guillem, Ramon... with whom I hope we can continue finding excuses to have a lunch together once our thesis will be finished.

I want to also thanks the DAC department for his support for my research: starting from the basement, all LCAC team, who helped a lot to setup the bondia cluster; to the top floor, where the administrative team has kindly helped with any question I had. Thanks also to the BSC people that we have shared many experiences during lunch time.

Finally, I would also to thank all the people from DAMA-UPC: Arnau, Xavi, Jordi Balasch, Miquel, Vanessa, Sergio, Norbert, Nuria, Pere, Raquel, Robert, Aleix, Victor, Carles, Jairo, Jordi Nin... and to those who used to be in the group but decided to leave. Even thanks to Joan because although he seemed not to help in this thesis (well, who knows...), he has provided many fun hours during the breaktime. To all of them, thanks for all the coffee breaks, the lunch times, the laughs, and the hard time hours shared in the work.

To all of you, and to all who my fragile memory is not able to remember when I am writing this, sincerely thanks.

Agraïments

Aquesta tesi no hauria estat possible sense la intervenció de moltes persones, que han inspirat la majoria de paraules del llibre. Alguns de forma conscient i d'altres sense adonar-se'n. Encara que el text ha de seguir un ordre, la seva importancia no està relacionada amb ell: crec que si manques algun d'ells, aquesta feina no hauria estat la mateixa. Per això gracies per estar aquí.

Primer de tot, vull donar a gracies a en Josep Lluis Larriba Pey, o senzillament Larri. Ell ha estat el meu director i el meu amic per buscar consell. Hem passat moltes hores discutint i llençant idees a la pissara, algunes bones, i d'altres no tant. Gràcies per escoltar totes elles.

Vull agrair a Mihai Surdeanu la seva ajuda en el desenvolupament de la tesi. Ell em va introduir al tema de *question answering*, i els fonaments d'aquest treball no s'haurien pogut començar sense ell. Encara que esta lluny d'aquí, ha estat a prop tant a Barcelona com a l'altra banda del món. Només trobo en falta que no poguem compartir les nostres darreres idees al voltant d'un café. Li disitjo molta sort amb la seva futura recerca i la seva nova petita aventura, Hugo, que espero li doni tantes hores felices com la seva recerca.

La meva familia també ha jugat un paper molt important en aquest assoliment, els meus anteriors i els meus futurs. Gracies a Jose Luis i Esther que m'han guiat des de que era un nen, m'han estimat tots aquests anys i són en última instancia la causa que em porta a escriure això. La meva germana Marta també ha estat al meu costat tots aquests anys: jugant des de que erem nens i encara em ganes de jugar ara. Ella m'ha ajudat molt aquests anys i aquesta tesi no podria haber estat oberta sense la seva intervenció. Moltes gràcies per disenyar les cobertes de la tesi! També vull donar les gracies als meus avis, a la iaia Antonieta i a l'avi Josep, que han tingut cura de mi i m'han ensenyat moltes coses sobre la vida des de la seva experiencia que son més valuoses que qualsevol recerca.

Marta Perez també ha estat molt important per aquesta tesi. M'ha ensenyat molt sobre com analitzar dades i preparar analisis experimentals, els resultats més profunds provenen dels seus consells. Gràcies també a Josep Aguilar pel seu consell des de l'experiencia. Ha proveit una llum a seguir, que mostrava com el treball dur pot esdevenir en una tesi.

El camí es més important que el destí, i en aquest camí he trobat molts amics molt abans de decidir que escriuria una tesi. Moltes gracies a la colla de Sant Adrià (Dan, Vivo, González, Marcos, Manu, Fabián, Otero, Tamara, Ana, Monica, Patri...) als quals vaig conèixer de petit, han crescut amb mi i espero que continuar tenint-los a prop amb els anys.

Quan vaig començar la tesi doctoral, alguns amics de l'universitat van començar amb mi. En cert moment deiem que erem una generació perduda, pero crec que ja hem deixat d'estar perduts. Gracies a Edgar, Pere, Juan, Francesc, Leo, Guillem, Ramon... amb els que espero poder continuar trobant excuses per organitzar dinars un cop hagim acabat les nostres tesis.

També m'agradaria donar gracies al DAC pel seu suport a la meva recerca: començant des de les plantes baixes, a tot el equip del LCAC que em van ajudar molt configurant bondia; fins a la planta alta, on administració m'ha solucionat els dubtes relacionats amb la tesi. Gracies a la colla del BSC amb els que hem compartit les nostres experiencies durant els dinars.

Finalment, vull agrair a tota la gent de DAMA-UPC: Arnau, Xavi, Jordi Balasch, Miquel, Vanessa, Sergio, Norbert, Núria, Pere, Raquel, Robert, Aleix, Víctor, Carles, Jairo, Jordi Nin... i a tots els que han estat al grup pero han decidit marxar. Fins i tot gràcies al Joan perque tot i que no hagi semblat ajudar en la tesi (bé, qui sap...), ha donat moltes estones de diversió durant el temps d'esbarjo. A tots ells, moltes gràcies pels cafes, els dinars, els riures i el temps durs compartits a la feina.

A tots vosaltres, i a aquells que la meva fràgil memòria no ha estat capaç de recordar quan escric això, sincerament moltes gràcies.

Contents

Part I

Introduction

Chapter 1
Introduction

1.1 Motivation

Information Retrieval (IR) deals with the representation, storage, organization and access to information items. The representation and organization of the information should provide the user with easy access to the information in which the user is interested [12]. IR systems have become ubiquitous in our everyday life because of the huge amount of digital information available, thanks to the data digitalization and the Internet. Internet is an enormous collection of data that cannot be categorized manually, and the search for information is only possible when guided by an IR search engine.

Current IR systems deal with giant data repositories: the major search engines crawl more than a trillion unique URLs now [49], and the number will continue to grow. The location of useful information in these huge repositories requires very efficient architectures and algorithms to achieve a good performance.

The details of the architecture in major search engines have evolved with the new available technology and algorithms [35]. However, some fundamental characteristics are latent in their designs: distributed computing and data caching. One single computer is far from achieving the throughput required by major search engines, and the engineers deploy these systems on clusters of computers, often based on commodity hardware [22,74]. Although this architecture accumulates the processing power of several computing nodes, it is not enough to rely on the accumulation of hardware, because the amount of resources needed would become prohibitive. Fortunately, the workload of a search engine typically follows the power law distributions, which imply that some queries appear very frequently and their computation can be reused by future incoming queries. Caches provide a scalable solution that stores the partial computations of popular past queries, and reduce the total computation in the system [10]. The combination of these two characteristics sets a situation that has lead us to the current wide presence of IR systems.

Traditional IR search engines provide searches based on keywords to retrieve a relevant document. Despite the fact that they are indisputably useful, many users' queries target only a very short part of a document, like a paragraph or an entity. However, it is difficult to achieve the desired precision of an answer without a deeper understanding of the document content [2, 8].

Question Answering (QA) is an example of the more advanced features that will be available in future search engines, for example searching for references to named entities. QA systems are search engines with a considerable emphasis on Natural Language Processing (NLP). The input query for a QA system is a question expressed in natural language, like "What country is Aswan High Dam located in?", which allows more expressiveness than a keyword based search. The answer returned by a traditional IR system is a full document where the answer is stated, and the user must read the full document in order to find the desired answer. In contrast, QA systems process the document and just return a precise answer to the query, which typically is one or a few words. For the example above, the answer returned by a QA system would be "Egypt".

QA technology has significantly improved the quality of its answers in the course of the time obtaining notable precision due to the new research induced by the QA tracks of some information retrieval conferences such as the Text REtrieval Conference (TREC) or the Cross Language Evaluation Forum (CLEF) [29, 78]. In a very recent paper, Roussinov et al. showed that current QA systems outperform the precision[1] achieved by major web search engines by more than 50% when users look for queries with short answers [94]. This is a very promising result that indicates that QA is mature and the technology is a useful tool to locate information.

In spite of the good QA precision, Roussinov et al. indicate that one of the major hurdles in the large-scale acceptance of QA systems is their speed and scalability [94]. QA systems are resource intensive applications that require more computing power than that provided by current technologies. The fastest QA systems are at least one order of magnitude slower than their IR counterparts. Therefore, in order to implement QA at the scale of current IR systems, more research is needed to improve the time performance of such discipline.

In this thesis, we address the poor time performance of QA systems. We propose and analyze different techniques related to the cache management of QA systems, either for systems with a single computer or distributed systems. Our work follows an incremental approach: (i) we start from the foundations, studying the cache allocation for each computing block of a QA system deployed in a single computer; (ii) we design a distributed system that implements a new data structure, called Evolutive Summary Counters,

[1] Measured as Mean Reciprocal Rank (MRR).

which is able to distribute summaries of the recent access history in each node of the network: (iii) based on the statistics retrieved from the Evolutive Summary Counters, we propose different algorithms, which are able to transfer and locate the data in a distributed cache; and (iv) finally, we study the impact of caching in the load balance of QA systems, and we propose cache-aware load balancing policies.

1.2 Objectives

This thesis studies how to improve the throughput of a distributed question answering system. We propose and analyze techniques that improve the local cache of a QA system. Besides, we propose new cooperative cache management algorithms that improve the data placement, search and load balancing of a distributed QA system. We summarize the objectives of this thesis as follows:

- To study the impact of caching in a QA system and how to partition the available memory in a computing node.
- To design and analyze a data structure (Evolutive Summary Counters) that allows compact registration and diffusion of the recent data accesses in a distributed system.
- To propose and study algorithms to perform placement (ESC-placement) and search (ESC-search) for documents in a cooperative cache for a distributed search engine.
- To understand the impact of caching in the load balance of a distributed search engine.
- To propose cache-aware load balancing algorithms (Probability Cost and Affinity) that improve the performance of a system.
- To provide a statistical analysis of how the data should flow in a distributed question answering system: query transfer (load balance) vs. cached content transfer (cooperative cache)
- To compare the proposals implemented in this thesis with the state of the art proposals for search engines.
- To contribute to the research community a modular implementation of a distributed QA system.

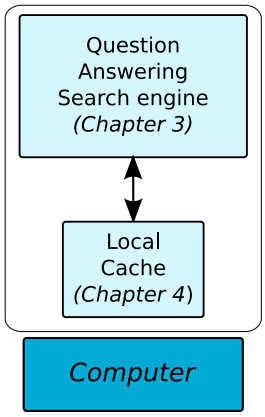


Figure 1.1: *Diagram of the components studied in Part II and chapter where they are first described.*

1.3 Contributions and organization

We divide our work into ten chapters that are grouped in the following four blocks:

- **Part I** introduces the basic concepts related to this thesis and motivates our research.
  - Chapter 1: We introduce the topics of the thesis and motivate the research.
  - Chapter 2: We present the preliminary concepts related to the research within this thesis.

- **Part II** introduces and studies our proposals for caching in the context of a single node QA system. In Figure 1.1, we depict a schema of the blocks described in this part.
  - Chapter 3: We describe our QA system. We detail the internal algorithms of each computing block in the system and give an evaluation of its precision. This analysis was reported in [40].
  - Chapter 4: According to the data processing pipeline of a question answering system, we propose the multi-layer cache policy. We also present an analytical model that demonstrates the usefulness of multi-layer caches, given the typical query distributions and QA data types. Our contribution is in this chapter is the proposal of multi-layer cache for QA. The proposals in this chapter are published in [38].

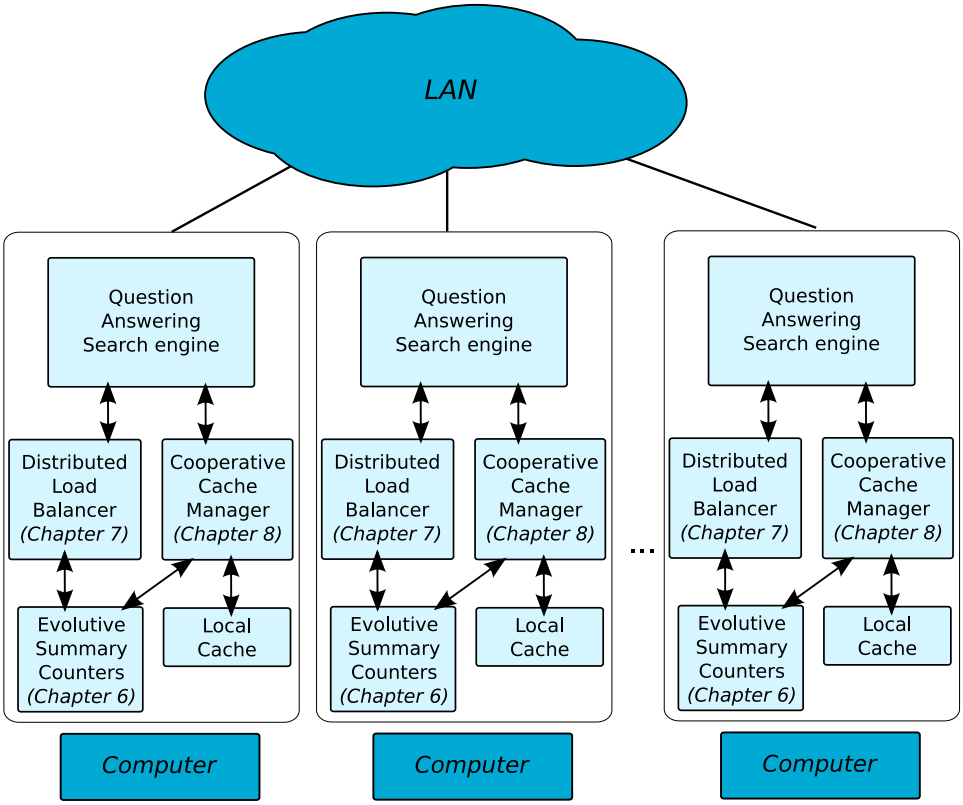


Figure 1.2: *Diagram of the components studied in Part III and chapter where they are first described.*

  - Chapter 5: We analyze multi-layer caches statistically. This analysis covers a wide variety of configurations that provides an experimental validation of the concepts introduced in Chapter 4. Additionally, it gives some hints about the configuration of multi-layer caches. Our contribution in this chapter is the analysis of the multi-layer caches presented in the previous chapter.

- **Part III** describes a distributed version of the single-node QA system introduced previously. Using this system as a platform, we also investigate here several new cooperative caching algorithms. In Figure 1.2, we illustrate a a schema of the blocks described in this part.

  - Chapter 6: We present the distributed question answering architecture, on which we test the distributed proposals of our thesis. In this chapter, we detail the overall distributed architecture depicted in Figure 1.1. Furthermore, in this chapter we introduce the Evolutive Summary Counters (ESC), which is a data structure that records the approximate usage frequency of the doc-

uments accessed in a node. ESC can be efficiently summarized into ESC-summaries, which facilitate the dissemination of the data access trends among the nodes belonging to the distributed system. Our contribution in this chapter is the presentation of the Evolutive Summary Counters data structure. The proposals in this chapter are published in [41] and [37].

- Chapter 7: Using the statistics provided by the ESC-summaries to propose a cooperative caching placement scheme, ESC-placement, which manages the whole contents of the network from a global perspective. Furthermore, we propose and evaluate ESC-search, which is a search protocol that establishes a probabilistic description of the available contents in a distributed system. Our contribution in this chapter is the introduction of ESC-placement and ESC-search for cooperative cache management. The ideas proposed in this chapter are described in [37].
- Chapter 8: We analyze the impact of caches in a distributed system, and we conclude that cache-aware load balancing improves the system performance. In this chapter, we propose cache-aware load balancing solutions that improve the overall cache hit rate and keep the workload in a distributed system balanced. Our contribution in this chapter is the proposal of the cache-aware load balancing algorithms: Probability Cost and Affinity. Our cache-aware load balancing policies are published in [41].
- Chapter 9: This chapter contains a global analysis of the techniques already presented in previous chapters with statistical models. First, we analyze jointly the cooperative caching and cache-aware approaches to determine their benefits and conclude which is preferable for a question answering system. Moreover, we analyze the configuration of the ESC data structure in order to obtain the maximum performance. Our contribution in this chapter is the statistical analysis of the ESC configuration, and the analysis of the overall distributed system performance. This statistical analysis is published in [39].

- **Part IV** sums up the contents of the thesis and future work.
  - Chapter 10: We draw the conclusions obtained for this thesis and present some future work that can be derived from this thesis.

Chapter 2

Preliminary concepts and related work

2.1 Introduction to Question Answering

The Internet has provided a large amount of data, which can be used to locate information instantaneously thanks to current IR technology. A web search engine is able to find relevant documents among billions of websites in less than one second. Users typically express a query to an IR search engine with a set of keywords and some simple operators. Then, the system retrieves the most relevant documents for that query. Unfortunately, a significant amount of information is difficult to locate with current web search technology. Some information requests are difficult to express with keyword based queries, and it is difficult to assess if an answer is valid from the document title and a snippet of a document. In order to locate some web contents, some queries need to be reformulated with complex boolean expressions or operators like excluding keywords until the result is found, which requires both user experience and time. For example, a query such as "Who is the manager of Microsoft?" in a search engine, returns a set of results such as the ones shown in Figure 2.1. The most relevant results correspond to management software, conferences about management sponsored by Microsoft, or Microsoft related sites without information about Microsoft hierarchy. Besides, the list of results returned is a collection of links to the websites. The user must click on the result, load the webpage and read the document if he wants to know if the desired answer is in the result list. Furthermore, users are not usually familiar with complex search expressions to narrow the search, and they are not willing to add query variants to locate the information [100].

Question Answering offers an intuitive query language and precise answers. The QA goal is to locate, extract, and provide specific answers to user questions expressed in natural language [94]. For instance, a QA system that receives a query like "Who won the 2008 U.S. Presidential Election?" an-

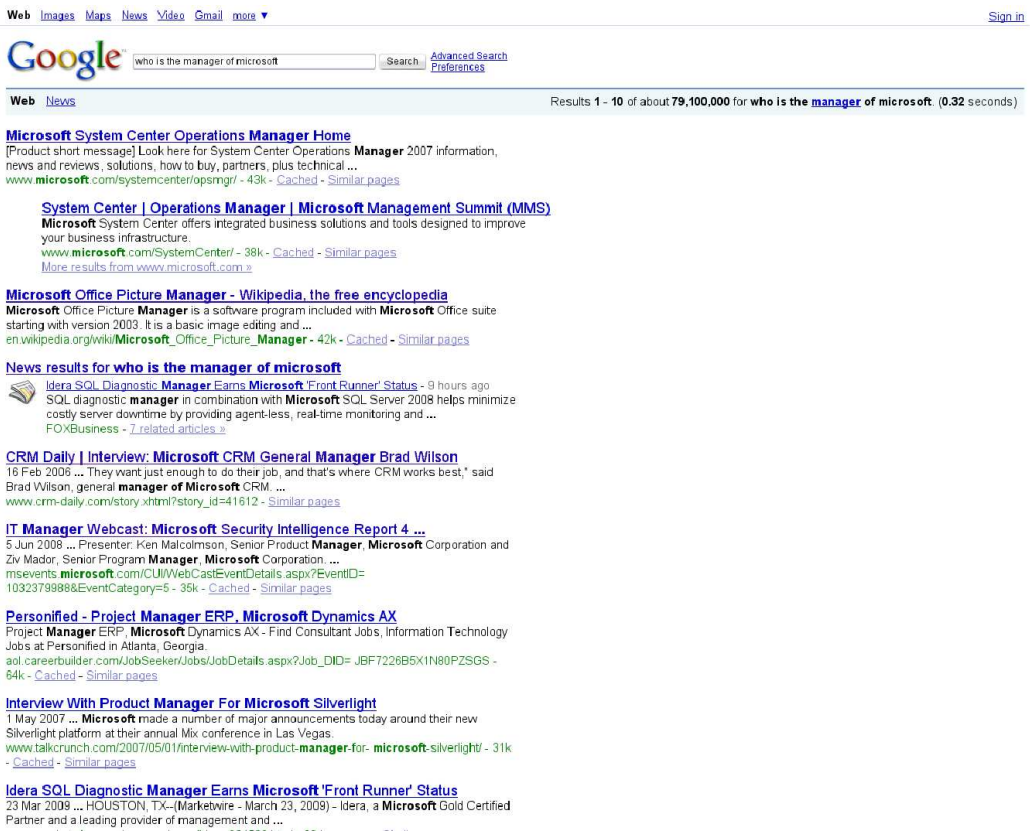


Figure 2.1: *Sample query to a web search engine*

swers "Barack Hussein Obama" or simply "Barack Obama". QA systems in general have many important real-world applications, such as search engine enhancements or automated customer service.

The QA system used as a testing base for this thesis answers *factoid* questions. A factoid question answering system is able to respond queries whose answer corresponds to a concrete and objective answer such as a person name (Eg: Who discovered Penicillin?), a date (Eg: When was Penicillin discovered?) or a location (Eg: Where was the discoverer of Penicillin born?). Other types of questions which are not factoid are, for example, manner questions (Eg: How is Penicillin prepared?) or reason questions (Eg: Why is Penicillin effective against bacteria?).

The architecture typically implemented by state-of-the-art QA systems consists of several components linked sequentially [28, 75, 80]. We show the schema of such a system in Figure 2.2. The output of each computing block corresponds to the input of the following block. A QA system is typically a sequence of the three following blocks:

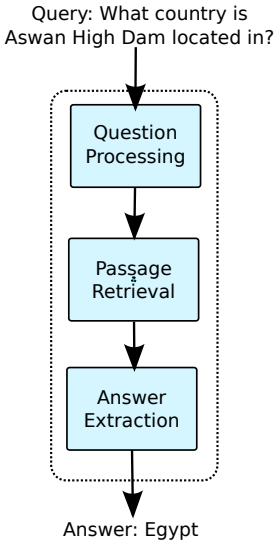


Figure 2.2: *Question answering system architecture*

- **Question Processing (QP):** This computing block is the entry point to the system, which receives the query formulated by the user. The query is transformed from a natural language statement into a computer representation of the query. QP parses the query and extracts the relevant keywords from the user query. The list of keywords is typically weighted or sorted because some keywords are more relevant than others. For example, quoted expressions are classified as very relevant, while stopwords like some prepositions or the verb "to be" may be set with a very low weight or may be removed. QP usually adds variants of the keywords based on syntactic similarity or even synonymy. For example, QP may annotate as a variant of the keyword "manager" its plural form, "managers", or verb forms like "manages" or "managed".

  Furthermore, QP detects the question type. The classification is usually performed by machine learning algorithms that apply statistical information to predict a question type. Examples of answer types are: person names, dates, money amounts, etc. In the later stages of query computation, the question type helps to match the answers returned by the system to the type of answer expected by the user.

  QP also selects *question focus words*, that define the implicit focus of the question. Because of this, they do not typically appear in the document that contains the answer to the query. For example, in "What is the name of the prime minister of France?" the word "name" is a question focus word that appears in the query, but it is not likely to

appear in a document near the answer. In general, the question focus word is annotated as a keyword with small relevance, but it plays a fundamental role in the classification of the question type.

- **Passage Retrieval (PR):** This computing block retrieves the relevant passages from the collection according to the user query. PR internally implements an IR search engine, which retrieves the documents following any of the popular search models: boolean, vector space, probabilistic [12]... QA systems usually do not implement specialized IR engines, and index the collection with the aid of standard IR libraries such as Lucene [68], or become a meta search engine on top of other search engine [116]. The IR search engine scores the documents according to the keywords extracted in QP and retrieves a large set of documents from the collection. This result set from the IR engine is refined in the next steps of the QA system.

  Many PR implementations perform multiple queries to retrieve the candidate documents from the collection. For example, PR may include more keywords for queries with terms that appear in many documents to get more specific results. Or, it may apply query expansion to add variants of the keywords in the query [12], such as adding different verb conjugations. Once the selected documents have been read from the document repository and PR extracts snippets of text, which we call *passages*.

- **Answer Extraction (AE):** The task of the answer extraction module is to detect and select the most adequate answer to the user's query. The detection process is often based on *Name Entity Recognizers* that detect the entities in a text, and those that match the query classification performed by the QP module become candidate answers. The name entity recognizers are often implemented with machine learning techniques such as Support Vector Machines (SVM) or maximum entropy.

  Finally, each candidate answer is scored according to its likelihood of being a correct answer. In this step, the passages are processed with natural language tools that help to understand the meaning of the text and decide if the candidate answer is suitable. The best results are presented to the user as the answers to the query.

2.2 Data Caches and Information Retrieval

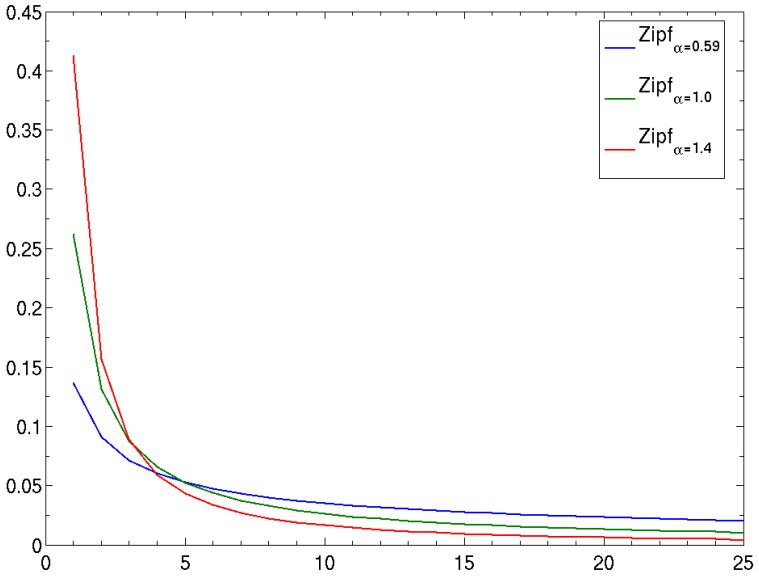
Data caches. Caches are one of the most popular techniques for improving the system performance in computer architecture, because most programs show repetitive trends; in other words, the workloads exhibit locality. The

data locality is usually differentiated into two categories: the temporal locality and the spatial locality. The temporal locality indicates that once a data item has been accessed, it is likely that it will be accessed in the near future. The temporal locality is the base of web proxies that cache the accesses to websites [112]: if a page is downloaded, there is a high probability that more users will access this website in the near future. The spatial locality denotes that if a data item is accessed, it is likely that its nearby data items will be accessed too. An example of spatial locality is the cache in a processor organized by lines. If an instruction is executed in a processor, it is very likely that the next instruction will be read from the following address.

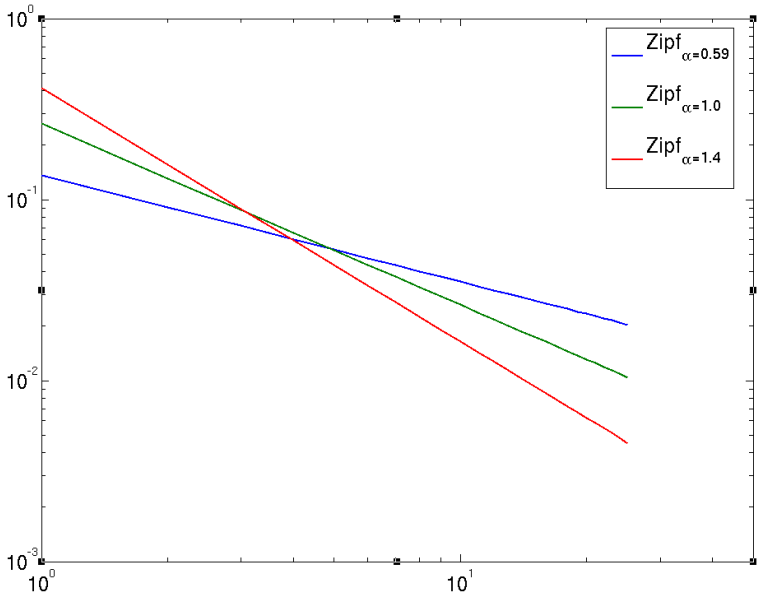
Modern computers implement different caches at different levels of the hierarchy to avoid repeating expensive operations. In this way, while the processor relies on hardware caches to avoid the access to main memory, the applications themselves rely on main memory as a cache for disk. This thesis focuses on caches in the main memory of the computer. These caches mainly exploit the temporal locality of data accessed by programs and keep in the main memory a copy of the data retrieved from the disk, which is a much slower device than memory, and reduce the number of disk accesses.

All applications in a computer benefit from caches in memory, because all modern operating system maintains a *page cache*, which is transparent to the program and keeps a copy of the most accessed pages from disk in the main memory. Nevertheless, applications with expensive computational costs customize the distribution of the available memory for caching to their particular workload. For example, in databases, the engine divides the available memory among the queries running concurrently in the system [102]. It allocates a different amount of memory to each operation of the query plan depending on its complexity: a hash join that expects a large build table receives more memory than a hash join for a small table.

IR workload description. In the field of search engines, caching is necessary to obtain a good system performance, because modern IR search engines compute query results from textual collections orders of magnitude larger than the main memory available [10, 13, 113]. The analysis of query logs from search engines shows that the workload of an IR search engine follows zipfian distributions, which are characterized by their large skew. The zipf distribution is a discrete statistical distribution defined over the natural numbers with a single parameter α. The probability for the i-th element to appear in one sample is proportional to $i^{-\alpha}$. Hence, the probability plot of a zipf distribution depicts a line in a log-log scale of both axes with a descending slope α. In other words, the popularity of items decreases exponentially: the number of accesses to the 10 most accessed items is similar to the number of accesses to the items between the 11-th and the 100-th items for $\text{Zipf}_{\alpha=1.0}$. This effect is illustrated in Figure 2.3(a), where we plot the probability distribution of several Zipf distributions. The most popular



(a)



(b)

Figure 2.3: *Probability distribucion of a zipf law for several values of α in linear (a) and log-log scale (b).*

items are far more popular than the rest, and the set of the most popular elements is a very small portion of the universe. This set of not very frequently accessed elements in a Zipf distribution is usually referred as *the long tail* because of the tail-like shape in the distribution plot. In the plot, we also observe the influence of the parameter α: the larger the parameter α the more skewed is the distribution and thus more elements belong to the long tail. We plot the log-log scale of this distribution in Figure 2.3(b), which depicts a line for all the possible values of α.

There are many papers that characterize the query logs extracted from different search engines, and there is a consensus that the workload of an IR search engine follows skewed zipf distributions with different parametrizations according to the query log [9,17,20,71,99]. In general, the query logs fit Zipfian distributions with a variable parameter α, which is in the range from 0.59 to 1.40, and with typical values below 1.0.

A pattern of accesses following a zipf distribution is adequate for the deployment of caches because it exhibits a high degree of temporal locality, and it is profitable to keep the most accessed documents cached in the system's main memory [3, 11]. Nevertheless, the cache policies must take into account the long tail because most of the documents belong to the tail and they are rarely accessed, with long lapses of time between accesses. According to Belady's cache policy (which is the optimal), the document with the largest time until the next access, or in other words documents that constitute the long tail, must be the next cache victim [18]. We note that membership to the long tail evolves because the query logs are not static over time. Some queries change their popularity over time [51]; a sudden peak in the number of instances of a query is not uncommon [50], and the most queried topics vary throughout the day [17]. Thus, static algorithms are not sufficient, and it is necessary to apply solutions that adapt to the changes in the query distribution [44].

Currently no query logs from real-world QA systems are publicly available. Hence, no analysis for query distributions exists for QA. Because of this, in this thesis, we consider that it is reasonably safe to assume that the knowledge requested by the users may not differ substantially between QA systems and keyword based IR systems, since QA can be considered a subclass of IR. While it is true that the variability of natural language questions is higher than that of keyword-based queries, this variability can be reduced through the detection of paraphrases [42]. Therefore, the experiments in this thesis assume that QA query logs follow zipfian distributions similar to those already published for IR systems.

Caching for IR search engines. An information retrieval system receives a set of keywords, for which it has to search a set of relevant documents. IR search engines cache the two basic data types that they handle during their execution [97]: (a) precomputed answers, and (b) posting lists, which

correspond to the final and partial computation of a query, respectively. (a) The cache of precomputed answers stores the document identifiers returned by the search engine from a previous computation of a query. The search engine maps the sequence of keywords input by the user to the list of identifiers, such as the Uniform Resource Identifier (URI) of the document. Hence, the search engine skips all the computation if a query is resubmitted to the system. (b) IR engines find the relevant documents of a query with the aid of inverted indexes. The inverted index is a collection of *posting lists*, one for each different keyword that appears in the document collection. The posting list for a keyword is a list of documents containing that keyword. In order to locate the documents containing several keywords, the search engine traverses the posting lists associated to each of the keywords in the query, and finds the document identifiers that intersect in all the lists. Search engines keep in memory a cache of posting lists that reduce the I/O of the system.

Caching either precomputed answers and posting lists have important advantages one over the other. On the one hand, answer caches have two advantages: (i) a precomputed answer has a smaller memory footprint than a posting list cache entry, hence more answers than posting lists can be cached in a fixed pool of memory. (ii) If a query finds the answers stored in cache, there is no additional computation for this query, and thus the query is answered immediately. However, if a query only finds the posting lists, the IR system still has to intersect the posting lists of all keywords to find the query results. On the other hand, (i) the posting lists are more versatile, because a precomputed answer is only valid if the upcoming query has been issued before to the system. However, a posting list cache stores partial computations, and the posting list of a keyword can be used by two different queries if they share the keyword.

The IR community has proven that caching is a tradeoff between the two types of cache [10]. In order to get the maximum performance from a keyword based IR search engine, Baeza et al. demonstrated that the IR system must implement the two caches and partition the memory among them. This result for IR is in accordance with our multi-layer cache analysis for Question Answering [38], which is reported in this thesis in Chapter 4.

Question Answering. In the case of question answering, no proposals for cache architectures exist. Some QA systems implement caches for previous answers [65] or take advantage of the caches of IR systems to reduce the passage retrieval execution time [116]. However, previous work by Surdeanu et al. remarks that the computational time of Question Answering is dominated by the reading of the documents from the disks, and the natural language processing analysis of the documents [105]. He quantified the cost of the computational blocks as 30% in PR and 70% in AE, which indicates

that caching for QA should be handled differently from that described for IR systems. In this thesis, we address this problem and analyze the cache architecture for a question answering system.

2.3 Cooperative caching

The computing requirements of some applications surpass the resources available in a single computer. It is necessary to accumulate the processing power of several computing nodes to achieve the desired system throughput and reliability as well as to manage the huge data repositories available. The most popular architecture for this purpose is the *cluster*, which is a set of computers interconnected with a fast network that acts as a single computing device. These services usually facilitate the execution of parallel code and distribute the workload of long computational tasks. The size of a cluster is highly variable and ranges from a few nodes to thousands of nodes. Many applications benefit from cluster architectures: for example, web search engines index and find information in the Internet using clusters [16], and the most popular configurations in the supercomputing area are cluster-based too (82% according to top 500 [73]).

The clusters ultimately rely on the communication between the nodes interconnected and the algorithms that manage the data exchanges. Nowadays, local area networks provide very fast communication interfaces. Gigabit ethernet networks are inexpensive, and there are many alternatives that offer lower latencies and even more bandwidth like Mirynet, Infiniband or 10 gigabit Ethernet. Furthermore, the network technology keeps improving: some hardware vendors already announced routers with 100 gigabit ethernet ports for sale in the coming months [59], and faster networks are expected to be available in the near future.

These fast networks have made it possible to add a new level to the cache memory hierarchy called *cooperative caching* [5]. In a cooperative cache, each node of the network has a pool of memory dedicated to caching data, which can be accessed by the rest of nodes through a network. The cooperative cache gives the illusion to the application that it has a cache larger than that available in one computer because the cooperative cache combines the cache contents available in all the nodes of the network. Furthermore, the cooperative cache policy determines, transparently to the application, in which node a cache content should be stored and how many replicas are available in the network. The cooperative cache can be used to save time from any of the computing resources: it can reduce the number of slow disk accesses [4, 32], store the results of long computing tasks that are executed often [66], or even reduce the network traffic to a central server [45]. With the current fast connections, these algorithms can be applied not only to supercomputers or scientific clusters, but also to distributed servers or desktop

oriented applications, which can take advantage of the available memory in idle nodes present in many local area networks.

We can distinguish two organizations in cooperative caching architectures: hierarchical and non-hierarchical. In the non-hierarchical architecture, all the nodes have equal importance and there are no dependencies among them. An example of non-hierarchical architecture corresponds to peer-to-peer systems. The hierarchical architecture defines dependency relations among the nodes, usually corresponding to a tree. An example of a hierarchical architecture is the implementation of the DNS service, which depicts a tree hierarchy of the different name servers available. Our proposals in this thesis target architectures that have no centralized points, and we therefore focus on distributed cooperative caching for non-hierarchical structures.

2.4 Survey of cooperative caching algorithms

In the next section, we survey the most relevant cooperative caching algorithms found in the research literature, and we describe them according to three dimensions that define the cooperative caching algorithm. (i) In the *data placement* section of each algorithm, we describe the algorithm that distributes the cached contents among the available nodes in the network. In other words, the data placement algorithm decides the number of replicas for each cache entry, and in which nodes each replica is stored. In order to improve the data placement, some algorithms *forward* the cache entries, which is the action of transferring the cached data from one node to another. (ii) The *data search* section refers to how the system locates a data element in the network. Note that search and data placement are different actions: the search operation locates the host node of a document given the document identifier, whereas the data placement operation selects the most adequate target node given a cached document. (iii) The *replacement* section describes the local cache policy, i.e. which entry is evicted from the local cache of a node (and possibly forwarded to a different node) when the available memory of a computer is full.

The techniques reported here, with the three dimensions described, are explained as in the articles where they were proposed. This does not mean that the techniques explained here cannot be combined: for example, the *data placement* algorithm from one section may be implemented with a different *replacement* algorithm. In general, the three dimensions are not orthogonal but have a high degree of freedom. At the end of the chapter, we include a comparison of the main features of the algorithms in Table 2.1.

An important issue for some applications that implement cooperative caching is the data coherence protocols, which can be seen as an additional fourth dimension to the three mentioned previously. The coherence proto-

col guarantees that the serialization order of the read and write operations, when multiple nodes simultaneously access a document, fulfill a set of requirements. These requirements vary among different applications because depending on the environment, the protocols can be relaxed. For instance, Lustre implements read and write coherence if several clients access a file, and hence it needs a cooperative caching algorithm guaranteeing that all read and write operations are immediately visible to all nodes [69]. However, another file system such as the Google File System, which is more oriented to read-only data and appending data to files, does not define a write order for concurrent updates to a file, and thus, the cooperative cache would not need such strong coherence protocols [48].

In the case of question answering, we consider that the collection is a set of read only documents, and we accept the addition of new documents to the data collection. In other words, the content of a document is never modified once it is added to the document collection, and thus no cache coherence protocol is needed. Therefore, we consider that the design of the coherence protocol of the cooperative caching algorithms is beyond the scope of this thesis. The survey also reflects this point, and we do not discuss the cache coherence protocol implemented by each of the algorithms reviewed.

2.4.1 N-Chance forwarding

One of the first practical pieces of work for cooperative caching was performed by Dahlin et al. in [34]. These authors analyze some techniques using the memory of idle computers in a network, which improve the overall cache hit ratio. The work is based on the principle that the access to another computer's memory is faster than retrieving the data from the local disk, so finding the desired data in a remote node will reduce the response time of a system. Among the cooperative algorithms presented in [34], the most beneficial is *n-chance forwarding*. The objective of this algorithm is to extend the LRU of a node by forwarding the victims of the cache to other nodes.

Data placement: The forwarding only takes place when there are no more copies of the evicted entry in the network (otherwise it is discarded). The node that forwards the data chooses the target node at random and sends the entry to it. The receiver node adds the new entry as the most recently accessed in its own cache. Additionally, a counter is kept for each entry that counts the number of times it has been forwarded. When the counter goes beyond a threshold t, the entry is not forwarded and it is discarded. When an entry is accessed in cache (i.e. an entry hit), the counter is reset. According to the simulations performed in [34], there is no significant improvement when the number of forwardings is more than two.

Data search: The location of the entries is done with the help of a distributed tracking strategy. The ids of the documents are hashed and each identifier is assigned to a node of the network. So, each node is responsible for a set of identifiers and must be conscious if, at a certain moment in time, it is possible to find a document in cache, if there is any cached copy in the network, and if so where it is located. In order to maintain the location procedure, the nodes must report the placement changes to the corresponding responsible node. The drawback of this strategy is that if a node crashes or we wish to add new nodes dynamically, then it is necessary to restart the cooperative cache to redistribute the hashes.

Alternatively, we can reduce the maintenance if the network supports multicast. The searches can be resolved by sending a multicast request message to the other caches in the network [70]. Nodes that store a copy of the desired document reply with a positive answer. This protocol is more efficient in case that the number of forwards is high compared to the number of searches.

Replacement policy The local replacement strategy in each node is the well-known Least Recently Used (LRU). LRU eliminates the entry which has its last access earlier in time. This policy can be very efficiently implemented ($O_{(1)}$) with a queue to record the order of the last access to each document, and a small hash to locate the elements in the queue.

2.4.2 Global Memory Management

Shortly after the appearance of n-chance forwarding (see section 2.4.1), Feeley et al. introduced the Global Memory Service (GMS) in [46]. GMS approximates a LRU queue by using a central coordination mechanism, in contrast to the completely distributed placement of n-chance forwarding.

GMS divides the available memory in each node of the cluster into a *local* and a *global* section. The *local* portion stores the locally accessed pages in memory; the global portion, stores the entries forwarded by other nodes. One of the aims of GMS is to use the available memory of idle nodes belonging to the same LAN. In many companies, there is a significant number of online computers that do not make full use of all their resources, while some nodes are performing intensive computing tasks that could be executed faster if more memory were available. GMS tends to increase the *local* portion in nodes executing memory intensive tasks, which increases the number of local hits. On the other hand, GMS increases the *global* portion in idle nodes because this memory can be used by other nodes of the network.

Data placement: The algorithm divides the time into intervals of time called epochs. An epoch starts after a fixed period of time, or when the number of page replacements has exceeded a threshold M. Each epoch has

one node, called the initiator, which calculates the algorithm parameters for the next epoch. Once a new epoch starts, all the nodes in the system send the age of their contents to the initiator. Then, the initiator computes for each node i a weight w_i, which is the fraction of the oldest M pages stored in node i. The computed weight w_i is used for probabilistic forwarding: when a page needs to be forwarded from a node, the destination cache is chosen at random by using w_i as weighting factor. The initiator also computes an estimation of the youngest document that would survive during the next epoch if a global LRU was present; this value is called *minAge*. When an entry is forwarded, if it is older than *minAge* it is automatically discarded. Finally, the initiator selects a new initiator for the next epoch, which is the node with largest w_i.

Data search: The location procedure is similar to the one described for *n-chance forwarding*. It implements a global hash map that, given a document identifier, returns the node where the document is currently cached. The hash map is distributed among all the nodes in the network, and each node stores a portion of it. In order to locate the document, the requester node must contact the node that is responsible for that document.

Replacement policy The local replacement basis is the LRU policy. However, this is influenced by the division of the cache into local and remote pages. A detailed description of the *global* and *local* management is presented in Figure 2.4.

2.4.3 Hint Cooperative Caching

One of the problems common to the previous approaches is the maintenance cost of all the data structures. All the changes in the cooperative memory must be reported and updated, which may constitute a bottleneck for some workloads. Sarkar et al. proposes a search method that uses inexact information (hints) to find the data [98]. Sarkar describes the hint algorithm in a network in which nodes read and write files from a file server. Each node, including the file server, stores a hint list with information about where to find a page, and which is the oldest entry in each node. The local hints are updated each time a new cache entry is obtained from the server or there is a forward from one node to another client. All communications are done point-to-point. The algorithm distinguishes two types of cache entries: master and non-master copies. A master copy is the first copy retrieved from the server to the clients; otherwise, the copies are non-master.

Data placement: The forwarding policy is based on the age hints of each cache: each node has a hint about the age of the oldest entry in each node of the system. The hints are updated each time that there is a forwarding,

The cache entries in the node P are classified into *local* and *global* pages. If P does not have the searched page in the local cache, the following algorithm is executed:

Case 1, *The faulted page is in the global memory of another node, Q:*
The desired page in the global memory of Q is swapped with any page in the global memory of P. Once the desired page is brought into the memory of P, the faulted page becomes a local page, increasing the size of the local memory of P by 1. The local/global memory balance of Q is unchanged.

Case 2, *The faulted page is in the global memory of the node Q, but the memory of P contains only local pages:*
The least recently used page in P is exchanged with the faulted page in Q. The size of the global memory in Q and the local memory in P are unchanged.

Case 3, *The page is on disk:*
The faulted page is read and is stored in the memory of P, where it becomes a local page. The oldest page in the cluster (say, on node Q) is chosen for replacement and is written to disk if necessary. Finally, a global page on node P is sent to node Q, where it continues as a global page (if P has no global pages, the selected page is the least recent local page in P).

Case 4, *The faulted page is a shared page in the local memory of another node Q:*
The faulted page is copied to node P, leaving the original in the local memory of Q. The oldest page in the cluster (say, on node R) is chosen for replacement and is written to disk if necessary. A global page from node P is sent to node R, where it becomes a global page (if P has no global pages, the selected page is the least recent local page in P).

Figure 2.4: *GMS detailed algorithm, adapted from [46]*

hence the age is communicated at the same time as the data is forwarded. Nodes only forward master copies and choose as the target the oldest cache according to the age hints. An improvement of the basic algorithm is the use of a discard cache[1]. The discard cache is located in the main server and stores master copies that were mistakenly replaced by older entries because of the inexactitude of the age hints.

Data search: The search is based on the location hints. The hint list only stores where are the master copies placed. If a node does not have a local copy of a document, the node checks in its hint list for the location of the entry with highest probability and then does the request. If there is no local hint information for the entry, the client requests the hints from the server. If the node receiving the request does not have the entry cached but more recent hints, the hints of the requester are updated. The location process is done recursively until the document is found.

Replacement policy Each cache manages its pool of memory using a LRU policy.

2.4.4 Hash based

Multiple copies of the same data can become hard to manage if we have updates. Moreover, with the new network technologies it is relatively cheap to retrieve information from other computers in a local network. With these premises, Cortes et al. presents the "cooperative cache with no coherence problems" algorithm (CCCP) because the cooperative cache only allows one node to store the data [31–33].

Data placement: In CCCP, there is no forwarding method like in the previously described techniques: the documents are from the first access placed according to their identifier. We have two different versions for this policy: (a) the documents are placed according to a hash which maps them to a determined node [31]. (b) The placement was improved in [33], by the same authors, with more flexibility. Cortes introduced a new type of centralized service, placed in a node of the network, called *repartition server*[2] which collected periodically stats from all nodes, and according to them performs a different data distribution that balances the number of accesses in each node. One of the conclusions in [33], is that redistribution is an expensive task and should be done with care. The distribution access patterns can change quickly over time and produce a lot of document traveling

[1] The concept is similar to a victim cache of a processor in the computer architecture context [58].

[2] The node is not necessarily exclusively dedicated to data redistribution and may perform other tasks such as caching.

that will not be compensated by a better redistribution of documents. In order to prevent this phenomena, [33] presents the *lazy-limited* approach as the best alternative. This consists in not transferring the ownership of a document instantly, but delaying the transfer until it is first used. A second constraint is that a limited number of entries can be transferred in each redistribution. These two rules prevent the situation in which cache entries are constantly moving through the network, possibly overloading it, without being accessed.

Data search: The location of documents in the network is guided by hashing the document identifier. Each document is identified uniquely, and its final placement is determined by the hash of its identifier. So, the location process is limited to calculating the hash and checking which node is responsible for the entry. If the *repartition* process is enabled, the only difference is that the *repartition server* is in charge to send the new assignations periodically to all nodes.

Replacement policy Each node manages its pool of memory by using a variant of a LRU, which they call PG-LRU. PG-LRU differs from LRU in that X% of least frequent elements in cache are marked as the "queue-tip". In [32], Cortes experiments with the queue-tip size and estimates that the best performance is obtained when the queue-tip is about 5% of the total cache size. When the cache is full, if there is an entry in the queue-tip located in the same node as the client that is making the request, it is discarded. Otherwise, the last element of the LRU is erased, as in a regular LRU. This policy is useful when there are multiple processes in the system that have different pool sizes, because it protects clients that are accessing a small set of documents from other clients that are accessing a lot of different documents.

2.4.5 Locality-Aware Request Distribution

Locality-Aware Request Distribution (LARD), proposed by Pai et al., provides a different approach to the placement problem [82]. The objective of this technique is to achieve a high data locality, even if this may lead to the sacrifice of some global hits. Pai's reasoning is based on the locality principle: if a data entry has been processed in one node, subsequent accesses to the same data are likely to be cheaper if the computation is carried out in this node. The architecture of the system is a single entry point, which receives all requests, and sends the query to a node selected from the pool of processing servers.

Data placement: LARD's data placement is heavily influenced by the use of a central coordinator in the system. All requests to the system go

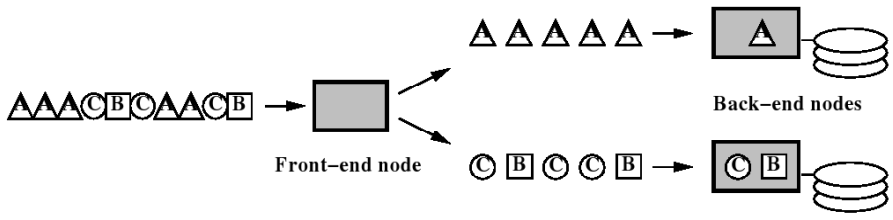


Figure 2.5: *Diagram showing LARD distributing the requests by type to maximize the local hit rate.*

through a single entry point, which forwards clients' requests to the rest of nodes. The coordinator chooses a fixed server depending on the type of request: for example, all the requests looking for the first bytes of a file are always forwarded to one server. This procedure maximizes the probability that the server has the data cached, and reduces the number of duplicates in the network because only one server can cache a determined data item. Figure 2.5 depicts an example of this procedure.

However, the number of requests of a certain type may be enough to overload a server, even if these requests are local hits. In this case, Pai propose to set a maximum load threshold [82]. Once a server crosses the threshold, the following requests can be delivered to a less loaded node. The coordinator establishes a dynamic set of servers that can answer a request for each type of entry. When the initial server is overloaded, more servers are added to the set as requests keep arriving. When the number of requests decreases, the number of servers shrinks. A detailed description of the algorithm is shown in Figure 2.6.

Data search: The nature of this algorithm leads to the central coordinator (the entry node to the system) deciding what node to send the requests, as explained above.

Replacement policy LARD uses *Greedy Dual-Size* (GDS) which works well for web server workloads [23]. GDS associates a value (H_{entry}) to each entry in the cache. New entries in the cache initialize H to the ratio $cost/size$, where the $cost$ is a measure of how expensive it is to load the document into the cache. If the cache is full, the entry with the lowest H, min_h, is evicted; and all the remaining entries reduce the value of H by min_h. When a document is accessed, its H is restored to its original value.

2.4.6 Static placement

Korupolu et al. analyzed the data placement problem from a theoretical point of view in [62], achieving an optimal placement algorithm for a data

```
while (true)
  fetch next request r;
  if serverSet[r.target] = ∅ then
    n, serverSet[r.target] ← {least loaded node};
  else
    n ← {least loaded node in serverSet[r.target]};
    m ← {most loaded node in serverSet[r.target]};
    if (n.load > T_high && ∃ node with load < T_low) ||
        n.load ≥ 2T_high then
      p ← {least loaded node};
      add p to serverSet[r.target];
      n ← p;
    if |serverSet[r.target]| > 1 &&
        time() - serverSet[r.target].lastMod > K then
      remove m from serverSet[r.target];
  send r to n
  if serverSet[r.target] changed in this iteration then
    serverSet[r.target].lastMod ← time();
```

Figure 2.6: *Pseudo-code of LARD with replication.*

set which is accessed following a known distribution and a dataset which is stored in a network with known distances between nodes. In his paper, Korupolu models the system like a tree, where the caches are the leaves and the internal nodes represent the network topology. For example, an ethernet LAN is represented as a root node with all the nodes as children. On the other hand, a system that runs in two different sites, A and B, is a tree of height three where computers in site A are descendants of a common parent a, and computers in site B are descendants of a common parent b; the nodes a and b are descendants of the root of the tree.

Korupolu's solution is a reduction of the placement problem to the Minimum Cost Flow (MCF, [7])[3]. The drawback of this procedure is that it is computationally prohibitive. For a problem with n nodes and m documents, the reduction transforms the placement to a minimum cost flow problem with an input graph of $O(n \cdot m)$ vertexes. The fastest known algorithm for MCF is quadratic, so the cost of finding the optimal placement is $O(n^2 \cdot m^2)$. Moreover, the procedure needs to know all the document frequencies, and the solution is static: it must be rebuilt from scratch if the document frequencies change.

Later in [61], Korupolu analyzes some algorithms that approximate the best solution empirically, but with better computational costs than the optimal algorithm introduced in [62]. We will refer to the algorithm in [61] as

[3]The MCF is a graph problem where each node represents a provider (which puts fluid into the network), a customer (which demands fluid from the network) or a transfer node (a node which represents a link in the network). Each edge represents a pipe of limited capacity where fluid can be transferred, with an associate cost for each unit of fluid transferred. The solution of the problem is a flow distribution in the network that gives to the customers the requested flow with the lowest possible cost (not exceeding either the production of provider nodes or the pipe capacities).

static placement (SP) and the rest of the section is a description of it.

Data placement: The intuition of the algorithm is that it combines three optimization procedures: (a) the most frequent documents are placed where they are used, so they are accessed locally; (b) it tries to keep the unique copies in some node of the network; (c) SP takes into account long network latencies and introduces some duplicates of the documents in nodes that suffer long latencies in the access to documents.

The algorithm follows a bottom-up traverse of the network tree. First, we define $dif_{(d,n)}$ as the access frequency of the document d multiplied by the difference between a cache miss and a local hit from the furthest node in the subtree where n is the root. Initially, SP fills the caches of each tree leaf l with the locally k-most frequently accessed documents. For each cached document d there is an associated value, called $benefit_{(d)}$, which corresponds to $dif_{(d,l)}$. Additionally, all nodes have a value called potential ϕ, which is initialized to 0.

The procedure then targets the parents of the last processed nodes iteratively until we reach the root of the tree. Each node is responsible for optimizing the cache of its descendancy subtree according to the following procedure. For node n, it first picks the documents that have more than one occurrence in the caches of its descendants and selects the entry with highest $benefit$, which is called the *primary copy* of the document. All primary copies, *pc*, increment their $benefit$ by $dif_{(pc,n)}$. Second, each non-cached document,*nc*, sets a new variable, called *value*, to $dif_{(nc,n)}$. The potential of the node, ϕ, is initialized to the sum of the potentials of its children. Once these values are set, the swapping process starts: in each iteration it picks the smallest benefit *primary copy* (*pc*), the smallest benefit cached non-*primary copy* (*npc*) and the highest valued non-cached document (*nc*):

while $value(nc) > min(benefit(pc), benefit(npc) - \phi)$ **do**
  if $[benefit(pc) < benefit(npc) - \phi]$ **then**
    Substitute *pc* for *nc*, setting $benefit(npc) = value(npc)$. The potential ϕ is incremented by $benefit(nc) - value(npc)$
  end
  else
    Substitute *npc* for *nc*, setting $benefit(npc) = value(npc)$. The potential is updated to $\phi = max(0, \phi - benefit(npc))$
  end
end
Add the potential of all the none cached items to ϕ

The process is then executed again in the parent node until the root is reached.

Data search: The placement can put documents anywhere in the network. So, the choice of the location algorithm is completely orthogonal to the placement method, and any location method can be used. Note that the location process is not taken into account in the optimization process done by the placement.

Replacement policy: This algorithm is static, so cache contents are not updated. The only case where caches change their contents is when a new placement of all the documents in the network is built, and all the cache entries are substituted by the new placement of the documents. The rebuild process can be triggered when the miss rate is over a certain threshold, a certain amount of time is spent, etc. But this process should not happen too often as it can change all the cache contents, and consequently it is expensive.

2.4.7 Expiration Age (EA)

One of the recent approaches to coordinated data placement is Expiration-Age [91], by Ramaswamy et al. This is a dynamic placement which takes into account how long is a document expected to live in the cache after its last hit. This policy is intended to transfer the documents that are accessed more than once to the caches where they are expected to be stored longer. EA also encourages high local hit rates, because documents are copied to the nodes where the data are being accessed. Expiration Age can be applied either to hierarchical or flat topologies.

Data placement: the first difference between Expiration Age and other methods is the moment where the system performs the placement algorithm: other dynamic methods do the placement when entries are evicted from caches, whereas in expiration age it is performed when data is retrieved from another node. Each server has an estimator, called the *expiration age*, which is the average time that cache victims have spent in cache since their last hit. The idea of the estimator is to know if the cache is storing a stable set of documents, or if the cache contents are evicted very often. When a server retrieves a document from another computer in the network, they exchange their *expiration ages*. Then, two possible situations can arise:

- The requester has higher *expiration age*: the requester stores a copy of the document and the responder does not update its LRU.
- The responder has higher *expiration age*: the requester does not store a copy and the responder puts the document at the top of its LRU.

Data search: The location protocol in [91] relies on the Internet Cache Protocol (ICP). This is a standard cache protocol to perform queries in a hierarchy of web caches [110]. When a node does not have a document in cache, it queries all its siblings[4]. The siblings reply with a hit/miss message and the requester selects one of them to retrieve the document. In case none is able to satisfy the query, the parent node is asked and tries to resolve the query recursively.

Replacement policy Ramaswamy et al. implement an LRU policy as we mentioned above [91]. In order to calculate the *expiration age* of an entry, the cache adds a field to all documents with the timestamp of their last access. Note that the value of the *expiration age* of a cache is influenced by the local cache algorithm and the estimator may need to be adapted to the algorithm.

2.4.8 Cache Clouds (CC)

Ramaswamy et al. also proposed another algorithm to manage cooperative caches, called Cache Clouds [90, 92]. Similarly to EA (section 2.4.7), CC tries to estimate the benefit of storing a document in the cache, but here the estimator is calculated for each entry and not for the full cache. The nodes are connected with a flat structure, which can be either LAN or WAN. CC has been designed to become an efficient cache for dynamic web content delivery in the Internet.

Data placement: In CC, the decision to store or not to store a document is made when the cache receives the document, either from another cache (remote hit) or from the source repository. In CC, the evicted entries are not forwarded. The decision to store a local copy in the cache of a just retrieved document is taken by a heuristic cost function called *utility*. If the *utility* of a document is over a defined threshold, set by the administrator of the system, the cache will keep a copy of the retrieved document. Otherwise, the cache will send the page to the client without storing a copy of the served data. The formula to calculate the *utility* of a document is:

$$\begin{aligned} Utility_{(d,c)} &= W_{Cop} \times Cop_{d,c} + W_{Spa} \times Spa_{d,c} + \\ &+ W_{Upd} \times Upd_{d,c} + W_{Acc} \times Acc_{d,c}. \end{aligned}$$

The variables of the previous formula are the following: the number of copies of the document in the network (Cop), the disk space needed (Spa), the update frequency (Upd), and the access frequency (Acc). Additionally

[4]ICP is conceived for hierarchical networks. In a flat network all other caches are siblings, and consequently, all will be queried. In this configuration, the protocol is similar to a broadcast of the query to all caches in the network.

the administrator has to combine the weights of the variables adjusting the weights W_x. The weights should be set accordingly to the knowledge of the query distribution, the data accessed, and the hardware of the caches.

Data search: CC uses a distributed location process inspired in the use of consistent hashing [60]. As in [60] the documents in the system are identified by their hash, and each cache in the system maintains the location information of a consecutive range of the identifiers. Nodes retrieve documents by contacting the node responsible for the document and then querying the server which has the desired data. However, in CC the designation of the subrange limits is not guided by a randomized algorithm. At fixed intervals of time, the system starts a balance process where all servers choose a coordinator, and send: the current subrange that each node is managing, the average access rate to the entries in the subrange during the last period[5], and the computing power of the node. The coordinator uses these values to calculate an average access rate for each document, and divides the document range into subsets which will balance the system load. For example, in a network with two computing nodes A and B, with uniform access rate to all documents, and where A doubles the computing power of B, A becomes responsible for the first $\frac{2}{3}$ of the document set, and B for the last $\frac{1}{3}$. Ramaswamy et al., in [92], also describe a two-level architecture for practical implementations: the lower level will work as previously described; the upper level, divides the documents by a static hash. The two-level architecture, reduces the problem of collecting the usage stats of the documents and facilitates a faster redistribution by the coordinator.

Replacement policy: Each cache manages its contents with an LRU queue. As previously described in the placement section, not all the retrieved documents enter the cache but only those which have a high enough *utility*. The optimal *utility* parameter has to be estimated by the administrator of the cooperative cache system.

2.4.9 Locality Aware Cooperative Cache

Recently, Jiang proposed the Locality Aware Cache (LAC) [57]. This technique is based on the concept of the reuse distance to measure the temporal locality of each entry. The *reuse distance* is the number of distinct blocks accessed between two consecutive references to the block. In LAC there is an important correlation between the data placement and the replacement; for

[5]Nodes in the network can collect more precise usage stats by taking the access rate of several document groups instead of a unified value for the whole range. But if we take this to the extreme, nodes will store the access rate of each of the assigned documents and will become too computationally expensive.

a clearer explanation we have changed the explanation order from previous sections.

Replacement policy The replacement policy is based on a queue ordered by the last usage of the document. LAC bases its decisions on the *reuse distance*, which is the time between two accesses to a document. The elements are inserted into the queue after each usage. So, the queue provides an efficient structure to calculate the *reuse distance* in a node, because the position determines in a relative way when the last access of each document occurred. The queue in LAC does not only store information about the elements that are currently stored in memory, but it also stores the identifiers of elements previously deleted from the cache: these elements will be called *shadow* entries. *Shadow* entries do not store the data and take up a much smaller space than a regular cache entry. The documents that are stored in memory can have two different states: *cached* or *forwarded*. Entries added to the queue are set to one of the possible states:

- *Shadow:* If the entry is not in the queue (it is the first access or it left the queue because it had not lately been accessed). Consequently, no document is cached in memory the first time it is accessed. This avoids the problem of polluting the cache with a single pass sequential read of a file.
- *Cached:* If the block was present in the queue and its last access was before the last access of the oldest *cached* document, which in turn, becomes a *forwarding candidate*.
- *Forwarding candidate:* The entry was previously in the queue and its last access is after the last access of the oldest *cached* document. For example, in Figure 2.7, entry G is accessed and its last access is later than the oldest *cached* document (C). So, G is tagged as a forwarding candidate.

Data placement: Data is forwarded when the cache is full. When a *shadow* entry becomes either a *cached* or a *forwarding candidate*, it may be necessary to free memory. Then, the oldest *forwarding candidate* is sent to another node and becomes a *shadow* document. LAC tries to forward the documents to the nodes where the oldest cached entries are stored. All nodes have a global timer. The time is divided into intervals of time, called epochs, which are sequentially numbered. Servers store the epoch of the oldest cached entry in each node, and send the forwarding entry to the node with oldest epoch. If the sender node has the oldest epoch, the entry is discarded. Each server maintains the epoch information of other servers using hints (see section 2.4.3), and the hints are updated when two servers contact each other for a forwarding operation.

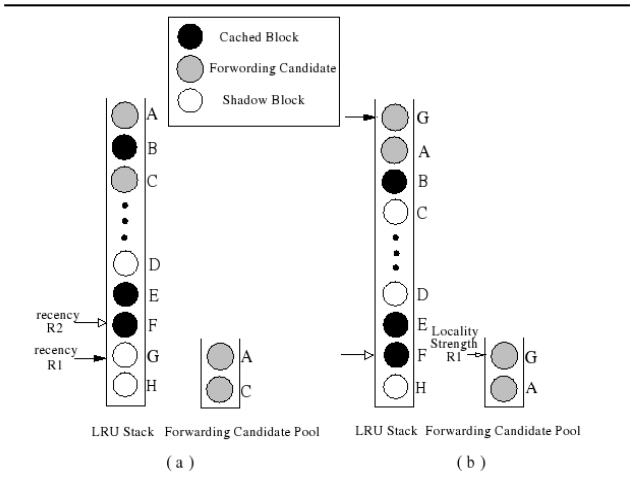


Figure 2.7: *Locality Aware Cooperative Cache. (a) Document G is requested from the cache and is a cache miss; the light arrow points to the last cached document. (b) Once G is read, we store it as a forwarding candidate and C is forwarded.*

Data search: LAC uses the location hint protocol. Each node has a hint list that indicates where a document is with a high probability, and updates the hints when a forwarding is performed. The detailed description of the algorithm is reported in Section 2.4.3.

2.4.10 Distributed Hash Tables

Recently, cooperative caching has been also introduced into databases: Lillis and Pitoura [66] used cooperative caching in an XML database. They applied a distributed hash table (DHT) algorithm to store XML partial results. The intuitive idea behind a DHT is that documents are hashed, and according to the hash they are assigned to a node in a peer-to-peer network with tolerance to addition and removal of nodes. Among the many popular DHT alternatives (CAN [93], Pastry [95], Tapestry [115], etc.) they applied the Chord [101] algorithm. The main drawback of DHT compared to other cooperative caching approaches is that the initial node must contact several nodes to locate a document, which increases the latency of the search.

Data placement: The data is placed according to Chord algorithm [101]. In this DHT algorithm, each data item is hashed to a key with a large number of bits (m), which is typically 160 bits due to the application of the hash function SHA-1. So, each document is uniquely identified by an m bit identifier. The identifiers are mapped as a ring from 0 to $2^m - 1$, which means that 41 and 42 are consecutive identifiers, as well as $2^m - 1$ and 0.

When a node joins the system, it picks a random 2^{m-1} number, which is its position in the ring. The new node is responsible for all the identifiers between the new node location and the next node in the ring (previously in the network). For example, in a network with four nodes at locations 0, 2^{m-2}, 2^{m-1} and $3 * 2^{m-2}$, each node is responsible for the ranges [0, 2^{m-2}), [2^{m-2},2^{m-1}), [2^{m-1}, $3 \cdot 2^{m-2}$) and [$3 \cdot 2^{m-2}$,0) respectively.

Data search: The Chord location procedure finds an entry contacting $O(\log n)$ nodes. Each node in the network is responsible for a subset of the document identifiers, thus the problem of locating a document is to find the the node that is responsible for that document. The location procedure is very similar to a skip list [86]: each node stores a set of pointers to other nodes in the ring that are placed at exponential increasing distance. It stores a pointer to the following node in the ring, the 2nd one following, the 4th one following... up to one pointer in the node on the opposite side of the ring, which makes up a total of log(n) nodes. In order to locate the node responsible for an entry, the algorithm traverses the ring with the aid of the pointers, which reduces the distance between the current node and the target node by at least half of the distance. Therefore, the algorithm contacts at most $O(\log n)$ nodes in a way similar to a binary search.

Replacement policy In this paper, two different replacement techniques are compared: *IndexCache* and *DataCache*. The first stores the queries computed locally in the cache, and the DHT is used as an index that indicates where to locate the document. The second alternative transfers the data to the node that is responsible for that data chunk.

2.4.11 Summary cache

Summary caches is a location procedure introduced by Fan et al. in [45] to alleviate network bottlenecks in a set of proxies implementing the ICP protocol. Each proxy has its pool of memory that is managed locally, and in case the requested data is not available it queries a subset of its neighbours.

Data placement: Data is placed in the cache server that accessed to it. There is no document forwarding.

Data search: Each computing node maintains a count bloom filter that represents the locally cached contents. When a document is added to the cache, the corresponding entries for the document in the count bloom filter are increased. When a document is evicted, the counters are decreased to reflect the cache removal. Periodically, each node broadcasts its bloom filter to the rest of nodes to update its locally cached contents. The location

procedure first checks if the data is available locally in the computing node; otherwise it tries to locate it remotely. The search is guided by the summary received from each node. The server only queries those nodes were the latest summary indicates that the document was cached.

Replacement policy The LRU policy is applied in each node.

2.4.12 Broadcast Petition Recently

Broadcast Petition Recently is a cooperative caching algorithm proposed by Dominguez-Sal et al. oriented towards clusters in local area networks that access to a document collection following a skewed distribution [38]. The algorithm monitors the latest requests to detect the documents that are very frequently accessed in a node, and creates a local replica of these documents to avoid the network penalty of remote accesses. BPR corresponds to the first cooperative caching algorithm implemented for a question answering system, which proves that it is possible to achieve a superlinear speedup for these search engines.

Data placement: Data placement is performed with the document requests: there is no forwarding of cache victims. Each node stores a list of the document identifiers requested in a recent fixed window of time (for example, in the last 30 seconds), which is called *recent document list*. Each new document request is added in this list with its corresponding timestamp. The documents not requested remotely in this time window are removed from the list.

Documents retrieved remotely are not introduced into the local cache by default. The system only stores a local copy of the remotely accessed documents if this document is currently listed in the recent document list. When a document is retrieved from a node, the document provider updates its local cache and adds the requested document as the least recently used. This procedure reduces the replication rate of documents that are not the most accessed in the system, while keeping in the cooperative cache the documents accessed by any node of the distributed system.

Data search: The location algorithm of BPR consists in a broadcast of the document identifiers not found for the current query in the local cache. The nodes that have an available copy of the requested document can send it to the requester node. This algorithm is a simplification of the ICP [110] when caches do not constitute a hierarchy and all the caches have a sibling relation.

Replacement policy The replacement algorithm implemented is the multi-layer caches described in Chapter 4 of this thesis.

2.5 Load balancing

Many applications related to the access of huge data repositories, such as search engines, are distributed among several computing nodes and need load balancing [24]. However, most of the research on load balancing algorithms is focused on modeling applications which only require one type of resources, which is not the case of question answering.

Load balancing algorithms typically consider the CPU usage because, historically, this was the bottleneck of many applications, and still is in many computationally expensive distributed programs. In general, the load balancing algorithms monitor the CPU usage in each node (or only neighbouring nodes depending on the architecture) and estimate the cost to process a task in a node. Then, a task is forwarded to a different node, if the receiving node is less loaded and keeps a more balanced load among the nodes in the system [52, 111]. A more flexible approach to the CPU-based algorithms is the job-based techniques, because they can be applied to non CPU bounded applications. For example, Hui et al. propose a load balancing algorithm, where each node has a queue of pending tasks, and a new task is sent to the node with the least tasks queued [55]. However, job-based techniques are not adequate for environments where the cost to compute a task is not uniform because they might accumulate many expensive computing jobs in a single node. The round robin policy is also a job-based load balancing strategy, which is often implemented in many distributed servers as an initial distribution of the work that is complemented later with other load balancing algorithms [24].

On the other hand, there are applications that access large data repositories, which do not fit in main memory, such as large database systems or scientific computation applications [36]. In most of these applications, the memory wall between the disk and the main memory is the bottleneck of the system, and the load balancing algorithms are adapted to improve the I/O of the system. Typically, an I/O load balancing algorithm monitors the disk usage of the system and evaluates if remote execution might be profitable, which is a similar schema to that previously described for CPU load balancing algorithms. One example of I/O load balancing is IOCM. It monitors the disk load in each node of the network and forwards a task if the task is expected to be completed earlier in a remote node, considering its current state plus the network costs of transferring the task [88]. Parallel applications which need large I/O bandwith may also implement data stripping, which distributes the data blocks that will be read consecutively in different computing nodes, and hence allow for the parallelization of the read/write operations on disks. One early implementation of this technique is in the Zebra file system, which distributes the blocks of each file in multiple servers following a round robin policy [53].

Given that many applications do not consume a single resource, we also

find a few techniques that combine the use of multiple resources. For example, Surdeanu et al. combine the use of CPU and I/O and propose the Weight Averaged Load (WAL) algorithm [105]. WAL takes two independent measurements in each node of the system: one for the CPU load and another for the I/O load. Then, the queries are assigned to the nodes least loaded according to a formula that weights the CPU and the I/O load in each node. Although we describe WAL in more detail in Chapter 8, its general idea is that the queries which are expected to use CPU will be assigned to nodes with low CPU load, and the queries which are expected to use I/O will be assigned to nodes with low I/O load. Qin et al. extended WAL for applications with large memory footprints, and considered the memory usage of the application as a third type resource, in addition to CPU and I/O [89]. So, they add to the WAL formula a new term that indicates the memory usage of a node. Moreover, Andresen et al. combine the CPU load in each node with the network cost to forward a task through the network [6], which is relevant for geographically distributed systems.

In relation to the cache implications of load balancing, there are some articles that introduce heuristics to benefit the assignation of frequent tasks to the same subset of nodes. LARC, described in the previous section, is an algorithm that selects the servers according to a locality policy and the CPU load in each node [82]. Unfortunately, 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. [26]. 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. 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 discuss later in Chapter 8 of this thesis.

Additionally, in the general literature of load balancing, the algorithms can either be implemented as non-preemptive or preemptive [63]. The former indicates that once a task starts, it cannot be moved to another node until the next step (which in some cases may mean the completion of the task). On the other hand, a preemptive load balancing implementation allows for the relocation of a task in the middle of its execution. In this thesis, for experimental simplicity we will implement our techniques as non-preemptive, though all our proposals can be implemented preemptively.

Regarding distributed architectures for QA, the only contribution from the literature is the already mentioned WAL algorithm, which 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, as we do in this thesis, because their question answering system did not implement caching features. Given that WAL is the best previous technique for QA, we will consider WAL as the baseline against which to compare our proposals.

2.6 Summary and conclusions

This chapter has been devoted to the description of related publications to the main topics of this thesis. We have described the caching architecture of a modern information retrieval search engine, and the previous research related to performance for question answering systems. We also provided a survey of the most relevant cooperative cache management algorithms published. Finally, we include a summary table in which we compare the previously described proposals.

2.6.1 Table summary of cooperative caching algorithms

We review the main characteristics of the cooperative caching algorithms described in this chapter in Table 2.1. The characteristics under comparison for placement are the following:

Centralized: It indicates if any step of the algorithm requires a centralized process.

Failure tolerant: In case a node of the network crashes (or a new node is added), the system can continue working without a cache reorganization.

Forwards victims: It is affirmative for those algorithms that try to relocate the cache victims of the replacement policy.

Manual configuration: It denotes algorithms having some parameters that must be tuned by a system administrator.

Randomized algorithm: It denotes algorithms in which the placement of a document relies on some random generated number, or an amortized algorithm such as a hash policy.

Replicates: The algorithm replicates cache contents in more than one node of the network.

And for search:

Access remote data: The algorithm searches for data in a different node from the requesting one.

Centralized: It denotes that the search correctness depends on a centralized server, at least for some cases.

Family: We classify the search algorithms into four families.

- Direct: The algorithm does not need to contact any node to locate the information.
- Broadcast: The algorithm queries all the nodes in the network.
- Hint: The algorithm relies on imprecise information of the documents available in the cooperative cache.
- DHT: The algorithm uses a distributed hash table to locate the contents.

Fixed node : Given a document identifier, the node responsible for the management of the document is fixed.

Single step : It indicates that the algorithm contacts all the nodes involved in the search in one step, possibly in parallel.

Placement	N-Chance	GMS	Hints	Static Hash	LARD	Static	Exp. Age	Cache Clouds	LAC	DHT	BPR
Centralized	No	Yes	Yes	No	Yes	Yes	No	No	No	No	No
Failure tolerant	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes
Forwards victims	Yes	Yes	Yes	No	No	No	No	No	Yes	No	No
Manual configuration	Yes	No	No	No	Yes	No	No	Yes	No	No	Yes
Randomized algorithm	Yes	Yes	No	Yes	No	No	No	No	No	Yes	No
Replicates	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	Yes
Search	N-Chance	GMS	Hints	Static Hash	LARD	Static	Exp. Age	Cache Clouds	LAC	DHT	BPR
Access remote data	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Centralized	No	No	No	Yes	N/A	Yes	No	No	No	No	No
Family	Direct	Direct	Hint	Hash	N/A	Direct	BCast	DHT	Hint	DHT	BCast
Fixed node	No	No	No	Yes	N/A	Yes	No	No	No	No	No
Single step	No	No	No	Yes	N/A	Yes	No	No	No	No	Yes
	N-Chance	GMS	Hints	Static Hash	LARD	Static	Exp. Age	Cache Clouds	LAC	DHT	BPR

Table 2.1: *Comparison of cooperative caching algorithm*

Part II

Caching for Question Answering Systems

Chapter 3
Question Answering

In this thesis, we implement a complete Question Answering system capable of answering open natural language factoid queries in English. The system follows a typical QA architecture divided into three blocks: question processing, passage ranking and answer extraction. In this chapter, we detail the internal implementation of these computing blocks and evaluate the precision of the final system.

3.1 Question Processing

Question Processing (QP) is the first component of the QA system. It receives a query from the user and extracts the query target and converts the query into a flexible set of queries that a keyword based search engine can compute.

3.1.1 Question Classifier

The question classifier detects the query type. The first step in the classification is to perform a part-of-speech tagging of the query, as shown in Figure 3.1. Then, the detection of the query type follows a machine learning approach.

The system maps each query received to a two-level taxonomy consisting of 6 question types and 53 subtypes, which is shown in Table 3.1. This taxonomy is inspired by Xin and Roth [114]. Nevertheless, our classification mechanism is different: instead of using a hierarchy of $6 + 53$ binary classifiers (one for each type and subtype), we opt for a single Maximum Entropy multi-class classifier that extracts the best tuple <`type:subtype`> for every question. We choose the single-classifier design because it significantly improves the classification response time, which is a paramount requirement for any interactive system. We compensate for the possible loss of accuracy with a richer feature set. Formally, our question classifier assigns a question

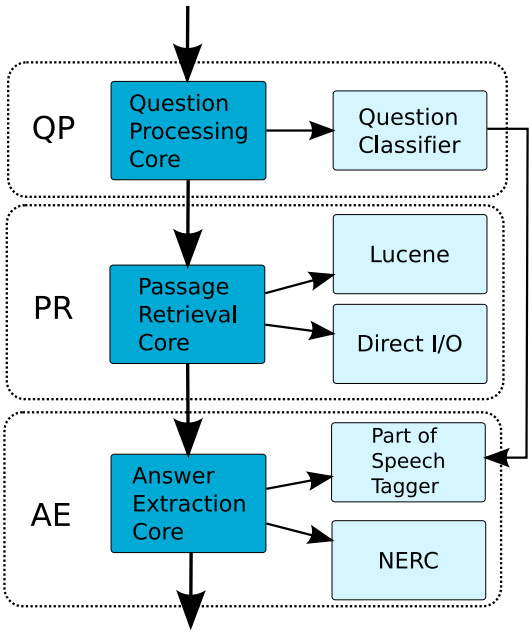


Figure 3.1: *Detail of the internal implementation of each computing block.*

type	subtype
ABBREVIATION	abbreviation, expression abbreviated
ENTITY	animal, body organ, color, creative work, currency, disease, event, food, instrument, language, letter, other, plant, product, project, religion, sport, symbol, system, technique, equivalent term, vehicle, special word
DESCRIPTION	definition, description, manner, reason
HUMAN	group, individual, title, description
LOCATION	city, country, mountain, other, state
NUMERIC	angle, code, count, date, distance, money, order, other, period, percent, speed, temperature, size, weight

Table 3.1: *Question Processing categories recognized by the QA system*

$\phi_{\mathbf{sequence}}(\mathbf{x})$
foreach($x_i \in \mathbf{x}$) add features:
w(x_i), l(x_i), **sem**(x_i), **prox**(x_i),
w(x_i) · w(x_{i+1}), l(x_i) · l(x_{i+1})
foreach($c \in$ **sem**(x_i), $c' \in$ **sem**(x_{i+1})): $c \cdot c'$
foreach($c \in$ **prox**(x_i), $c' \in$ **prox**(x_{i+1})): $c \cdot c'$
$\phi_{\mathbf{qfw}}(\mathbf{x})$
add features:
w(qfw(**x**)), l(qfw(**x**)), **sem**(qfw(**x**)), **prox**(qfw(**x**)),
w(x_0) · w(qfw(**x**)), w(x_0) · p(qfw(**x**))
foreach($c \in$ **sem**(qfw(**x**))): w(x_0) · c
foreach($c \in$ **prox**(qfw(**x**))): w(x_0) · c

Table 3.2: *The feature extraction functions for the question classifier.* w - *token word,* l - *token lemma,* p - *token POS tag,* **sem** - *set of semantic classes (from [114]) that contain this word,* **prox** - *set of proximity-based word sets (from [67]) that contain this word,* qfw - *the QFW detection function.* · *stands for string concatenation.*

class – i.e. tuple <`type:subtype`> – to each question, using the function:

$$qc(\mathbf{q}) = \underset{c \in \mathcal{C}}{\arg\max}\ \text{score}(\phi(\mathbf{q}), c) \tag{3.1}$$

where **q** is the sequence of all the question words, e.g. {"What", "is", "the", "Translanguage", "English", "Database", "also", "called"}; $\mathcal{C}$ is the set of all possible question classes; score is the classifier confidence; and ϕ is a feature extraction function. ϕ is computed as a composition of several base feature extraction functions (all of them detailed in Table 3.1):

$$\phi(\mathbf{q}) = \phi_{sequence}(\mathbf{q}) + \phi_{sequence}(\mathbf{h}) + \phi_{qfw}(\mathbf{q}) \tag{3.2}$$

where **h** is the sequence of heads of the basic syntactic phrases in the question, e.g. {"What", "is", "Database", "called"} for the above example, $\phi_{sequence}$ extracts n-gram features from a sequence of words, and ϕ_{qfw} extracts features related to the Question Focus Word (QFW). A QFW is a type of word that appears in queries and helps to formulate a query in natural language, but does not typically appear in a snippet of text that contains the query. It emphasizes one aspect of the question and helps to make the type of answer to the query more precise. For example, "database" is the QFW of the previous example, and "city" is the QFW in "What city is the capital of Tuva?". QFW is usually the head of the first noun or verb in the question, skipping stop words, auxiliary and copulative verbs. We have implemented the detection of the QFW with 7 surface-text patterns, which are reported in Algorithm 1.

In the system, there is a one-to-one mapping from each question type to the category of the expected answer, which is detected later by the Named Entity Recognition and Classification (NERC) module from AE. For example, the tuple `LOCATION:country` for a query entails that the answer type

Input: Query
Output: Question Focus Word
// Match the query to one of the following patterns
switch *Query Pattern* **do**
  case *What/Which <be> name/type/kind of/for <QFW>...?*
    // Eg: ``astronaut is the QFW in ``What was the name of the first Russian astronaut to do a spacewalk?''.
  end
  case *What/Which type/kind of <QFP>...?*
    // Eg: ``bridge'' is the QFW in ``What type of bridge is the Golden Gate Bridge''.
  end
  case *Auxiliary verb ... <QFW-mainverb>?*
    // Eg: ``located'' is the QFW in ``Where is Kyzyl located?''.
  end
  case *Who/What <be> <skip-phrase> <QFP>...?*
    // Where the skip phrase is one of the following: ``considered to be'', ``known to be'', ``known as'', `` ''. Eg: ``father'' is the QFW in ``Who was considered to be the father of psychology?''.
  end
  case *How <ADJ-QFW>...?*
    // Eg: ``tall'' is the QFW in ``How tall is the giraffe?''.
  end
  case *The first NP headed by a non-stop word or the first non-auxiliary VP*
    // This rule captures most of the What questions not identified by the previous patterns such as: ``state'' in ``What state has the most Indians?''. Also, questions where the QFW is a verb inmediately following the question stem, e.g. ``wrote'' is in ``Who wrote the Farmer's Almanac?''.
  end
end
if *The selected QFP is an NP followed by a possessive followed by another NP* **then**
  Select the second NP as the QFW;
  // Eg: The fourth pattern marks the QFW of the question: ``What is California's state tree'' as California. This post-processing step moves the QFP to the correct phrase ``state tree'', which yields the correct QFW ``tree''.
end

Algorithm 1: Question Focus word rules

is a named entity of type LOCATION. Because the NERC module used in the thesis extracts only 3 types of entities (Person, Location and Organization), we select questions whose answers correspond to those types. We map question types to answer types using the following mapping:

Answer Type	type:subtype
PERSON	HUMAN:individual
LOCATION	LOCATION:{city, country, mountain, other,stated}
ORGANIZATION	HUMAN:group

Note that the three selected types make answer extraction more difficult. There are two main reasons: (a) our answer types are difficult to identify. For example, [106] report F-measure differences of more than 16 points higher for Money entities than Organization ones. Moreover, [106] show that Person and Organization types figure among the hardest types to recognize. (b) The considered answer types yield more answers (in most corpora). For instance, [106] show that the number of location names in the Switchboard corpus are six times more frequent than the money entities [107]. If the number of entities is high then it gives the system more options to choose from and consequently increases the probability of error. These two characteristics make the question types answered by our search engine representative of the complexity of QA systems.

3.1.2 Keyword generation

The system generates an ordered set of keywords. The algorithm associates to each keyword a priority that is used by the passage retrieval block. Priorities are assigned solely on the basis of their part of speech (POS) tags and lexical context. All non-stop keywords are grouped in descending order of priority as: (1) words that appear within quotes, (2) proper nouns, (3) numbers, (4) contiguous sequences of nouns and adjectives, (5) contiguous sequences of nouns, (6) other adjectives, (7) other nouns, (8) verbs, (9) adverbs, (10) the QFW, (11) other words. For example, for the question "What is a measure of similarity between two images?", the set of sorted keywords extracted by this algorithm is: {"two", "images", "similarity", "measure"}.

We perform keyword expansion on verbs. The QA system stores a list made up of the bare infinitive, present, past, gerund and past participle forms of 3,500 different verbs. Each verb in the query is expanded with all the previous tenses. The expansions are taken as alternative forms (and equivalent to the original) in any of the following computing blocks.

3.2 Passage Retrieval

Our passage retrieval algorithm is inspired by the query relaxation algorithm of [84], which adds or drops query keywords depending on the quality of the information retrieved. Furthermore, our implementation is capable of adjusting not only the set of keywords used, but also the proximity between the keywords.

The retrieval algorithm consists of two main steps: (a) in the first step all non-stop question words are sorted in descending order of their priority, and (b) in the second step, the set of keywords used for retrieval and their proximity is dynamically adjusted until the number of retrieved passages is sufficient.

In the second step, the actual passage retrieval is implemented using the algorithm described in Algorithm 2. The set of keywords **K** is initialized with all keywords with priority larger than the priority assigned to verbs, and the current proximity is initialized with some default value (20 words in our experiments). The algorithm is configured with four parameters: $MinPass$ and $MaxPass$ – lower and upper bounds for the acceptable number of passages (currently 5 and 1000), $MinProx$ and $MaxProx$ – lower and upper bounds for keyword proximity (currently 20 and 40 words).

The actual information retrieval (IR) step of the algorithm (step (1) in Algorithm 2) is implemented using a boolean IR system that fetches only passages that contain *all* active keywords in **K** at a proximity $\leq p$. Passages do not have a fixed size, but rather they extend as long as there are keywords whose distance is smaller than proximity p. We implement this first step with the aid of a public library IR library, called Lucene [68] (see Figure 3.1), and a small library (implemented by us) to read the text of the documents from the hard disk using direct I/O (see Appendix B). Lucene indexes a collection of documents and is able to retrieve efficiently the set of document identifiers that contain a set of keywords. Although Lucene is also able to store the text of the document, we do not use this functionality because in this thesis we want control the I/O produced by the reading of the documents. Direct I/O is a special mode in which the I/O of the system bypasses the caches of the operating system. This mode employs the usual operating system calls, but it gives full control of the I/O buffers to the program without the interference of the operating system.

3.3 Answer Extraction

The Answer Extraction (AE) is a sequence of two steps: (i) first, it identifies candidate answers from the relevant passage set according to the expected answer type extracted in QP; and then, (ii) it ranks the answer(s) according to their relevance to the formulated query.

Input: Keyword set with priorities
Output: Set of passages
(1): Retrieve passages using keyword set **K** and proximity p;
if *Number of passages* $< MinPass$ **then**
  if $p < MaxProx$ **then**
    Increment p;
    Go to step (1);
  end
  else
    Reset p;
    Drop the least-significant keyword from **K**;
    Go to step (1) ;
  end
end
else if *number of passages* $> MaxPass$ **then**
  if $p > MinProx$ **then**
    Decrement p;
    Go to step (1) ;
  end
  else
    Reset p;
    Add the next available keyword to **K**;
    Go to step (1) ;
  end
end
return *The current set of passages*

Algorithm 2: Passage retrieval algorithm

The NERC component is fundamental in QA systems because it recognizes the candidate answers for a given query. Answers not recognized by the NERC module are missed as possible solutions, reducing the QA system precision. We have two implementations that recognize entities in a text, both of which are able to recognize persons, locations and organizations. They work as external libraries as depicted in Figure 3.1. The first is an off-the-shelf NERC module, yamcha [64], which is distributed as a binary library. Yamcha implements a classification algorithm based on support vector machines. Additionally, we have implemented a second classifier based on maximum entropy, which implements the set of features described in [106] in order to train the machine learning algorithm. Both implementations require that the input text is preprocessed with a part-of-speech tagger. For this task, we also use a public library called TnT [109], which is based on a Markov chain model.

The ranking step scores each candidate with a combination of several heuristics [84]. The heuristics correspond to simple patterns based on the

density and the position of the keywords. All these heuristics do not require elaborated additional natural language processing, but a basic tokenizer. The following are the six heuristics implemented by the system:

(H1) *Same word sequence:* computes the number of words that are recognized in the same order in the answer context;

(H2) *Punctuation flag:* 1 when the candidate answer is followed by a punctuation sign, 0 otherwise;

(H3) *Comma words:* computes the number of question keywords that follow the candidate answer, when the later is succeeded by a comma. A span of 3 words is inspected. The last two heuristics are a basic detection mechanism for appositive constructs, a common form to answer a question;

(H4) *Same sentence:* the number of question words in the same sentence as the candidate answer.

(H5) *Matched keywords:* the number of question words found in the answer context.

(H6) *Answer span:* the largest distance (in words) between two question keywords in the given context.

H1 looks for sentences in the document collection with a high similarity with the query formulation. H2 and H3 aim at the identification of appositions that may contain the answer of the query. The last three heuristics (H4, H5 and H6) quantify the proximity and density of the question words in the answer context, which are two intuitive measures of answer quality. For each candidate answer we combine the heuristics in the following formula:

$$score = \mathbf{H1} + \mathbf{H2} + 2\mathbf{H3} + \mathbf{H4} + \mathbf{H5} - \frac{1}{4}\sqrt{\mathbf{H6}}. \tag{3.3}$$

The weight of each heuristic was adjusted to optimize a training set of 200 queries, as reported in [84].

The set of candidate answers are sorted by *score* and the top results are returned to the user in two forms: the entity itself, and a short snippet of text around the selected entity.

3.4 Evaluation of the Question Answering System

Setup: We evaluate the system output with the aid of TREC-8 resources (which provides a document collection, a set of questions, and its corresponding set of answers) and we compare our results with those scored by the top participants in that edition. In TREC-8, each system was allowed

	Single entity	Text snippet
Ideal QP	0.37	0.52
Full system	0.33	0.47

Table 3.3: *MRR of the question answering system.*

to output a ranked list of answers, in which an answer was a text snippet with up to 250 characters, for each question. The quality of the answer was judged as the Mean Reciprocal Rank (MRR) score, which measures if the system is able to retrieve a correct answer in the first positions of the generated ranking: it assigns a score of $\frac{1}{k}$ to each question, where k is the position of the correct answer, or 0 if no correct answer is returned. An answer is considered correct if the text snippet contains the correct answer for that question. In TREC-8, the score of a system was measured as the average MRR score for the whole question set. An ideal QA system, which always returns the correct answer in the first position, would score 1, and one that is not able to find the correct answers would score 0. In addition to the evaluation procedure of TREC-8, we also measured the MRR of our system for outputs that are limited to a single entity (which is typically one or a few words). This second score is smaller than the 250 characters one because the amount of text given by the system is shorter and hence improves the difficulty of the task.

Experiments: We ran two configurations of our system, which we report in Table 3.3: the first configuration features a system where query classification was manually assigned, and the second was fully automated. The difference between this two configurations evaluates the quality of the QP classification. The QP module is critical because if the query is misclassified, the system will not be able to locate adequate entities to answer the query. Nevertheless, we observe that the QP system is very accurate and there is a small difference between the fully automated system and the manual classification.

We complement our experiment results showing the MRR of our system for the two previously described evaluation criteria in Table 3.3: text snippets (like in TREC-8) and single entities. This experiment tests the quality of the AE, as well as, the overall system precision. We observe that the MRR of the complete system evaluated like in TREC-8 (0.47) is significantly above the average of the TREC-8 participants, which was 0.35 [108]. This indicates that the algorithm implemented in AE is good because it is able to identify the most relevant passages for a question. Besides, the single entity evaluation benchmarks the quality of our rules identifying the relevant answer in a passage. Regarding this aspect, our result indicates that the precision of the system is good. The MRR for text snippets indi-

cates that the answer appears on average in position 2.12 ($\frac{1}{0.47}$), which is less than one position far from the average position for single entity answers ($\frac{1}{0.33} = 3$).

All in all, the results for the full system, evaluated as text snippets like in TREC-8, is 0.47. This result scores in the top 5 participants of TREC-8 whose highest results were 0.64, 0.54, 0.51, 0.48 and 0.47 [108]. Furthermore, it is significantly above the average and the median of the TREC-8 participants, which were 0.35 and 0.38 respectively.

Since the aim of the TREC-8 evaluation was the precision of QA systems, the execution time of the systems (and its hardware setup) was not gathered together. However, as already mentioned, the individual algorithms for each computing block of our QA system figure among the fastest of those presented in the QA literature because we do not implement very NLP complex analyses, and thus, we consider that the speed of our system is representative of a fast QA system. The measured throughput of the overall system with the yamcha NERC is 0.03 queries per second, and 0.05 queries per second for the maximum entropy implementation. Regarding the time spent by the program in each computing block, we found that for both of our configurations more than 98% of the execution time is spend in PR and AE. For the maximum entropy library, the system spends 72% of the time in PR and 26% in AE; and for yamcha, 29% in PR and 70% in AE. This performance breakdown of a QA system is in accordance with the analysis by Surdeanu et al. [105], which reports that most of the execution time is spent in PR and AE, with a significant fraction of time in both modules. In this thesis, we run our experiments with the maximum entropy library, unless we specify otherwise.

The base system with minor modifications has also been evaluated in several international evaluations where it obtained state-of-the art results [40, 47, 104]. Hence, our question answering system is able to achieve results close to the state-of-the-art systems, and we consider it is a good testbed to test the new optimizations for next generation search engines described in the rest of this thesis.

3.5 Summary and conclusions

In this chapter, we have introduced the typical architecture of a question answering system, which is composed of a sequence of computing blocks: question processing, passage ranking and answer extraction. Question processing analyzes the user's query formulated in natural language and extracts the most relevant keywords for the query. Additionally, it also detects the valid answer types for the query. Passage retrieval reads from the document collection the most relevant documents and filters a set of relevant passages that are processed with natural language processing algorithms in answer

extraction. Finally, answer extraction detects and ranks the most relevant answers to the query.

The chapter details our implementation of a question answering system that follows this architecture. We evaluate our QA implementation and obtain results in the state-of-the-art in recent question answering evaluations [40, 47, 104]. The question answering system presented in the chapter corresponds to the test environment for the new algorithms developed in the following chapters of the thesis.

Chapter 4

Caches for question answering systems

Caching is a fundamental technique to improve the performance of a system. In this chapter, we propose and analyze multi-layer caches. This cache organization is able to store different data types and targets applications made up of several computing blocks, which have expensive computational costs. The chapter starts with the description of the multilayer cache, exemplified by its application to question answering systems. Then, a theoretical analysis is presented of the cache and finally a validation of the model is given. The analysis of multi-layer caches extends to the next chapter, in which we take a broad empirical analysis to discover which are the most adequate scenarios to deploy a multilayer cache and how to configure them.

4.1 Multi-layer Caches for Question Answering

Many search engines are structured as a sequence of computing blocks that receive as input a data collection, process the items, and select a subset of them according to an evaluation function. Each new computing block performs a deeper analysis to reduce the number of candidate answers with the objective to improve the precision of the system. This architecture is observable in question answering where we find Passage Retrieval (PR) and Answer Extraction (AE), as explained in the previous chapters. In the former, the application selects a subset of documents from the full document collection; in the latter, the output of PR is processed with natural language tools, it is ranked, and only a small list of results is given to the user.

In each new computing block, the data size grows because each computing block adds more refined information to the result from the previous block. This requires seeking a tradeoff in the partition of the memory dedicated to caching the results from each of the computing blocks. On the one hand, the last computing blocks of the system process larger data entries,

because it accumulates the information from the previous steps. On the other hand, a cache entry from the last computing blocks contains data processed at a deeper level of the system pipeline, and that potentially reduces more the amount of computation.

Our multi-layer approach reserves a memory pool for storing the partial results of each computing block. The cache is divided into several layers with one layer for each different processing component that has intensive resource requirements. Given that processing is cumulative, each cache entry that appears in one of the layers stores the result for that layer, and for all the layers corresponding to previous execution phases.

In the actual case of QA, the system has two computing blocks (PR and AE) and therefore we allocate a cache layer for each of this blocks. Figure 4.1 depicts the architecture of our QA system extended with multi-layer caching. As the figure indicates, we add a cache manager that handles all the operations related to caching to the original QA system. The cache itself is divided into two layers: the first is dedicated to caching PR results and the second is used to cache AE data. The PR layer caches only the raw text (*r-documents*[1]) retrieved from disk. The AE layer stores processed documents (*p-documents*), which are the retrieved raw-text documents as well as their syntactic and semantic analysis. Then, a cached document is stored either as an r-document in the PR layer, or as a p-document in the AE layer, but not in both layers simultaneously. Nevertheless, note that the architecture is inclusive, i.e., a p-document will yield a cache hit for both AE and PR operations.

The fact that the caching unit is a document instead of a question increases the probability of a hit because: (a) the same natural language question can be formulated in many distinct forms, but will most likely require the same documents, and (b) the documents needed by questions that are different but on the same topic are likely to overlap significantly.

As shown in Figure 4.1, the two layers are managed by the same cache manager, which implements all local cache operations. We employ different policies for intra and inter layer operations:

- For *intra-layer* operations, which control the internal management of a layer, we use a least recently used (LRU) algorithm. The intra-layer policy aims at maintaining the most accessed documents in cache for as long as possible.

- For *inter-layer* operations, which address the communication between the two layers, we use a promotion/demotion algorithm. When an r-document is read from disk for the first time it is promoted to the PR

[1]We currently use as the indexing and caching unit one TREC document, because TREC documents are reasonably small (4KB on average). However, for larger documents this method can be immediately adapted to a strategy that indexes/caches smaller units, e.g., for HTML documents we can index and cache static passages (<p>).

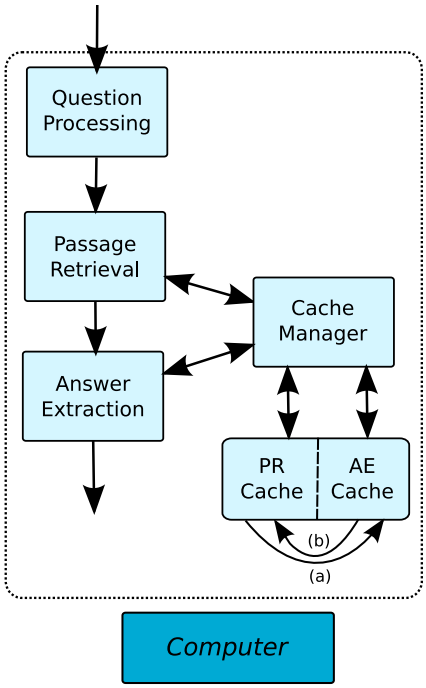


Figure 4.1: *Diagram of the three sequential modules of our QA system: Question Processing, Passage Retrieval and Answer Extraction. The cache manager stores the r-documents retrieved from disk in PR, and the p-documents generated in AE.*

layer. Then, if the PR cache is full, the oldest r-document is demoted from PR (i.e., it is completely removed from the cache). When we miss a p-document in the AE layer but the r-document exists in the PR layer, we promote the entry to the AE layer (operation (a) in Figure 1). Then, if the AE cache is full, the oldest p-document is demoted to the PR layer (operation (b) in Figure 4.1). When the cache demotes a p-document from the AE to the PR layer the syntactico-semantic information is deleted, and it becomes an r-document. This inter-layer policy keeps the entries in the AE cache alive for as long as possible because they are more expensive to generate than the corresponding entries in the PR layer.

This two-layered cache adapts to the specific characteristics of the QA task: (a) the computational time spent by PR and AE adds more than 95% of the execution time of QA system; (b) the PR component accesses a larger number of documents than the AE module – due to the filtering at the end of PR only a subset of the documents accessed in PR reach AE; (c) the p-documents in the AE layer save more execution time than r-documents because their data is used in both the PR and AE phases; but

(d) a p-document is significantly larger than the corresponding r-document (approximately five times larger in our setup). We show in the next sections that, due to these facts, both cache layers are necessary to reach optimal performance.

4.2 A Model for Two Layer Local Caches

The performance of a multi-layer cache depends on how the available memory is divided among the cache layers. Depending on the problem, i.e., time spent on each unit and its request frequency, some layers should be allocated more memory than others. The optimal partitioning maximizes application performance, or, in other words, minimizes the application response time. Note that, even though we model a two-layer cache because QA has two resource-intensive sequential components, a similar reasoning could be applied to caches with more layers.

We model the application performance with a cost-based formula:

$$\begin{aligned} TotalTime \;=\; & RD \cdot \left[H^{PR}_{s_{PR}} \cdot C^{PR}_{Hit} + \left(1 - H^{PR}_{s_{PR}}\right) \cdot C^{PR}_{Miss} \right] + \\ & PD \cdot \left[H^{AE}_{s_{AE}} \cdot C^{AE}_{Hit} + \left(1 - H^{AE}_{s_{AE}}\right) \cdot C^{AE}_{Miss} \right] \end{aligned} \tag{4.1}$$

where $TotalTime$ indicates the overall application response time for a given input[2]. The total time is defined as the sum of execution times for each phase of the system: the first row in the formula accounts for the PR block and the second row for the AE block. RD is the number of r-documents processed in the PR layer per question, and PD is the number of p-documents processed in the AE block per question. C^{PR}_{Hit} and C^{PR}_{Miss} are the average cost (measured in time units) to process a cached and a non-cached r-document in the PR phase; C^{AE}_{Hit} and C^{AE}_{Miss} are the costs to process a cached and non-cached p-document in the AE block. $H^{PR}_{s_{PR}}$ and $H^{AE}_{s_{AE}}$ are the probability of hitting the cache contents for the PR and AE layers, which store a total number of s_{PR} documents and s_{AE} passages. We compute these two probabilities later in this section.

We assume our system has a finite amount of memory, which can allocate up to T documents. So, if each p-document has a size δ times larger than an r-document ($\delta \geq 1$), because the p-document includes the natural language analysis of the document), we can write the following relation between s_{PR} and s_{AE}:

$$T = s_{PR} + \delta \cdot s_{AE} \tag{4.2}$$

The probabilities $H^{PR}_{s_{PR}}$ and $H^{AE}_{s_{AE}}$ are related variables because the total memory in the system is limited. Each document in the collection is accessed by following a probability distribution Φ. We assume that the most

[2]We remove QP from the model, because its execution time is negligible compared to the rest of the QA system.

frequently requested documents are cached. If we sort the documents by access frequency, the hit rate corresponds to the cumulative distribution function of Φ, which we call $F_{(x)}$. The equation below models the execution time, taking into account the amount of memory dedicated to each layer:

$$\begin{aligned} TotalTime &= RD \cdot \left[F_{(T-\delta \cdot s_{AE}+s_{AE})} \cdot C^{PR}_{Hit} + \left(1 - F_{(T-\delta \cdot s_{AE}+s_{AE})}\right) \cdot C^{PR}_{Miss} \right] + \\ & \quad PD \cdot \left[F_{(s_{AE})} \cdot C^{AE}_{Hit} + \left(1 - F_{(s_{AE})}\right) \cdot C^{AE}_{Miss} \right] \end{aligned} \tag{4.3}$$

Note that a document found in the cache can be either in the PR layer (with size $T - \delta \cdot s_{AE}$) or in the AE layer (with size s_{AE}). So, the hit rate in the PR layer, $H^{PR}_{s_{PR}}$, is $F_{(T-\delta \cdot s_{AE}+s_{AE})}$, because the two layers contribute, and the hit rate in the AE layer, $H^{AE}_{s_{AE}}$ is smaller ($F_{(s_{AE})}$), because only the AE layer contributes.

We can analytically obtain the portion of cache dedicated to p-documents which minimizes the total execution time. It corresponds to the solutions to the derivative of $TotalTime$ with respect to the free variable s_{AE} equal to zero:

$$\frac{\partial TotalTime}{\partial s_{AE}} = 0.$$

This equation can be solved given a query distribution. We focus on zipfian query distributions because they are the typically found in the query logs of search engines. In a Zipf distribution, the probability of requesting a document, d_i, from a collection is exponentially distributed according to a parameter α: $\Phi_{Zipf(d_i)} \sim 1/i^{\alpha}$. Then, we set the query distribution to Φ_{Zipf}, and we study the function according to the cache partitioning of the cache.

For Zipf functions ($\alpha > 0$), the $TotalTime$ function is U-shaped when varying the proportion of cache between AE and PR[3]. Intuitively, this means that the minimum execution time is not neither when we have only an AE cache or a PR cache, but for an intermediate partitioning: a part of the cache for PR and a part for AE, improves the performance.

In Figure 4.2, we plot the execution time depending on how we divide the memory between the layers for a typical QA distribution. We show the normalized value of the $TotalTime$ function in the vertical axis, and the horizontal axis represents the fraction of the cache dedicated to the AE layer (the other fraction is dedicated to PR). We plot several charts: each one represents a different QA system where AE uses more complex models (hence more time consuming). We see that, for all systems, the smallest response time is achieved by dividing a certain percentage of the cache for p-documents and the rest for r-documents. Comparing the different systems, the optimal configuration tends towards storing more p-documents as the AE

[3]Except if some phase is overwhelmingly more expensive, which is not the case for QA.

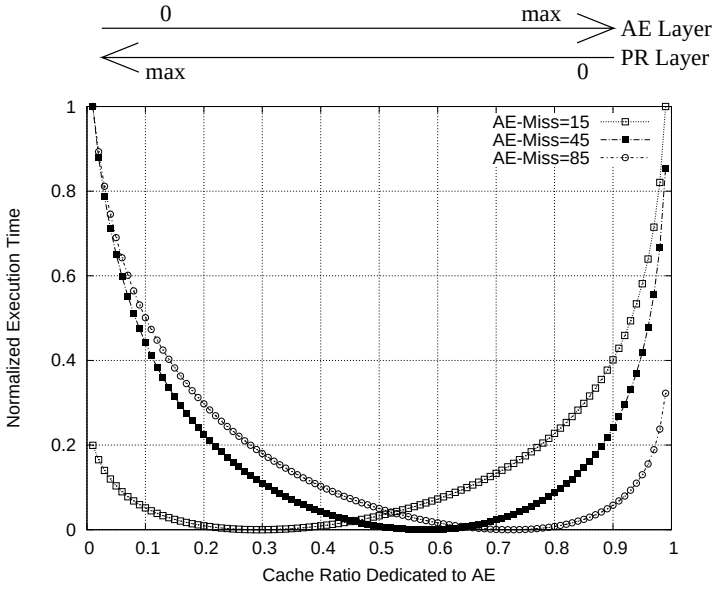
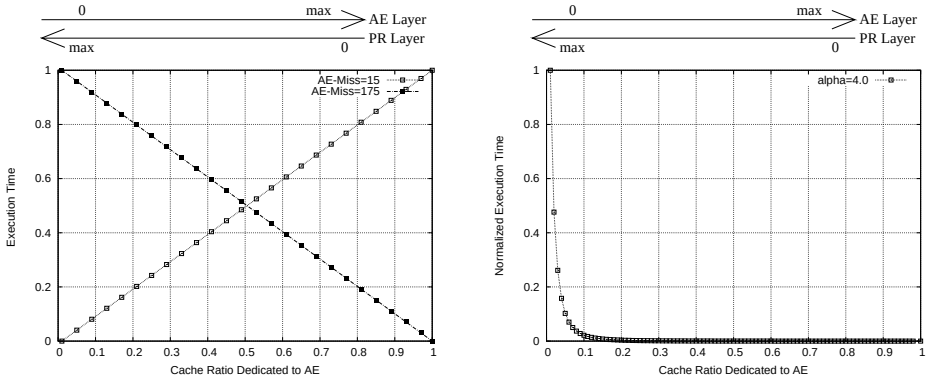


Figure 4.2: *Normalized execution time for typical QA distributions. Parameters:* $Zipf_{\alpha=1.0}$, $C_{Hit}^{AE} = 1$, $C_{Miss}^{PR} = 4$, $C_{Hit}^{PR} = 1$, $RD = 1000$, $PD = 100$, $C_{Miss}^{AE} = \{15, 45, 85\}$.

block becomes more costly. This analysis justifies that a multi-layer cache improves the performance of a QA system, and, furthermore, that there exists an optimal partitioning of the memory between the cache layers.

For completeness, we discuss the parametrization of multi-layer caches for other distributions, even though such an input is not expected for real QA systems, or more generally other types of search engines. For distributions with very low skew, the Zipf distributions tends to the uniform distribution. For example, the $\text{Zipf}_{\alpha=0.0}$ is equivalent to the uniform distribution. In a uniform distribution, all the elements have the same probability of being accessed, and hence caches are not very effective. If we set $F_{(x)}$ as the uniform distribution in Equation 4.3, then the $TotalTime$ is a linear function as we observe in Figure 4.3(a). Therefore, the optimal partition of the caches is at the extremes, i.e. the optimal partition is to dedicate all the available memory to the layer that is more computationally expensive. On the other hand, for extremely skewed distributions, such as $\text{Zipf}_{\alpha=4.0}$, only a very reduced set of documents is frequently accessed. Then, the application does not significantly improve its performance once the p-documents for this set fit in the cache. We observe this pattern in Figure 4.3(b), where the throughput draws a long planar zone for caches with more than 10% devoted to AE. If a system detects an extremely skewed distribution, it can reduce T up to allocate all the very frequent p-documents and use the deallocated memory for other tasks without a significant performance penalty.



(a) Zipf$_{\alpha=0.0}$, $C^{AE}_{Miss} = \{15, 45, 85\}$. (b) Zipf$_{\alpha=4.0}$, $C^{AE}_{Miss} = 15$.

Figure 4.3: *Normalized execution time for general zipf distributions, which are uncommon for QA. Parameters:* $C^{AE}_{Hit} = 1$, $C^{PR}_{Miss} = 4$, $C^{PR}_{Hit} = 1$, $RD = 1000$, $PD = 100$, $C^{AE}_{Miss} = \{15, 45, 85\}$.

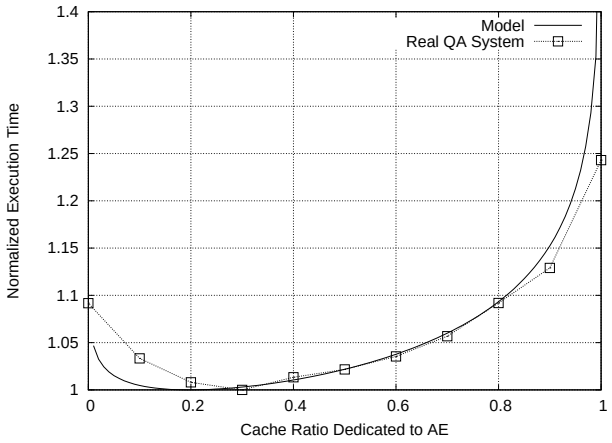
All in all, the analysis of the model in equation 4.1 shows that multi-layer caches are very effective for the typical search engine distributions. In a QA system, the system throughput is maximized when a portion of the cache is allocated for PR and another for AE.

4.3 Experiments with Multi-layer Local Cache

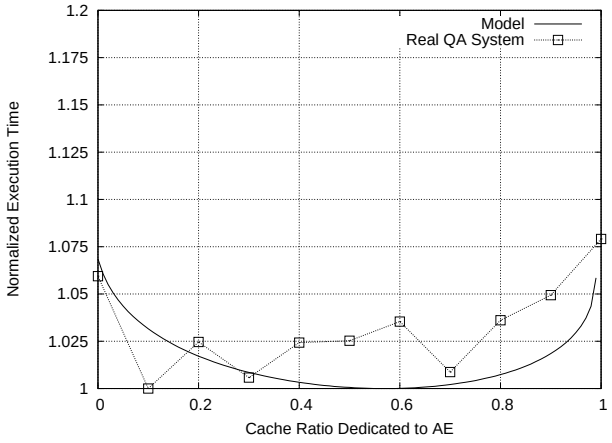
In this section, we evaluate the introduced model against some real configurations of our QA system. This section is a complement to the statistical analysis in the following chapter, which is an analysis of the multilayer cache behavior from an empirical point of view.

Setup: One single PC equipped with an Intel dual core CPU at 2.4Ghz and 2GB of RAM. We use the QA system described in Chapter 3. The question set contains 1200 queries randomly selected following Zipf$_{\alpha=1.0}$ and Zipf$_{\alpha=0.59}$ distributions. We use as textual repository the TREC document collection, which has approximately 4GB of text in approximately 1 million documents. In our experiments, we use direct I/O to read the documents from disk, hence we avoid the interferences produced by the operating system cache. The questions from the query sets are selected from the questions that were part of former TREC-QA evaluations (700 different questions). The response times are averaged over three executions.

Experiments: We assign to the QA system a fixed pool of memory for storing the PR and the AE layers. We execute several runs with different partitions of the available memory between the two layers. We set the size of an entry in the AE layer as 5 times the size of the entry in the PR layer ($\delta = 5$) to



(a)



(b)

Figure 4.4: *Execution time of the QA system in a single computer with different partitions. We superimpose the parametrization of the QA system using the model described in Section 4.2. Query distribution follows a Zipf$_{\alpha=1.0}$ in (a), and a Zipf$_{\alpha=0.59}$ in (b).*

accommodate the additional information generated by the natural language analysis.

In Figure 4.4, we plot the execution times for different partitions of the cache. The vertical axis is the total execution time. The horizontal axis represents the partitioning of the available memory between cache layers. The extremes of the graph correspond to single level caches: the left end is a PR only cache, and the right end is an AE only cache. The left plot corresponds to a query set generated following a Zipf$_{\alpha=1.0}$ distribution. Our experiments show that the best execution time corresponds to a partition-

ing of about 30% of the cache for the AE layer and the rest for the PR layer. These results are consistent with the theoretical analysis presented previously in this chapter. All other partitions yield higher execution times. We observe in the figure that the model introduced in the previous section predicts very accurately the behavior of the system. The right plot corresponds to a similar experiment but with a query set following a $\text{Zipf}_{\alpha=0.59}$ distribution. The results also show that the multi-layer cache is beneficial to the system. The relative benefit in this case is smaller because the cache is less effective for less skewed distributions. We observe that the model is less accurate than for the previous case, but the divergence is not large and we consider that the differences arise because of the simplifications of the model and the experimental noise. Our model predicts that partitioning the cache gives advantage over a single layer cache, and the predicted values are consistent with the observations.

4.4 Summary and conclusions

In this chapter, we have analyzed the caching architecture for a QA system. We have proposed a multi-layer cache configuration that stores partial document computations from the most expensive computing blocks in Question Answering: passage retrieval and answer extraction. We use this multi-layer configuration in the rest of the experiments in this document. We have also developed a model to study the benefit of multi-layer caches for any distribution and application. According to this model, the proper configuration of a cache for question answering assigns a part of the available memory to passage retrieval and another part to answer extraction. We also report a set of experiments that confirm the predictions of our theoretical model. Our conclusions about multilayer caches will be completed in the next chapter with the statistical analysis of our system.

Chapter 5

Multilayer cache statistical analysis

This chapter is devoted to the statistical analysis of the local memory manager. Statistical analysis provides an accurate description of the outcome of a system based on a planned set of experiments that are representative of the system behaviour. The model-building process is an iterative process whose objective is to find a final model good enough to describe the observed data as well as simple enough to be interpreted and to make predictions. This chapter is structured according to this process, preceded by a brief description of the main concepts related to the analysis of variance (ANOVA) method: we perform an initial evaluation of the observations; we generate different models and evaluate their adequacy; we validate our final model; and we draw some conclusions from the final model, useful for the configuration of multi-layer caches. Readers familiar with the statistical vocabulary and the ANOVA method may prefer to skip the first section and go directly to the system analysis.

5.1 Preliminary concepts

The ANOVA is a statistical technique for defining models that quantify the changes of a system's outcome, known as the *response variable*, to a set of *factors*. A factor is a categorical variable, with a small number of levels, under investigation in an experiment as a possible source of variation [43]. The ANOVA models are drawn from *factorial experiments*, in which the scientist defines a set of relevant factors that may affect the system, and tests the outcome for all combinations of the different levels of the factors.

In this chapter, we only consider *fixed effects factors*, which mean that the levels of the factor have been previously fixed by the experimenter. It is assumed that their influence in the response time is constant for each of the tested levels. The levels chosen correspond to those that the experimenter

Levels of A	Observations				Totals	Averages
1	y_{11}	y_{12}	$\cdots$	y_{1n}	$y_{1.}$	$\overline{y}_{1.}$
2	y_{21}	y_{22}	$\cdots$	y_{2n}	$y_{2.}$	$\overline{y}_{2.}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
a	y_{a1}	y_{a2}	$\cdots$	y_{an}	$y_{a.}$	$\overline{y}_{a.}$
					$y_{..}$	$\overline{y}_{..}$

Table 5.1: *Observations collected for a one-way ANOVA.*

thinks are the most interesting based on previous experiments and domain expertise of the experimenter. A second type of factor is the *random effects factor*, which corresponds to variables with a number of levels too large to obtain a sample of each one, and hence the levels are selected following a random uniform distribution. In the design of the experiment, if it is decided that a factor is a fixed effects type, the inference is done for all the levels chosen. Instead, if the factor is a random effects factor, then the conclusions are drawn for all the population of levels. An example of a fixed factor is the number of processors in a system, whereas a random factor is, for instance, the average CPU load of a processor in a shared system.

First, we review the simplest ANOVA model that describes the system outcome depending only on a single factor. Then, in Section 5.3, we generalize the concept to include as many variables as necessary to capture the system behavior.

5.2 One-way ANOVA

The one-way ANOVA describes the influence of a single factor on a given response variable. In Table 5.1, we summarize the experimental dataset necessary to study a system determined by single factor (A), with a different levels. For each level, we obtain n independent observations that we denote with letter y with two subindexes to indicate the corresponding level of A and the observation number respectively: y_{ij}. This design is a *balanced* experiment because the same number of observations is obtained for all the levels of the factors. In this case, the total number of observations is $N = n \cdot a$.

In the first subsection, we explain the model and how to interpret it. Then, we detail the ANOVA hypotheses required to accept the model and how to check them. In the last two subsections we explain the mathematical basis of the ANOVA procedures: how to perform the hypothesis test that determines the significance of each factor, and how to estimate the model parameters.

5.2.1 The one-way ANOVA model

The one-way ANOVA model is as follows:

$$y_{ij} = \mu + \tau_i + \epsilon_{ij} \begin{cases} i = 1, 2, \ldots, a \\ j = 1, 2, \ldots, n \end{cases} \tag{5.1}$$

where μ is known as the overall mean and denotes the expected outcome if no information about the level of the factor under test was available; τ_i is the effect that the i-th level of the factor A has on the response variable (also called *main effect*); and finally, ϵ_{ij} is known as the error term, and captures the variability in the system not explained by factor A. Thus, the model parameters are μ, τ_1,...,τ_a, which are assumed to be constants. However, ϵ_{ij} is assumed to be a random variable for all i and j with a normal distribution with zero mean and constant variance equal to σ^2.

Once the parameters of the model are estimated, it is possible to predict the behavior of the system for any level of A. The expected system outcome for a configuration is modeled as the sum of all the terms in the model for that precise configuration, excluding the error term. For instance, $\hat{\mu} + \hat{\tau_2}$ is the expected system outcome ($\hat{y}_2$) for a system configured with the second level of factor A[1]. There are several methods to estimate the terms of the model, which are usually computed with the aid of statistical software packages. In this thesis, we apply the Minimum Least Squares (MLS), which is equivalent to the Maximum Likelihood Estimation (MLE) for the ANOVA family of models, and also the Weighted Least Squares method (WLS). We describe these methods in Section 5.2.4. Anyway, note that the parametrization of a model is not valid unless it fulfills the ANOVA assumptions, which dictate that the ϵ_{ij} must be: (i) independent random variables, (ii) normally distributed with zero mean, (iii) constant variance (σ^2). We detail this verification process in Section 5.2.4.

The *residuals* correspond to the experimental error, which is interpreted as the influence of all the variables that are not considered in the model because their influence is assumed to be small. The *raw residual* (error term) is the difference between an observation and the model prediction for that configuration:

$$\hat{\epsilon}_{ij} = y_{ij} - \hat{y}_{ij} = y_{ij} - (\hat{\mu} + \hat{\tau}_i).$$

The residual is a very important term for judging the goodness of fit of a model because the smaller the errors the more accurate are the model predictions. However, the raw residual is not a convenient value to measure the error of the model because it does not scale with the outcome. For example, a raw residual equals to 1 s. is very large if the outcome of the

[1] We use the standard notation with a hat to denote the estimator of a given parameter. E.g: $\hat{a}$ is the estimator of the parameter a.

system is 2 s. but it is tiny if the outcome is 1,000 s. Therefore, the error must be normalized to interpret its magnitude. A common normalization is the *standardized residual* that is defined as the raw residuals divided by the square root of the estimated variance:

$$\hat{\epsilon}_{ij}^{std} = \frac{\hat{\epsilon}_{ij}}{\sqrt{\hat{\sigma}^2}}. \tag{5.2}$$

Depending on the method applied to estimate the model parameters, the variance (σ^2) is estimated with a different formula. The estimators of the variance for MLS and WLS are detailed in Section 5.2.4. The standardized residuals of a valid ANOVA model must fit a standardized normal distribution, $N(0,1)$. Otherwise, the model might be biased and the predictions might not be accurate for certain configurations.

An important coefficient in the ANOVA designs is the *coefficient of determination*, $\mathcal{R}^2$, which accounts for the portion of the variability in the data set explained by the model. For the particular case of the one-way ANOVA design, the coefficient of determination is calculated as:

$$\mathcal{R}^2 = \frac{n \cdot \sum_{i=1}^{a} (\overline{y}_{i.} - \overline{y}_{..})^2}{\sum_{i=1}^{a} \sum_{j=1}^{n} (y_{ij} - \overline{y}_{i.})}. \tag{5.3}$$

The numerator of equation (5.3) corresponds to the variability due to factor A, and the denominator corresponds to the total variability in the data. The closer $\mathcal{R}^2$ is to 1.0 the more variability is described by the model, and hence the better it is. A model is commonly considered good enough if $\mathcal{R}^2 > 0.70$, which means that it is able to explain more than 70% of the variability in the observation set.

A second important coefficient to take into account is the *coefficient of variation* (CV), which is a measure of spread for a set of data. It is defined as [43]:

$$CV = 100 \cdot \frac{\sqrt{\hat{\sigma}^2}}{\overline{y}_{..}}. \tag{5.4}$$

This coefficient does not depend on the unit of measure used to obtain the observations, and quantifies the spread related to the central value. Since the CV is a measure of dispersion, it should be small, and the CV of a good model is recommended to be below 5%.

The ANOVA procedure is able to detect which factors do not prove relevant to the system outcome, and hence must be dropped from the model. The factors that affect the system outcome are called $significant$. In other words, a factor is significant if there exists at least one of its levels for which the estimation of its corresponding parameter is statistically different from zero, that is, $\exists i$ such that $\tau_i \neq 0$. The significance is verified with what is known as an F-test, which we describe in Section 5.2.3.

The definition of significance corresponds to an existence condition. Thus, if a factor has proved to be significant, it is interesting to determine which of its levels are statistically different. For instance, it may be interesting to detect in model (5.1) if the levels 1 and 2 of factor A are equivalent. There exists several ways to compare two levels of a factor. Any of them yields the same conclusions if the differences are large. However, for small differences, the tests may bring different conclusions. In this thesis, we apply Tukey's test, which establishes a set of pairwise comparisons between a group of means, and detects the pairs of means which differ. Tukey's test is based on the studentized residue, and is similar to the comparison of means provided by the Student's t-test (for a detailed description of Tukey's test see [77]).

5.2.2 Model Hypotheses

The ANOVA analysis relies on some hypotheses that must be verified. Only if the following hypothesis are true, can the model be accepted as a good model:

- *Independence:* The residuals must be independent, and must not be correlated to the observations. Otherwise, the model is biased for certain combinations of factors. This hypothesis is verified by plotting the standardized residues versus the estimated results. If any structured pattern is observed it means that the residuals are not independent, and thus the model is not valid. Besides, the execution order of the experiments must be performed in a random order to ensure that observations constitute a real sample of the response variable.

- *Normality:* The differences between the model predictions and the observations must be normally distributed. This is checked visually with a histogram of the *standardized residuals*, which must fit a normal standardized distribution. Standardized residuals that do not belong to the interval [-1.96,1.96] (which corresponds to 95% confidence interval for the standardized normal distribution) must be looked at carefully, since they correspond to what is known as an *outlier* observation. In general, moderate departures from normality are of little concern if the factors have fixed effects. The zero mean assumption can be checked at the same time as the dispersion verification by observing if the residuals are centered in zero. Nevertheless, the zero mean assumption is always verified if the parameter estimation has been done correctly.

- *Equality of variance (homocedasticity):* The residuals for each configuration must have equal variances. This is tested graphically with two plots: (a) the residuals with respect to each one of the factors in the model, which shows that the variance is constant for all the levels of

the factors under consideration; and (b) the residuals as a function of the predicted values, which checks if the variance is consistent and independent of the value predicted. The equality of variance implies that all the configurations are predicted with the same accuracy. Nevertheless, it is a very restrictive requirement, and in most empirical experiments, it is not possible to completely fulfill it. In general, the ANOVA analysis is robust to small differences in the variance for different configurations, specially for models with only fixed factors and a large set of observations [76].

5.2.3 Testing the significance of a factor

The term significance is verified with *hypothesis testing*, which is a procedure for assessing whether or not sample data is consistent with statements made about the population [43]. A test of hypothesis compares one statement called the *null hypothesis* (H_0) against another hypothesis, called *alternative* (H_1), which is the statement we wish to accept. The objective of an hypothesis test is to reject the null hypothesis because it is not plausible according to the current set of observations and accept H_1. To perform the hypothesis test, which is to accept or reject H_0, it is necessary to know the distribution that a particular statistic follows under H_0.

Once the statistic distribution is calculated, it is necessary to fix what is known as the *significance level*(α) of the test, which is a value between zero and one (typically 0.05 or 0.01, which accounts a confidence of 95% and 99% respectively) that verifies:

$$P(reject\ H_0 | H_0\ is\ true) \leq \alpha. \tag{5.5}$$

From the set of data obtained from running the experiment, we obtain an observation of the statistic. If the observation falls within the central part of the distribution, i.e. the central part containing the (1-α)% of the distribution, it makes sense to assume that H_0 is true. Otherwise, H_0 has to be rejected since we do not have arguments supporting this hypothesis. Note that a test of hypothesis only proves that H_1 is true with high probability; but it does not provide any conclusions about the validity of the null hypothesis unless it is rejected.

Among all the hypothesis tests of level α, the *power* gives a method of discriminating which is preferable. The *power* of a test measures the probability of rejecting the null hypothesis when it is false [43]. Therefore, we should take always the hypothesis test that has the highest power.

F-test. The significance of a factor in the ANOVA is equivalent to the following hypothesis test:

$$\begin{aligned} H_0 &: \forall i, \tau_i = 0 \quad (Null\ hypothesis) \\ H_1 &: \exists i, \tau_i \neq 0 \end{aligned} \tag{5.6}$$

In order to accept or reject the null hypothesis of equation (5.6), the variability due to the factor A is compared to the residual variability. If the variability caused by factor A is statistically different from the residual variability, then we can conclude that factor A is relevant for the model. Otherwise, we conclude that the influence of A in the response variable is constant for all the levels considered, and we have a strong argument to drop A from the model. The procedure to detect if the variability derived from factor A and the residuals differ is known as F-test, because the distribution of the statistic under the null hypothesis is the Fisher Snedcor distribution (for further information about probability distributions see [77]).

The total variability in the system outcome is defined as the empirical variance of the whole set of observations (SS_T). This sum of squares can be decomposed algebraically into the addition of two different sum of squares: one describing the influence of factor A (SS_A), and another describing the variability not explained by the model (SS_E):

$$SS_T = \sum_{i=1}^{a}\sum_{j=1}^{n}(y_{ij}-\bar{y}_{..})^2 = n\cdot\sum_{i=1}^{a}(\bar{y}_{i.}-\bar{y}_{..})^2+\sum_{i=1}^{a}\sum_{j=1}^{n}(y_{ij}-\bar{y}_{i.})^2 = SS_A+SS_E. \tag{5.7}$$

The previous equation divides the total outcome variability into two independent components, which can be compared by dividing each sums of squares by their degrees of freedom (a-1 for SS_A and N-a for SS_E). The quotients obtained are known as *mean sum of squares* (MSS). The mean sum of squares for factor A (MSS_A) and the one for the error term are equal to:

$$MSS_A = \frac{SS_A}{a-1},$$
$$MSS_E = \frac{SS_E}{N-a}.$$

Due to the hypothesis of the ANOVA model, SS_E is a sum of squares of standardized independent normal random variables, and thus MSS_E follows a χ^2 distribution with N-a degrees of freedom: $MSS_E \sim \chi^2_{N-a}$. Furthermore, if H_0 is true, then SS_A is also a sum of normal standardized independent distributions, and consequently, $MSS_A \sim \chi^2_{N-a}$. Since MSS_A and MSS_E are independent variables, we can apply the F-Test to compare MSS_E and MSS_A. If H_0 holds, then the ratio of MSS_A and MSS_E follows a Fisher-Snedecor distribution, that is:

$$if\ H_0\ is\ true\ then\ F_0 = \frac{MSS_A}{MSS_E} \sim F_{a-1,N-a}$$

We reject H_0, and conclude that factor A is significant if:

$$\frac{MSS_A}{MSS_E} > F_{\alpha,a-1,N-a} \tag{5.8}$$

where $F_{\alpha,a-1,N-a}$ corresponds to the value for which the function $F_{a-1,N-a}$ leaves (1-α)% of the observations. Once we find that $F_0 \not\sim F_{a-1,N-a}$, we reject H_0 and we can conclude that the effect of factor A on the response variable depends on the level of A.

5.2.4 Parameter estimation

As already mentioned, Model (5.1) contains $a + 1$ parameters to estimate: μ, τ_1, τ_2, ..., τ_n. The common term for all configurations, μ, is the expected outcome if no information from the system configuration is provided. Hence, its estimator corresponds to the overall average of the observations $\mu = \overline{y}_{..}$[2].

One of the usual techniques to estimate the rest of the parameters is the method of *minimum least squares* (MLS), which minimizes the sum of the square errors of the error terms. Therefore, the function to minimize is the following:

$$L_{(\mu,\tau_1,\tau 2,...,\tau n)} = \sum_{i=1}^{a}\sum_{j=1}^{n} \epsilon_{ij}^2 = \sum_{i=1}^{a}\sum_{j=1}^{n} (y_{ij} - \mu - \tau_i)^2 . \tag{5.9}$$

The solutions to this optimization problem correspond to the solutions of a system of linear equations compound of the partial derivates of function (5.9) ($\partial L/\partial\mu = 0$, $\forall i : \partial L/\partial\tau_i = 0$). The resulting system of linear equations is underdetermined, and an equation that is consistent with the system interpretation must be added to find a unique solution for the system. Usually it is assumed the constraint $\sum_{i=1}^{a} \tau_i = 0$, which states that the sum of the deviations around the mean due to A is 0. Note that depending on the equation that we introduce to the undetermined system, we get different τ_i estimations. However, the differences among the levels of the studied factor, which is the fundamental value in the ANOVA analysis, remains constant independently of the constraint that has been assumed ($\forall i, j : \tau_i - \tau_j = ct.$). For example, the SPSS software introduces the constraint $\tau_a = 0$. The solutions to this system of equations take the form $\hat{\mu} = \overline{y}_{..}$ and $\hat{\tau}_i = \overline{y}_{i.} - \overline{y}_{..}$, according to the notation shown in Table 5.1. Further details of this derivation are available in [76].

Another common method to estimate the parameters of a model is the *Maximum Likelihood Estimation* (MLE), whose objective is to maximize the

[2]In fact μ can be set to any constant, and some authors prefer to set its value to 0. We prefer to consider μ as the average of observations because the remaining terms of the model are easier to interpret. Independently of the final μ chosen, the remaining terms are estimated with the methods described in this section.

likelihood function. However, for any lineal model, and in particular for the ANOVA models, the parameter estimations obtained by applying the MLS method are exactly the same as the ones obtained with MLE.

In this thesis, in addition to MLS, we also apply *Weighted Least Squares* (WLS), which is an adaptation of the MLS that is adequate if one of the factors exhibits a larger variance than the rest. The WLS sets a weight for each configuration in such a way that the configurations with a lower variance contribute more to the parameter estimation than the ones with a larger variance. This makes sense because the configurations with a lower variance are more precise. We must point out that WLS should only be selected after checking that it really improves the model goodness of fit compared with MLS.

The function to be minimized if the WLS is applied:

$$L_{(\mu,\tau_1,\tau 2,\ldots,\tau n)} = \sum_{i=1}^{a}\sum_{j=1}^{n} \epsilon_{ij}^2 \cdot w_i = \sum_{i=1}^{a}\sum_{j=1}^{n} (y_{ij} - \mu - \tau_i)^2 \cdot w_i. \tag{5.10}$$

The weight term, w_i, is set as the inverse of the variance for the i-th level of A ($w_i = \sigma_i^{-2}$), which indeed reduces the importance of the observations corresponding to levels with a large variance. The procedure to find the minimum is analogous to the one described for the MLS.

Variance estimation. The estimation of the variance depends on the optimization method applied to generate the model. For the case of MLS:

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{a}\sum_{j=1}^{n} (y_{ij} - \hat{y}_{ij})^2}{N - a}.$$

In the case of WLS, the variance estimator must take into account the weight terms:

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{a}\sum_{j=1}^{n} (y_{ij} - \hat{y}_{ij})^2 \cdot w_i}{N - a}.$$

5.3 Factorial design: n-way ANOVA

Scientific experiments often involve several factors, because the experiment outcome is influenced by multiple variables. Fortunately, the one-way ANOVA model can be generalized to the n-way ANOVA in order to capture as many factors as required, though in real applications it is uncommon to include more than 7 or 8 factors. The n-way ANOVA model helps to understand how the different factors are related, and quantifies their influence. It also allows us to determine the configuration that leads to an optimal response.

An example of an ANOVA model with three factors (three-way ANOVA)

is the following:

$$y_{ijkl} = \mu+\alpha_i+\beta_j+\gamma_k+\underline{(\alpha\beta)_{ij}}+\underline{(\alpha\gamma)_{ik}}+\underline{(\beta\gamma)_{jk}}+\underline{(\alpha\beta\gamma)_{ijk}}+\epsilon_{ijkl} \begin{cases} i = 1, 2, \ldots, a \\ j = 1, 2, \ldots, b \\ k = 1, 2, \ldots, c \\ l = 1, 2, \ldots, n \end{cases} \tag{5.11}$$

The ANOVA models with two or more factors contain a new type of term, called *interaction*, which appears underlined in the previous equation. The interaction describes the relationship between two (or more) factors. The *order* of an interaction is the number of factors used to define it minus one. An interaction between factor A and B indicates that the effect of factor A on the outcome depends on the level of B, and that of B on the level of A [85]. Let us suppose that the contributions of two given levels of the factors A and B are positive. Then, a positive interaction corresponds to a synergy between these two levels, which means that the increase of the individual contribution of each factor is not enough to explain the increase in the system outcome. A negative interaction of the two levels indicates that the factors are antagonistic for this combination of levels, and even though they individually increase the system outcome, the contribution of its combination is smaller than the sum of the individual contributions. Although the mathematical basis allows interactions of an arbitrary number of factors, it is uncommon to consider the interactions of more than two factors because of they are difficult to interpret. We notate interactions as the set of factors interacting inside a pair of brackets, and a sequence of subindexes indicating the corresponding levels of the interaction.

In general, the complete ANOVA model for k factors is the addition of: (a) the overall mean of the whole set of observations, (b) one term for each factor under study (k terms), (c) the set of interactions of any order, which are all the possible combination of terms from two to k elements ($\sum_{i=2}^{k} \binom{k}{i}$ terms), and (d) the error term. The hypotheses of the n-way ANOVA model are similar to those previously described for the one-way ANOVA, and thus the mathematical description of the model is similar.

The function to minimize in factorial analysis is a generalization of that described in equation (5.10). For model (5.11), the function to minimize in order to estimate the model parameters using the WLS method is the following:

$$L = \sum_{ijk} (y_{ijk} - \mathcal{F}_{(ijk)})^2 \cdot w_i \tag{5.12}$$

where $\mathcal{F}_{(ijk)}$ denotes the sum of the different terms defining the model with the exception of the error term, that is:

$$\mathcal{F}_{(ijk)} = \mu + \alpha_i + \beta_j + \gamma_k + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\beta\gamma)_{jk} + (\alpha\beta\gamma)_{ijk}.$$

If we estimate the model parameters with the MLS method, then it is enough to take $w_i = 1$ for all i in 5.12.

The system of linear equations that we get from the partial derivates of L has several degrees of freedom, and we impose a set of additional restrictions in order to get a single solution. The restrictions are the natural generalization of the ones introduced in the one-way ANOVA. For the three factor anova the equations are the following:

$$\sum_{i=1}^{a} \alpha_i = \sum_{j=1}^{b} \beta_j = \sum_{k=1}^{c} \gamma_k = 0,$$

$$\forall i : \sum_{j=1}^{b} (\alpha\beta)_{ij} = \sum_{k=1}^{c} (\alpha\gamma)_{ik} = 0, \quad \forall j \forall k : \sum_{i=1}^{c} (\alpha\beta\gamma)_{ijk} = 0,$$

$$\forall j : \sum_{i=1}^{b} (\alpha\beta)_{ij} = \sum_{k=1}^{c} (\beta\gamma)_{jk} = 0, \quad \forall i \forall k : \sum_{j=1}^{b} (\alpha\beta\gamma)_{ijk} = 0,$$

$$\forall k : \sum_{i=1}^{a} (\alpha\gamma)_{ik} = \sum_{j=1}^{b} (\beta\gamma)_{jk} = 0, \quad \forall i \forall j : \sum_{k=1}^{a} (\alpha\beta\gamma)_{ijk} = 0.$$

Assuming these additional equations, the resulting system has a unique solution that can be computed with any standard procedure for solving systems of linear equations, such as gaussian elimination.

The analysis of the model determines which factors and interactions are *significant* and which are not. Analogously to the main effects factor significance, it is said that an interaction is significant if $\exists ij : (\alpha\beta)_{ij}$ is statistically different from zero. For the factorial ANOVA, the statistical test to detect if a term is significant is a generalization of the one-way ANOVA test, introduced in Section 5.2.3. In the general factorial ANOVA design, the terms that do not prove significant must also be removed from the model.

Although all significant interactions increase the accuracy of the model, a researcher may remove the less significant interactions if $\mathcal{R}^2$ is large enough and the hypothesis of the model still hold. The reason is the *parsimony principle*, which states that among competing valid models, all of which provide an adequate fit for a set of data, the one with the fewest parameters is to be preferred [43]. As a consequence of the parsimony principle, models with fewer interactions are preferable because they are easier to interpret. For example, a five factor model with no interactions and $\mathcal{R}^2 = 0.85$ is preferable to the complete model with all the possible interactions (32 terms!) and $\mathcal{R}^2$ equal to one.

Example: A computer scientist wants to study the performance of the sorting procedure implemented in a distributed database. The first factor, denoted by A, accounts for the four different algorithms under test with levels i={1, 2, 3, 4}. Additionally, he adds a second factor to test the scalability, B, which accounts the number of nodes available. He has two computers in the experiment, which correspond to the two levels of B: j={1,2}. There are $4 \cdot 2$ different configurations, and for each one the experimenter performs 3

Execution Time (s)	1 Node			2 Nodes		
Algorithm 1	2.14	2.16	2.17	2.11	2.10	2.15
Algorithm 2	4.22	4.28	4.21	2.12	2.18	2.15
Algorithm 3	8.72	8.63	8.64	4.98	4.79	4.89
Algorithm 4	17.08	17.05	17.21	8.50	8.49	8.56

Table 5.2: *Example: Execution time of a sorting algorithm.*

Source	**Sum of Squares**	**F**	**Significance**
Model	517.91	24060	0.000
Intersection	1008.28	327898	0.000
Computers (A)	58.46	19014	0.000
Algorithm (B)	368.65	41805	0.000
Computers * Elements (A*B)	73.79	7999	0.000
$\mathcal{R}^2$	0.99		

Table 5.3: *Partial output from the SPSS ANOVA procedure for the dataset reported in Table 5.2*

observations, which adds up $N = 4 \cdot 2 \cdot 3 = 24$ observations. In Table 5.2, we show the summary of his experiments[3]. The model analysis, detailed in Table 5.3, shows that all terms are significant because their significance level is below 0.05. A visual inspection of the observation yields that the first algorithm is the best, which is confirmed because the estimator of τ_1 is considerably smaller than the estimators of τ_2, τ_3, τ_4. The model also shows that the optimal algorithm (A) is the first one for either one or two nodes (B). However, it detects a scalability problem, because there exists a significant interaction between the algorithm and the number of nodes (A*B). If one analyzes the model predictions, one observes that the first algorithm does not scale properly because for two nodes, the interaction $(\alpha\beta)_{11} - (\alpha\beta)_{21}$ is as large as the effect of the addition of the second node $\alpha_1 - \alpha_2$. Moreover, if via a Tukey test one compares the algorithms (with the number of computers fixed) one obtains that the first and the second algorithms are not statistically distinguishable. After the quantitative analysis, he may also have additional information from the algorithms that says the algorithm 2 is preferable because its memory footprint is smaller than algorithm 1. Thus, he may decide that the distributed database engine does not need to include the first algorithm, because for more than one computer the second is as good as the first and its memory requirements are smaller.

[3]The results of the experiment are fictitious and are only used to exemplify the statistical method.

Parameter	Term Value
μ	8.51
α_1	8.597
α_2	0
β_1	-6.397
β_2	-4.357
β_3	-3.630
β_4	0
$(\alpha\beta)_{11}$	-8.560
$(\alpha\beta)_{12}$	-8.520
$(\alpha\beta)_{13}$	-4.820
$(\alpha\beta)_{14}$	0
$(\alpha\beta)_{21}$	0
$(\alpha\beta)_{22}$	0
$(\alpha\beta)_{23}$	0
$(\alpha\beta)_{24}$	0

Table 5.4: *Parametrization of the terms in the example model*

5.4 Experimental design

In this section, we analyze the multilayer caches described in Chapter 4. Our objective is to find out how the system outcome changes to a set of relevant variables. Based on our previous empirical experiment, we decide to pick four factors that we think are relevant. These factors, and their corresponding levels, are as follows:

- *CacheSize*: This parameter accounts for how much memory the system allocates for the multilayer cache. We take as a cache unit the number of documents stored in the cache. We set four different values: $CacheSize = \{$ 5000, 10000, 20000, 30000 $\}$.

- *PercentAE*: This parameter is the fraction of the cache devoted to the answer extraction layer. The remaining fraction is allocated to passage extraction. A value of 1 dedicates all the available memory to answer extraction, whereas 0 assigns all memory to passage extraction. We set eleven different values that divide the interval [0,1] into ten parts of equal size, thus: $PercentAE = \{$ 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 $\}$

- *Distribution*: This determines the distribution of the query input set. Query logs from search engines follow power law distributions and thus we consider three different zipf distributions with varying skewness [9,71,97]. More exactly, we consider three Zipf distributions with a different parametrization: $Distribution = \{Zipf_{\alpha=0.59}, Zipf_{\alpha=1.0}, Zipf_{\alpha=1.4}\}$. We generate three query sets from the questions of TREC-QA, each compound by 800 queries [78].

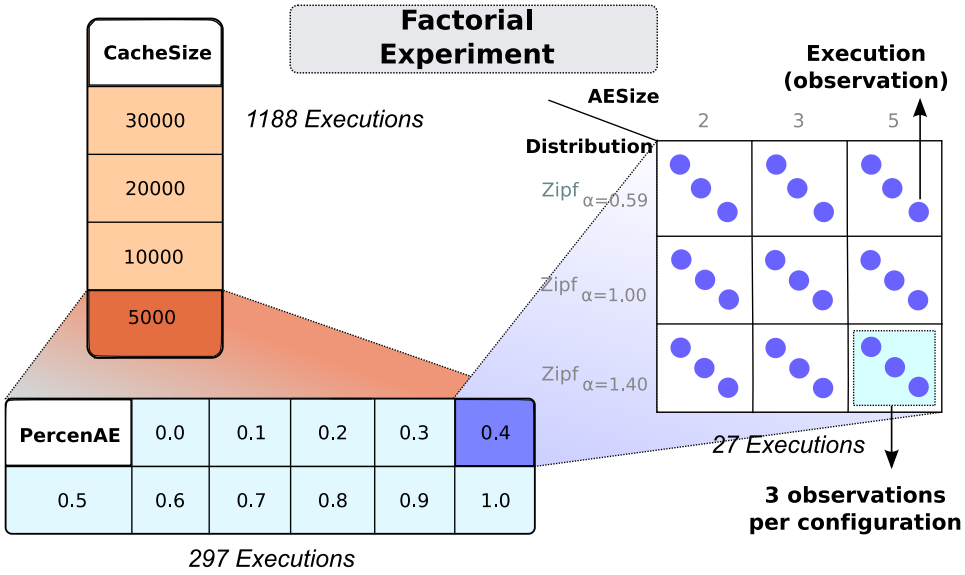


Figure 5.1: *Experimental design*

- *AESize*: A different QA system can rely and store different NLP analysis. We study how the amount of data in a p-document affects the performance of a multi-layer cache. This parameter accounts for the ratio of the size of a r-document and a p-document. For example, a value of 2 indicates that a p-document is twice as big as a r-document. We set three different values: $AESize = \{ 2, 5, 10 \}$.

The experimental design is a four-factor factorial design with fixed effects, since the levels have been fixed initially. So, the number of configurations is equal to 4·11·3·3=396. For each configuration we execute the QA system three times, which gives a total of $N = 396 \cdot 3 = 1188$ observations. The design of our experiments is drawn graphically in Figure 5.1.

Each observation uses two different computers: one hosts the question answering system with the multi-layer cache, and the other issues queries. We use as data repository the TREC document collection [78], which is stored in the local hard disk of the computer running the question answering system. The data collections has approximately 4GB of text in 1 million documents. The executions are performed in a cluster of homogeneous computers following a random order of executions and a random choice of computers. Each computer in the system is equipped with an Intel dual core CPU at 2.4GHz. The total execution time for this experiment corresponds to more than 70 days of computation in a single computer.

Our test environment did not have enough memory to execute the six configurations with the largest memory requirements ($CacheSize = 30000$,

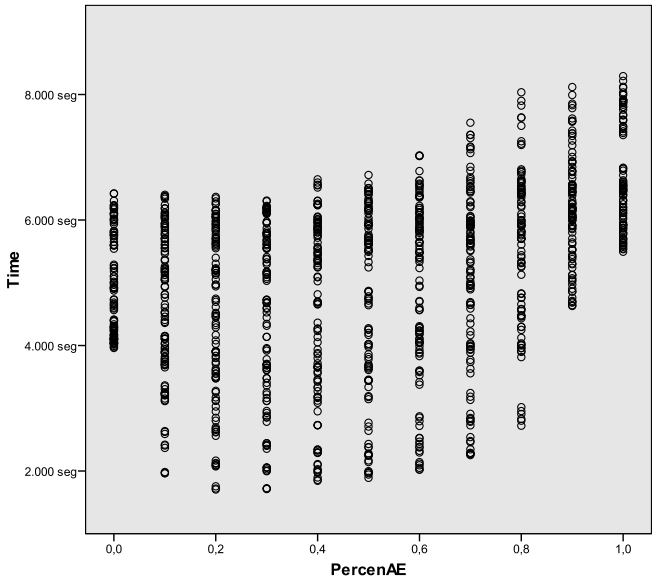


Figure 5.2: *Execution time for multi-layer caches.*

$AESize = 2$, $PercentAE = \{0.9,\ 1.0\}$, $Distribution = \{0.59,\ 1.0,\ 1.4\}$), and thus we removed these configurations from the final model. As a consequence, we finally have 390 configurations and 1170 observations. It is important to observe that we have removed just 1.5% of the observations, and they only belong to specific configurations. As a consequence, our conclusions will exclude these configurations.

5.5 Exploratory data analysis

It is fundamental to detect initially if the data set contains any anomalous observations, since the conclusions may considerably differ with or without them. An *outlier* (or anomalous observation) is defined as an observation that appears to deviate markedly from the other members of the sample in which it occurs [43]. We consider an outlier those observations that do not belong to the interval $(Q_1 - 1.5 \cdot HIR,\ Q_3 + 1.5 \cdot HIR)$ Q_1 and Q_3 being the first and third quartiles and HIR the half interquartile range defined as $(Q_3 - Q_1)/2$.

We start the analysis by plotting the observations as a function of each one of the factors. We find that for the different levels of $PercentAE$ it is difficult to assume the homocedasticity hypothesis because we observe in Figure 5.2 that the variance of the observations for the central levels of the factors is considerably smaller than the variance obtained for the extreme levels.

Thus, we consider it convenient to divide the analysis into two parts: one for the central values of $PercentAE$ (among 0.1 and 0.9, both included), and the other for the extreme values of $PercentAE$ (0.0 and 1.0). Below, we discuss the extreme values, and in section 5.6 the central values.

The values of $PercentAE$ 0.0 and 1.0 correspond to having all the cache devoted to either the passage extraction or the answer extraction block, respectively. In other words, the extremes of $PercentAE$ are the single layer caches. We observe that the variance exhibited by the data is much smaller for these configurations, which means that single layer factors are less affected by the combination of factors. This result is in accordance with our predictions from the theoretical model, in which the multilayer caches are better and highly affected by the cache partitioning and the query distribution. We also observe that, for all combinations of cache size, distribution and p-document size, a single layer cache is not the best configuration. It is preferable to give a portion of the memory available for caching to each of the layers to improve the performance. Due to the fact that the single layer configuration is not optimal, we do not analyze the model further for single layer caches and conclude that it is preferable to enable the two layers of caching for question answering. In summary, the results for single layer caches are in accordance with our experiments, where we found that multilayer caches were more effective than single layer caches for question answering [38]. According to this analysis, in the following sections the factor $PercentAE$ will be considered with nine levels that range from 0.1 to 0.9.

5.6 ANOVA Model for multi layer caches

5.6.1 Model selection

We have analyzed several ANOVA models with the SPSS univariate procedure for the four factors under study: $CacheSize$, $Distribution$, $AESize$, and $PercentAE$. We denote the main effect of each variable with the following assignation of greek letters:

- α_i: is the effect of the i-th level of $AESize$.
- β_j: is the effect of the j-th level of $Distribution$.
- π_k: is the effect of the k-th level of $PercentAE$.
- γ_l: is the effect of the l-th level of $CacheSize$.

We denote the interactions with the combinations of the previous letters. For example, the interaction between the j-th level of $Distribution$ and the l-th level of $CacheSize$ is denoted as $(\beta\gamma)_{jl}$.

The different models considered for the analysis of our data are in Table 5.5. The model with only main effects explains 77.8% of the variability in the data and yields to an estimated standard deviation equal to 674 s (see Table 5.5). This simple model is fairly good and explains a large portion of the variability. The standard deviation is small compared to the average execution time (μ = 5172 s), but still an important portion of the system variability (22%) is not explained. In order to increase the goodness of fit of the model, we include the effect of the first order interactions. We generate six different models, each one with the main effects plus one of the six possible first order interactions. The corresponding $\mathcal{R}^2$ value and standard deviation for each model are also reported in Table 5.5. We notice that three out of the six interactions are not significant and we can automatically drop them from the final model: $AESize * CacheSize$, $AESize * Distribution$ and $AESize * PercentAE$. Among the rest, not all the interactions are equally important. We observe that $Distribution * PercentAE$ and $Distribution * CacheSize$ are more significant than $PercentAE * CacheSize$ because they increase the goodness of fit significantly up to 0.853 and 0.890, respectively. This makes sense because both include the query distribution (β), which is known to be very important for determining the system throughput, and according to the theoretical model multi-layer caches are more effective for certain distributions.

The complete model with all the significant first order interactions, which is the most precise model with only first order interactions, achieves $\mathcal{R}^2$=0.975 and σ = 237s. There is still a gap between the best model with one interaction (which include $Distribution * CacheSize$) and the complete model, and thus we consider the inclusion of the most significant interactions of two factors in the model. We choose $Distribution * PercentAE$ and $Distribution * CacheSize$ because they are the most important, and the model quality improves significantly up to $\mathcal{R}^2 = 0.955$ and σ = 309s. This last model shows good precision and a small standard deviation, and therefore it is chosen as our final model:

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\beta\pi)_{jk} + (\beta\gamma)_{jl} + \epsilon_{ijkl} \tag{5.13}$$

Even though we remove the extreme values of $PercentAE$ from the model, we notice that there is still an important difference in the variances among the different levels of $PercentAE$. For this reason, we do the parameter estimation with WLS squares instead of the usual MLS. We execute the iterative WLS procedure of SPSS weighted by the factor $PercentAE$ (see Table 5.6. In this case, the model shows similar precision and smaller deviation ($\mathcal{R}^2 = 0.953$ and $\sigma = 222s$), and additionally, the distribution of the residues is less peaked. So, our final model is the one that appears in equation (5.13) estimating the parameters using the WLS method, weighted

Model	$\mathcal{R}^2$	σ	Signif.
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l$	0.778	674E3	-
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\alpha\gamma)_{il}$	0.782	676E3	No
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\alpha\beta_i)_{ij}$	0.782	672E3	No
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\alpha\pi)_{ik}$	0.783	678E3	No
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\beta\gamma)_{jl}$	0.890	479E3	Yes
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\pi\gamma)_{kl}$	0.799	655E3	Yes
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\beta\pi)_{jk}$	0.853	558E3	Yes
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\beta\pi)_{jk} + (\beta\gamma)_{jl}$	0.955	309E3	-
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\beta\pi)_{jk} + (\beta\gamma)_{jl}$ (estimated by WLS)	0.953	222E3	
$\mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\beta\pi)_{jk} + (\beta\gamma)_{jl} + (\gamma\pi)_{kl}$	0.975	237E3	-
$\alpha \equiv AESize$ | $\beta \equiv Distribution$ | $\pi \equiv PercentAE$ | $\gamma \equiv CacheSize$			

Table 5.5: *Model fitting (the response variable is in milliseconds).*

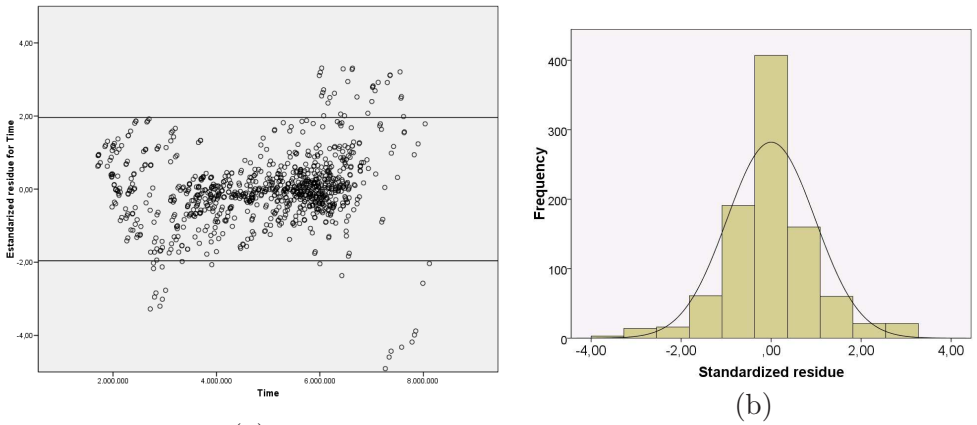
by $PercentAE$, because it explains 95.1% of the variability and fulfills the hypothesis of the ANOVA quite well, as we will see in the next subsection.

5.6.2 Adequacy checking

In this section, we have to check the ANOVA requirements described in Section 5.2. The independence among observations is accomplished by the experiment design, because the configurations are executed in random order.

In Figure 5.3(a), the standardized residuals appear as a function of the observed values. We observe that more than 96% of the residuals fall between the values -1.96 and 1.96. It is important to observe that no patterns are observed in the sense that the residuals do not increase or decrease as a function of the observed values. Although we do not observe severe patterns, we note that the model is more accurate in the central region because for these values the observations show less variability and the model can predict the outcomes very exactly. We also find big residuals for configurations with large execution times, some of them larger than 4. All these observations correspond to the most skewed distribution ($Zipf_{\alpha=1.4}$) and large values of $PercentAE = 0.9$. For these configurations the model is not accurate, and we are unable to draw conclusions. However, these configurations correspond to slow systems, whose configuration can be improved to reduce the execution time.

Figure 5.3(b) depicts the histogram of the residuals. The distribution is centered at 0 and has a standard deviation of 0.98. The residuals are more peaked than a normal distribution, because many observations generate a very small residual. Since our model contains only fixed effects, we validate



(a) (b)

Figure 5.3: *Standardized residuals of the model. (a) As a function of observed execution time. Lines indicate the level where a normal distribution has the 95% confidence interval (b) Histogram of the residuals.*

Factor	SS	D.F.	MSS.	F	Significance	Power
Distribution	2.13E15	2	1.06E14	2136.1	0.000	1.000
CacheSize	3.06E14	3	1.02E14	2048.0	0.000	1.000
AESize	9.12E12	2	4.56E12	91.57	0.000	1.000
PercentAE	1.48E14	8	1.85E13	371.13	0.000	1.000
Dist*PercentAE	8.00E13	16	5.00E12	100.45	0.000	1.000
Dist*CacheSize	1.18E14	6	1.97E13	397.16	0.000	1.000
Error	4.56E13	916	4.97E10			
$\mathcal{R}^2 = 0.953$		$\sigma = 222E3$		$CV = 4.31\%$		

Table 5.6: *Final ANOVA model generated for the multi-layer cache with WLS parameter estimation (the response variable is in ms)*

the histogram distribution, because the ANOVA is robust to such deviations and this does not invalidate our conclusions.

Figure 5.4 shows the predicted values versus the expected values. It can be appreciated that there is a very accurate matching between the two variables as a consequence that R^2 is very close to one. Furthermore, the residuals lay on both sides of the identity function and strange shapes are not observed. In conclusion, the hypotheses of the ANOVA model are fulfilled, and thus we accept Model (5.13) as a good model for our data.

5.6.3 Model discussion

Table 5.6 shows the summary of the ANOVA model generated from our analysis. The name of the factor or the interaction under consideration

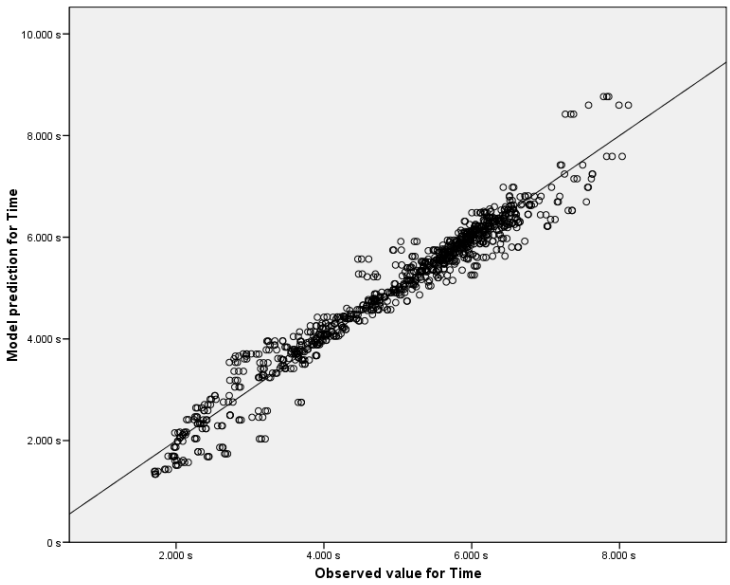


Figure 5.4: *Observed execution versus values predicted by the model.*

appears in the first column. The second and the third columns correspond to the sums of squares and degrees of freedom for that factor, respectively. The MSS column is the mean sum of squares, which is the quotient of SS by its degrees of freedom. Column F contains the statistic of the F-test necessary to carry out the hypothesis tests of the ANOVA. Finally, the last two columns tell us if the factor or interaction are significant and what the power of the hypothesis test obtained is.

The first conclusion is that the model is very accurate, because it explains 95.1% of the variability in the data set with a reduced CV. The most relevant variables to the system throughput are $Distribution$ and $CacheSize$, as we expected, because they set the general testing environment where the system is executing. However, the former cannot be controlled by the QA system because it is an external factor. Furthermore, although the latter is configured by the system administrator, it is limited by the total available memory in the system. Although the factor $AESize$ is significant, it has a smaller influence on the response variable than the other two variables. Regarding $PercentAE$, which is the main multilayer cache parameter parameter under study, the model shows that $PercentAE$ is also very important for achieving a good throughput, because $PercentAE$ is a significant factor and its F value is large.

The model includes two significant interactions: $Distribution{*}CacheSize$ and $Distribution{*}PercentAE$. The interaction $Distribution{*}CacheSize$ is the most influential one because the SS is larger and the degrees of freedom smaller. This is a consequence of the logarithmic hit rate growth with respect

to the cache size for skewed distributions [71]. Moreover, the interaction $Distribution * PercentAE$ indicates that the multilayer cache performance is dependent on the distribution: some distributions benefit more from the multilayer cache than others. These results agree with the conclusions from the theoretical model described in the previous chapter.

In Figure 5.5, we depict the predicted and observed values for different cache partitions and cache sizes. We observe the high accuracy of the model because the observations and the model are close. We see that the multilayer cache is effective for most cache sizes, with the only exception of the smallest tested cache size where the multilayer cache is just as good as a single layer cache. However, we note that this configuration has an extremely small cache and very low hit rate. This cache size could only fit on average the documents of the last 8 queries approximately. As noted by previous studies [71], LRU is not the best caching algorithm, but in general it is very close to the optimal. However, in caches with large contention like in our smallest cache size tested, it is preferable to implement more elaborate algorithms, such as LRU-k [81] or ARC [72], because they manage caches better with very limited resources. We believe that if we implement a more advanced cache algorithm the results would resemble the curves obtained for the other cache sizes.

In Figure 5.5, two trends for the multi-layer cache are observed. The first is the concave shape of the execution time, which is minimum for intermediate partitions of the multi-layer cache. This valley becomes deeper for larger caches. The second trend is that for the larger caches the minimum covers a wider region of configurations.

In order to see how the model behaves for the different distributions in Figure 5.6, we plot the best configuration for each query distribution, both for the observations and the model predictions. Here, we see that multilayer is effective for all the skewed distributions tested, and the throughput gains are larger for the most skewed distributions, which is in accordance with the inclusion of $Distribution * PercentAE$ in the model as a significant interaction. We highlight the throughput for the most skewed distributions, in which the multi-layer cache doubles the throughput of the best single layer caches.

Finally, from our analysis we want to deduce some recommendations to configure the multi-layer caches. The system administrator can customize the parameter $PercentAE$ of a system, and hence we study what its optimal value is. Because of the interaction $PercentAE * Distribution$, the optimal range varies for each different distribution under test. We compare the different levels of the $PercentAE$ variable for each distribution with a Tukey test, and the conclusions are that the execution time is optimal in:

- The interval [0.1, 0.6] for distribution $\text{Zipf}_{\alpha=0.59}$.
- The interval [0.1, 0.4] for distribution $\text{Zipf}_{\alpha=1.0}$.

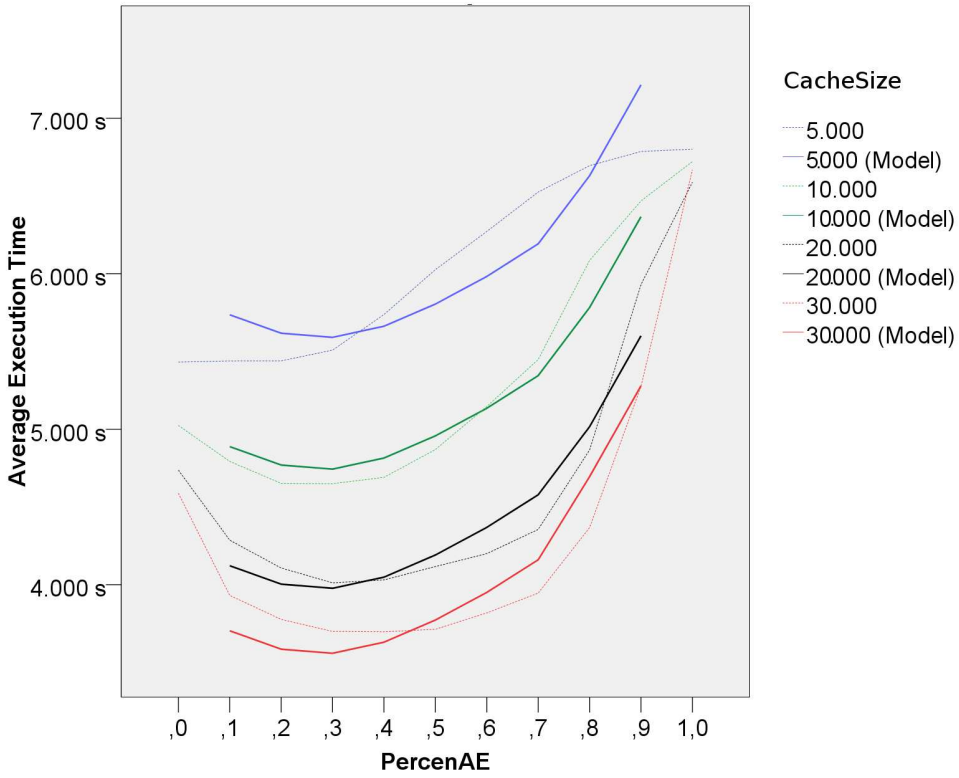


Figure 5.5: *Average of model predictions and observed execution time for different PercentAE and CacheSize.*

- The interval [0.2, 0.4] for distribution $\text{Zipf}_{\alpha=1.4}$.

According to the results, the optimal parameter range that includes all distributions is [0.2-0.4]. This indicates that the configuration of multi-layer caches is not difficult for the system administrator, because we reach optimal performance for a wide range of *PercentAE* settings. We also observe that the intervals are inclusive, and thus if we are able to configure our system for skewed distributions, we will have good results for less skewed distributions.

5.7 Summary and conclusions

In this chapter, we analyze the behavior of multi-layer caches for a broad range of configurations with the aid of an appropriate statistical model. This model is very accurate because it explains more than 95% of the registered variance in the observations. Among the different results, we emphasize the following conclusions:

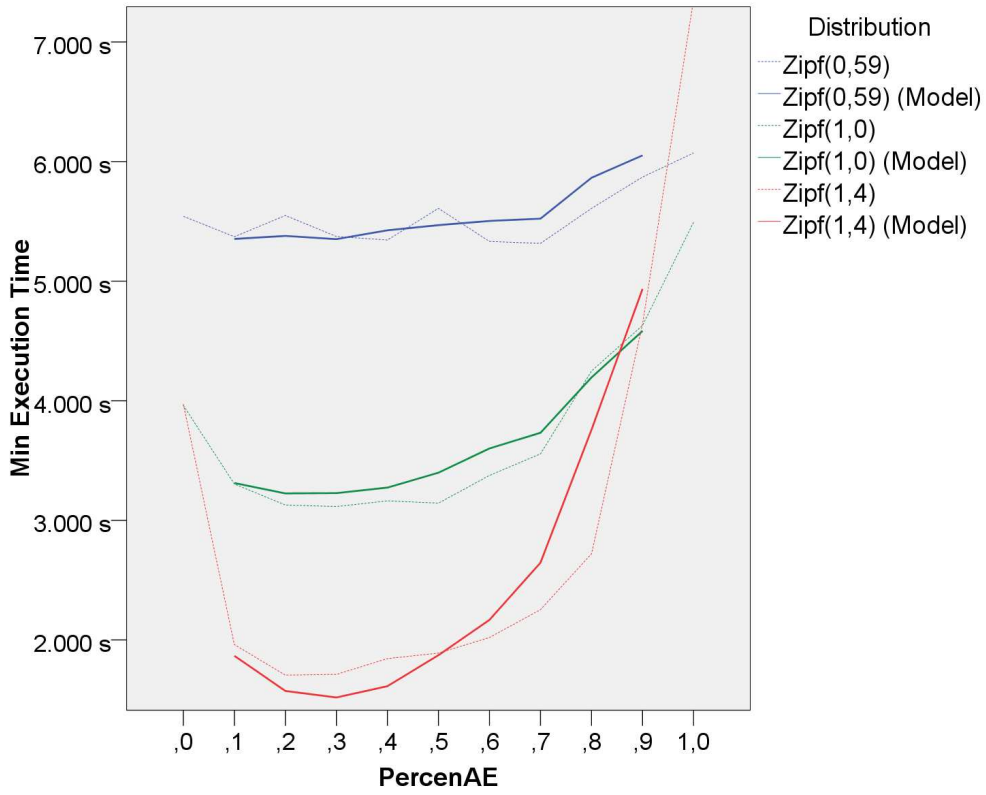


Figure 5.6: *Model predictions and observed execution time for different PercentAE and Distribution.*

- The statistical analysis confirms that multi-layer caches are more effective than single layer caches.
- The model determines that the most important effects for predicting the system performance are the workload distribution (*Distribution*), and the total memory available for caching (*CacheSize*). Nevertheless, the model indicates that how we partition the cache among the individual layers (*PercentAE*) is also fundamental.
- The multi-layer cache is effective for skewed distributions and its benefit increases when the skewness of the query distribution increases, as the interaction $Distribution * PercentAE$ indicates.
- The optimal partition between the passage retrieval and the answer extraction cache is a range of *PercentAE* settings, which depends on the query distribution and grows with the cache size. In our system, the narrowest range is a 20 percentile for the most strict configuration.

For the less strict configurations, the range widens but does not exclude the more strict configurations. According to the model derived, we recommend a partition between 20% and 40% of the available memory to the AE layer for systems similar to ours.

The statistical analysis presented in this chapter agrees with the predictions from the theoretical model introduced in the previous chapter. The analytical analysis presented in Chapter 4 draws a parabola similar to the one depicted by the statistical model predictions in Figure 5.5. These two different analysis reinforce each other, and both can be used to deploy and configure a multi-layer cache in a search engine.

Part III

Cooperative Cache Management for Distributed Question Answering

Chapter 6

Evolutive Summary Counters in a Distributed Question Answering Architecture

In this chapter, we propose the Evolutive Summary Counters (ESC) data structure as a device to improve the cache performance of a distributed system. In order to do so, we start by describing the QA distributed architecture of the system used in the experiments of this thesis in Section 6.1: (i) the implementation of a distributed QA system; (ii) the new operations in the cache manager to include the cooperative cache manager operations; and (iii) the load balancing scheduler. Then, in Section 6.2, we introduce the ESC, which are one of the main contributions of this thesis. The ESC is a general data structure that records the recent documents accessed in a node. This data structure is the cornerstone of our proposals related to distributed computing, which are introduced in the following chapters, because all our proposals take advantage of the summarized statistics generated from the ESC. Given that ESC is a generic data structure that can be shared by many algorithms, we delay the evaluation of its configuration until Chapter 9 when we apply statistical analysis to find the impact of the different parameters of the ESC in the throughput of the system. Finally, in Section 6.3 we show the common experimental setup used in the rest of the experiments in this thesis.

6.1 Overview of the distributed Cache Architecture

We target the cluster of computers system architecture, interconnected with a fast network. The nodes in the cluster can contact any other node of the system, and we define no hierarchy among the nodes. This description ressambles a peer-to-peer architecture, in which we have a high reliability

because there are no single failure points in the system.

The distributed search engine is a collection of replicas of the sequential search engine described in Chapter 4 in each node of the network, which we depict in Figure 6.1. We assume that all the indexes of the collection are shared and are accessible by all the computers. This ensures that an identifier determines a single document for all the nodes of the network.

In each node, we implement a multi threaded search engine. Each query in the system runs in its own thread, so several queries can be simultaneously executed in a node, even if they are running on the same computing block. In our system, the set of CPUs on a node share a waiting queue of pending tasks. We allow one more task running than CPUs in order to avoid many threads competing for the same resources while there is a thread ready to substitute an already finished thread. 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.

This architecture is in accordance to the development of distributed search engines. Large search engines partition the data collection in several blocks that are managed independently [12, 22, 87]. The incoming queries are classified according to the partitions and only the most relevant collections are queried. Each of these partitions can be assigned to a cluster of computers in order to improve the throughput and the system reliability. Our base architecture, targets the optimization of the groups of computers that share a collection index, and hence cooperation becomes fundamental to improve its performance.

6.1.1 Cooperative cache

In each computing node, we allocate a memory pool dedicated to caching tasks, which is implemented as a multi-layer cache such as that described in Chapter 4. This cache is managed locally with the promote/demote policy already described. We extend the cache manager of the local system (see Figure 6.1) to support distributed cache operations, which can retrieve data from a remote node and transfer documents to a different caching node. We define two operations to manage the communication between nodes:

- *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 (c) in Figure 6.1). The request includes the document identifier and a parameter that indicates if the data are needed for the PR or the AE block. The receivers of the request respond with the document if it is available in their caches (operation (d)).

- *Forward*: this operation transfers the least recently used cache entry from a layer to the same layer of another node in the network.

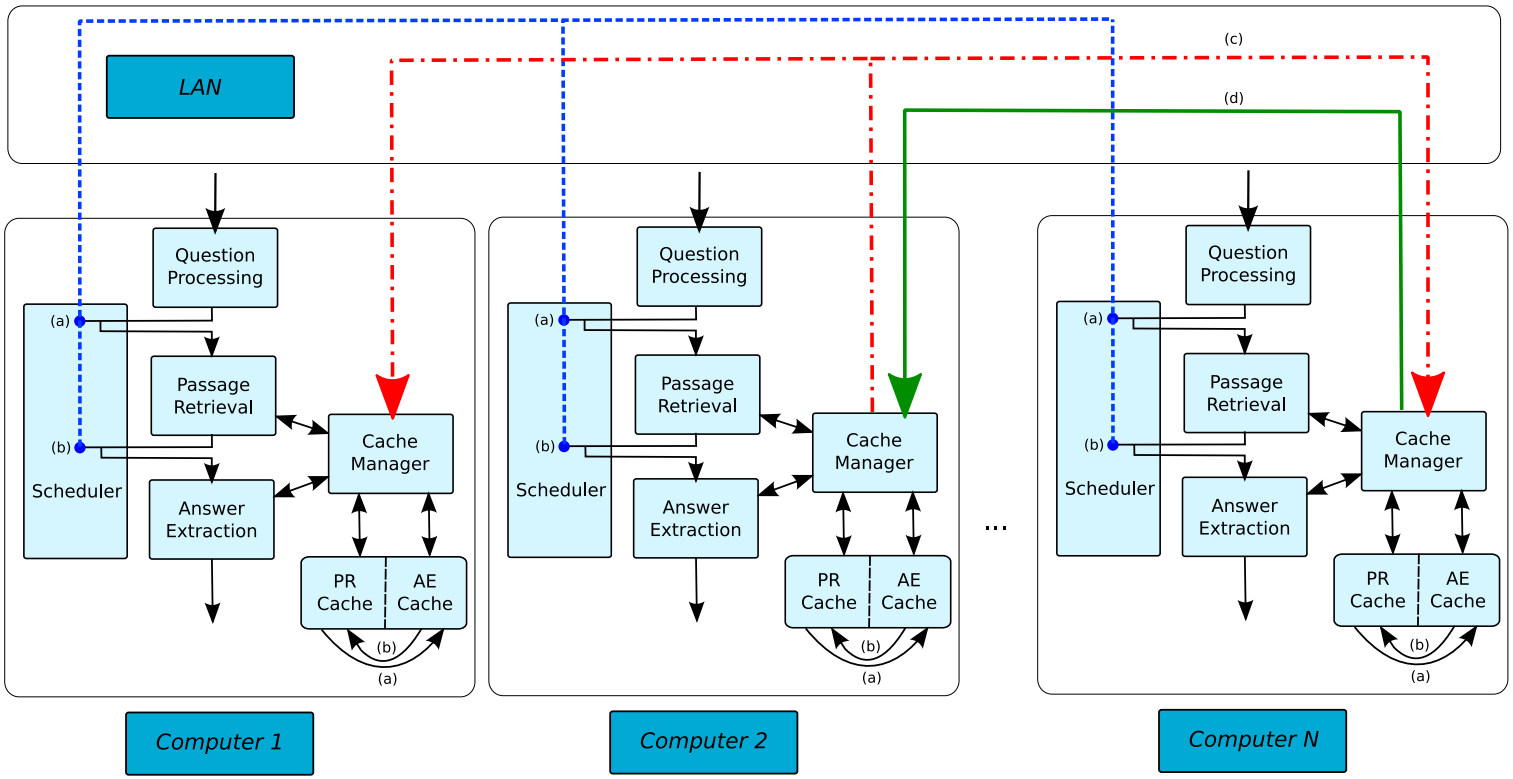


Figure 6.1: *Diagram of the three computing blocks of our QA system: QP, PR and AE. We implement two scheduling points, for PR (a) and AE (b), where the execution can continue locally or be forwarded to a different node. The cache manager stores the documents retrieved from disk in PR, and the processed data generated in AE. (c) Request a document in the cooperative cache. (d) Send the requested data.*

6.1.2 The Scheduler

We also extend our QA system to implement a load balancer, which we call the scheduler (see Figure 6.1), which can transfer the computation of a query from one node to another in order to balance the overall load. The scheduler is able to pick the state of a query at the beginning of a computational block and delegate the computation to the remote node. Our implementation of the scheduler is not preemptive and cannot interrupt the execution of a query in the middle of a computing block. For example, the scheduler is able to stop the execution of a query before starting the computation of the PR block in a node, and resume the PR computation in a different node without recomputing again QP. But once the computation in the PR block has begun, the query cannot be rescheduled until it reaches the next computing block (AE).

The scheduling points: We add two scheduling points to the system, which are depicted in Figure 6.1. The PR scheduling point (a) is triggered after query q reads the indexes from the document collection; it 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. We consider these to scheduling points because, as already mentioned, PR and AE are the most expensive tasks of the QA system with more than 95% of the execution time. Nevertheless, in Section 9.3, we evaluate the adequacy of including more scheduling points in a QA system. In our architecture, the queries that reach a scheduling point know the set of document identifiers involved in the current task of the query, before any expensive computation starts within the task. 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. The data unit in QP is the query: $Data_{(QP,q)}$ is 1.

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 it. 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 the number of forwards per computing block to one , i.e. a query assigned to a node for AE cannot be forwarded again to compute the AE block. However, it is possible for a query to be 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 ($Load^{IO}_{(i)}$) and another one for the CPU ($Load^{CPU}_{(i)}$). Each node sends its load measure to the rest of the nodes in the network periodically, or if their current value differs in more than fraction since its last update. Summarizing, all the nodes compute their local load, and receive recent load stats from the rest of the computing nodes.

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

$$Load^{CPU}_{(i)} = \sum_{q \in i} \left(C^{CPU}_{(task(q))} \cdot Data_{(task(q),q)} \right),$$

where $C^{CPU}_{task(q)}$ 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^{CPU}_{QP}, C^{CPU}_{PR} and C^{CPU}_{AE}. The system records the time spent in each different computing block of the recent queries answered by the node, and sets $C^{CPU}_{(task(q))}$ as the average time spent by the previous queries. Note that even if two queries were in the same computing block, the load contribution from a query that accesses a large number of documents would be larger than for a query that accesses a few documents because the number of units to process, $Data_{(task(q),q)}$, is larger. A similar procedure is used to calculate $Load^{I/O}_{(i)}$, and all the associated information related to I/O.

6.2 Evolutive Summary Counters

The Evolutive Summary Counters (ESC) are a data structure containing a set of k Count Bloom Filters. A Count Bloom Filter (CBF) [45] is a

counter-oriented extension of the Bloom Filters[1] [19], in which each presence bit of a Bloom Filter is substituted by a bit-counter. A CBF is a compact representation of a given data set, where the counters give an approximation of the number of occurrences of a certain data element in the set. A CBF may be implemented in different ways [1, 30] for different objectives. In this thesis, we use the variant introduced in [1]: the Dynamic Count Filters (DCF). The DCF implementation uses counters that adapt to the size of the values they store to reduce the size of the CBF structure.

The k CBFs in ESC are organized as a circular list. 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.

The objective of the Evolutive Summary Counters in distributed caching is to record a window of the history of the local accesses to the documents of a collection. We deploy one ESC in each node of a distributed system to record the number of accesses to the documents in different time windows. 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, a node receives one ESC-summary from each of the rest of nodes, and the distributed algorithms check the summaries to obtain a description of the system state in order to update their decisions dynamically.

The ESC-summary is computed as the aggregation of the k Count Filters of the ESC (see [21] for a detailed description of how to aggregate structures based on Bloom Filters). This aggregation may be a simple addition or a weighted addition. The ESC-summary generated is also a CBF. 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.

[1] A bloom filter is a data structure representing membership for a set of elements. It supports constant time operations for insertion, deletion and search. The filter locates its elements through multiple hash accesses, and if there are multiple hash collisions among different elements of the set then some elements may be detected incorrectly as members of the set, which are called false positives. The probability of errors can be reduced arbitrarily by enlarging the bloom filter for a fixed given dataset size. The use of approximate information reduces the size of the data structure, and computer applications in which the metadata size is critical may benefit from Bloom Filter based structures [21].

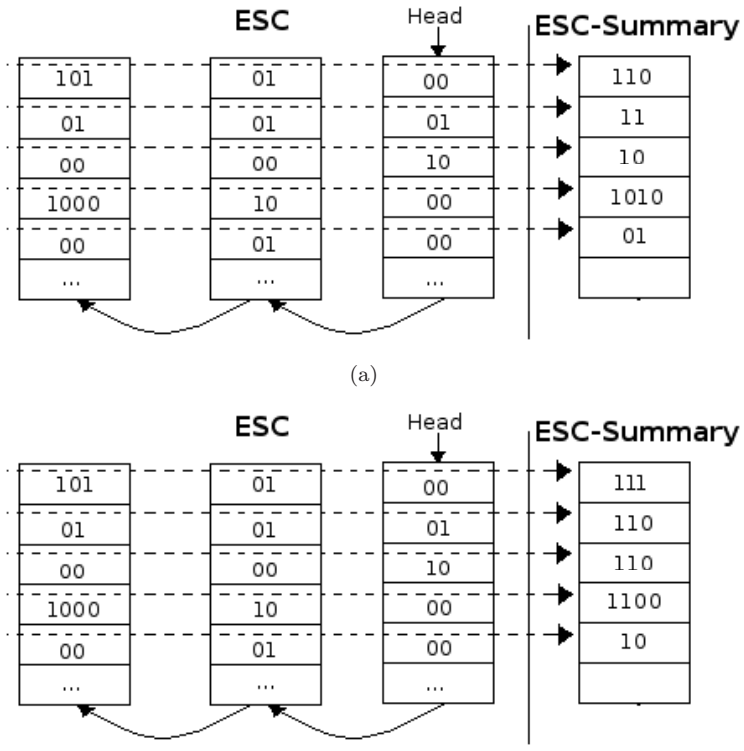


Figure 6.2: *Diagram of the Evolutive Summary Counters data structure for $k = 3$ and the ESC-summary building process for the plain summary (a) and the linear summary (b) implementations. Counters are in binary.*

We depict a diagram of an ESC in Figure 6.2 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 target architecture for ESC is inspired by the design of distributed search engines: a huge data collection is divided into several partitions [87], each of these managed by a separate group of nodes that, with the objective of improving the performance of the system, execute queries related to their partition (with the aid of the ESC) and share a local area network. For other configurations with large pools of nodes, it would be necessary to partition the nodes into teams to avoid excessive network traffic from the ESC-summary diffusion. Note that the ESC only use the network during the periodic ESC-summary update: a node can check at any moment the number of times a document has been accessed in the recent time in a certain neighbor node without any new communication.

6.2.1 Implementation details

In our system, the ESC are communicated using the broadcast address of the network and the UDP protocol. In order to be transmitted, the ESC-summary is serialized to an array of bytes. This array is partitioned into several blocks (with a maximum size equal to a single UDP packet), which are numbered sequentially starting from one. These set of packets is broadcasted using UDP and reassembled by the rest of peers in the network.

Since the UDP protocol does not give any guarantees of receival, we implement a simple policy to recover most of the lost packets. If one of the peers detects that one of the blocks of the message is missing, it will send a "retransmission request" message to the ESC-summary sender. Then, the ESC-summary sender will retransmit the summary again up to a maximum number of three attempts.

6.3 Evaluation of the distributed proposals

In this chapter, we have described the architecture of our distributed QA system and have introduced the different components that constitute our architecture. The evaluation of the techniques is modular and seeks to capture the impact of each of our proposals individually, as well as the final result of combining all of them.

In Chapter 7, we focus on the cooperative cache manager: we propose ESC-placement and ESC-search. Then, we evaluate them with no load balancing policy activated. Next, in Chapter 8, we focus on the scheduler: we propose Probability Cost and Affinity, and we evaluate them without the new cooperative caching algorithms in order to understand the impact of our load balancing proposals. Finally, Chapter 9 takes a holistic approach and evaluates the system as a whole: (i) we evaluate the configuration of ESC in order to improve the performance of our system with all the enhancements activated; and (ii) we study the interaction between our new proposals for the cooperative caching and load balancing.

Basic experimental setup: In the rest of Part III, we use a common workbench to test our proposals unless stated otherwise in the experiment description. For our test, we run the QA system described in this chapter 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 with 2GB of RAM. The default configuration allocates a multi-layer cache in each node with 6500 documents in the PR layer and 650 passages. Note that a passage is five times larger than a document, and this partition corresponds to the assignation of approximately 35% of the available cache for AE and 65% for PR, which fits the recommendations from Chapter 5.

In our experiments, we use the evaluation data sets provided by the TREC conference in the QA track of TREC-8 and TREC-9. Our document collection has approximately 4GB of text in one million documents that make up the TREC-QA document collection [78]. Each of the nodes in the system has a copy of the textual repository in its local disk. The storage is a single IDE hard disk unit with a capacity of 80 GB (Hitachi HDS72168). As already mentioned in the introductory sections, the typical query logs extracted from search engines follow zipf distributions. Therefore, we select from the TREC QA evaluation sets those questions that our system is able to answer (700 different queries) and we generate several query sets of 5000 queries following: $\text{Zipf}_{\alpha=0.59}$, $\text{Zipf}_{\alpha=1.0}$ and $\text{Zipf}_{\alpha=1.4}$ distributions. An additional computer, with the same hardware as the servers and connected to the same network, is used as a client. The client issues each new query to a different computer in a round robin fashion, which simulates the query repartition generated by round robin DNS. The reported results correspond to the average of three runs of the same experiment.

6.4 Summary and conclusions

In this chapter, we propose the Evolutive Summary Counters, which are the core of our distributed algorithms. This is an efficient data structure for capturing the approximate frequency of accesses to the documents from big text collections. We introduce the summarization method to build the ESC-summaries that will be exploited in the following chapters of this thesis.

We present the architecture of our distributed QA system. First, we describe the multithread version of the system introduced in Chapter 3. Then, we show the basic operations of the cache manager – request/response and forward – to operate on the cooperative cache. Finally, we describe how the system measures the CPU and I/O load in each node and how it reports it to the rest of computing nodes.

We also present the general evaluation workbench for our proposals in Part III of this thesis.

Chapter 7
Cooperative caching

This chapter describes the cooperative caching management proposed in this thesis, which relies on the ESC introduced in the previous chapter. First, we propose a strategy to distribute the contents of a cooperative cache dynamically, ESC-placement, which improves the data locality and the system hit rate and we evaluate it. Then, we propose ESC-search, which is an algorithm to locate the cache contents available in the cooperative cache, and we compare it to summary caches and analyze its computing costs. Finally, we show the interaction between ESC-search and ESC-placement and we propose a model to compute the communication overhead introduced by the ESC, and the number of messages sent by ESC-search.

7.1 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 cached document 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 document being processed as the caching node and transfers the document to that node. If the document is already present in the receiving node, our algorithm marks the document as the most recently used entry. Also, if more than one server has the same largest value, a random selection among these nodes is performed.

Apart from the CBFs in ESC, we have a 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

```
Input: Map<IpAddress, ESC> escMap, List<IpAddress> serversAvailable,
       Document cacheVictim
Output: IpAddress
if (cacheVictim.forwardCounter > MAXIMUM_FORWARDS) then
    // Do not forward if the cache victim exceeded the number of
       allowed forwards
    return NULL;
end
mostAccessed := NULL;
mostAccessedFrequency := -1;
foreach (IpAddress currentAddr : serversAvailable) do
    // Iterate and find the most accessed node
    currentESC := escMap.get(currentAddr);
    currentFrequency := currentESC.frequencyCount(cacheVictim.id);
    if (currentFrequency > mostAccessedFrequency) then
        mostAccessed := currentAddr;
        mostAccessedFrequency := currentFrequency;
    end
end
return mostAccessed;
```

Algorithm 3: Pseudocode of the ESC-placement node selection

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 by Dahlin et al. for the n-chance forwarding algorithm [34]. The full algorithm is listed in Algorithm 3.

7.2 Placement analysis

In this section, we compare the hit rate and the performance achieved with ESC-placement against the most relevant proposals in the research literature (see Chapter 2). We compare ourselves with Expiration Age, which is the most recent proposal for cooperative caching. We also compare ourselves with Broadcast Petition Recently, which is the best alternative published for QA. Additionally, we also select n-chance forwarding as a cooperative caching baseline because it is one of the first proposals of cooperative caching algorithms. And finally, we also report the cache performance of a non cooperative caching alternative. The details for each configuration are the following:

- *Local*: We allocate a multi-layer cache as described in Section 4.1, but we disable the remote operations of the cache manager. This configuration corresponds to a non-cooperative cache where the memory management is local in each node.

- *Random server selection (RSS)*: When an entry is selected for leaving the local cache, it is forwarded to a random server like in n-chance forwarding [34]. Each entry can be forwarded a limited number of times, since its last access, until it is discarded (which in our configuration is two times, according with [34]). A local copy is created for all documents that have been requested remotely.

- *Broadcast Petition Recently (BPR)* [38]: After a successful remote hit, the protocol creates a local duplicate of the remotely accessed page only if that entry has been accessed more than once in a fixed window of time. No forwarding is performed when an entry is removed from the local cache.

- *Expiration Age (EA)* [91]: This algorithm is based on 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. If the requester has a larger expiration age, then the requester stores a copy of the document and the responder does not update its LRU[1]. Otherwise, if the responder has a larger expiration age, the requester does not store a copy and the responder puts the document at the top of its LRU.

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 [110]. If any remote node has the needed contents, it sends them back to the requester. The broadcast guarantees to find a document if it is available in any node of the network. We observed no significant differences between plain and linear ESC-summaries for ESC-placement, and hence, we do not separate them in the placement experiments.

In these experiments, we evaluate the system with the yamcha NERC and the system configuration described in Section 6.3. The *plain* and the *linear* summary strategies performed similarly in the placement experiments. The results reported in the rest of this section come from the *linear* summary implementation.

We compare the above algorithms in different scenarios to test the performance of the system for different query distributions, cache sizes and search engine complexity. For each of those dimensions, we perform one experiment in which we vary the dimension tested.

[1]In [91] the experiments are applied to a local LRU cache policy. In the same article, some adaptations are described for other cache policies. We do not consider them because the implementation of the local cache policy is not the focus of our research. Thus, we only implement LRU, which is the local replacement policy that we apply to all caches.

7.2.1 Query distribution

In this experiment, we test the performance of the question answering system for a variety of query distributions. According to the previously mentioned studies, we generate three query sets following different Zipf distributions with varying parametrization: $\text{Zipf}_{\alpha=0.59}$, $\text{Zipf}_{\alpha=1.0}$, and $\text{Zipf}_{\alpha=1.4}$.

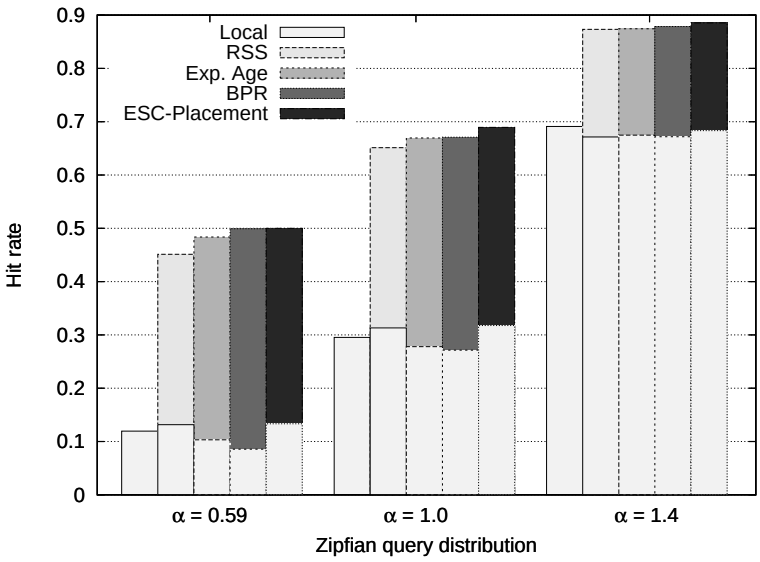
Figure 7.1(a) shows the hit ratio for the different algorithms tested. We plot one bar for each different algorithm tested and query distribution. The bars are grouped by the different distributions under test: the most skewed distribution corresponds to the rightmost group of bars. For the cooperative 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 full bar accounts for the addition of the local plus the remote hit rate. The remote hit rate corresponds to the darker part of each bar.

We observe that for all the algorithms the query distribution significantly affects the total hit rate, 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 cache 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.

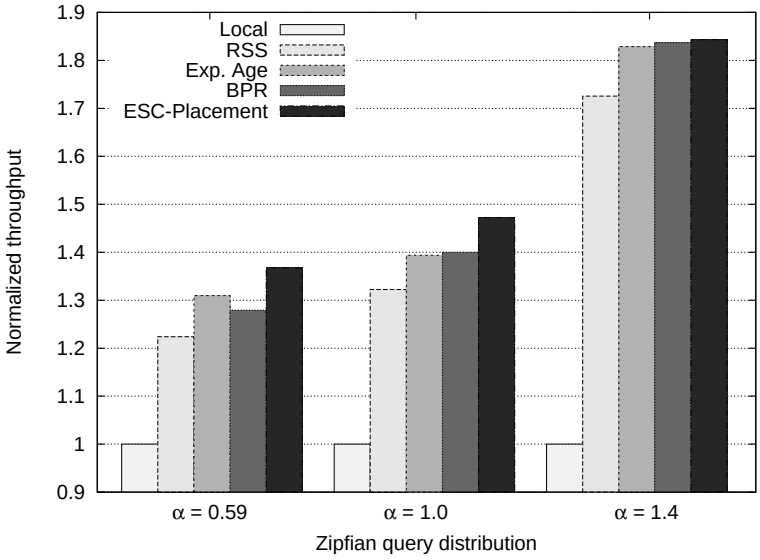
When looking at more skewed distributions, the ratio between local and remote hits changes. In the case of the $\text{Zipf}_{\alpha=1.4}$ distribution, we observe that most data can be retrieved from the local cache, though the cooperative cache still contributes to improving the hit rate.

In Figure 7.1(b), we show the execution time of the different algorithms. The hit rate has a direct influence on the system performance: cooperative caching algorithms are much faster than local caching. This huge difference can be used as a strong recommendation for the use of cooperative caching algorithms in a complex search engine.

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, with a throughput above 1.84 times a locally managed system for highly skewed query sets. We also observe that the local hit rate of ESC is superior to other cooperative caching algorithms, such as EA or BPR, because ESC not only encourages global hit rate but also locality: it sends a document to the node that has done



(a)



(b)

Figure 7.1: *ESC-placement performance for different query distributions. (a) Hit rate. (b) Throughput.*

most accesses to this document.

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 uses the remote hits more often than other algorithms such as EA or ESC-placement because it only replicates a document

locally if it has been accessed at least twice recently. Due to the additional network traffic of remote hits, we see in Figure 7.1(b) that BPR throughput is significantly smaller than for ESC-placement, specially for small skews. In our tests, Expiration Age obtained better performance than BPR for the less skewed configuration because its local hit rate is better than for BPR, but it is still slower 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 very high 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[2] 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 the additional network connections. All in all, ESC-placement obtains the best throughput of all placement algorithms investigated 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.

7.2.2 Cache size

In this experiment, we use Zipf$_{\alpha=1.0}$ as the query distribution, and we test several cache sizes. Figure 7.2 shows the hit rate of the cooperative caching algorithms. The largest cache size tested, which corresponds to a cache size of 13000 documents, has enough room to store approximately 12% of the requested documents in a node. We plot two lines for each cooperative caching algorithm: we use large points for the global hit rate and smaller ones for the local hits. The figure shows that the improvement of all the cooperative caching algorithms grows logarithmically towards the hit rate obtained with an infinite cache, 0.92. The local cache policy is far below the cooperative caching algorithms for all the tested configurations. Besides, the local cache hit rate improves at a slower rate than the cooperative caching global hit rate.

Similarly to the previous experiment, ESC is the best algorithm for all configurations, and BPR is very close for the configurations with the largest cache size. However, as in the previous algorithm the BPR locality is smaller because BPR is not aware of the contents of the nodes. ESC is the algorithm with the best cache locality, with a relative improvement of up to 30% over

[2]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 obtained that the optimal hit rates were 0.86, 0.92, 0.95 for α=0.59, 1.0 and 1.4, respectively.

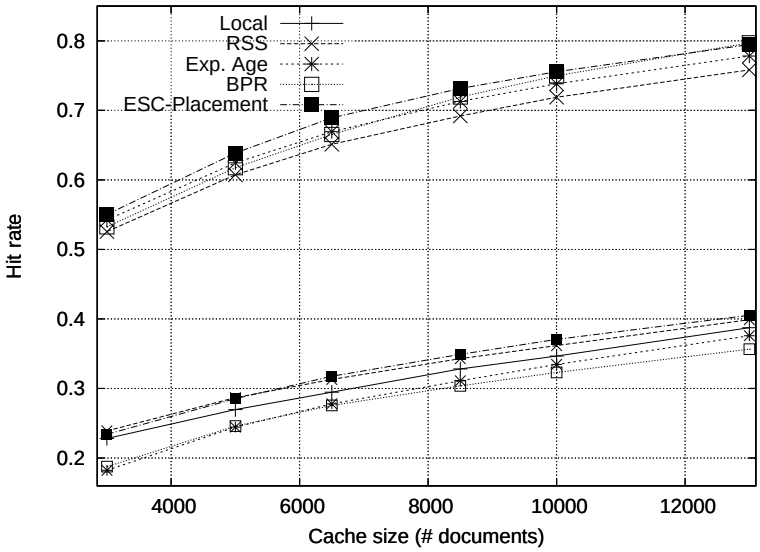
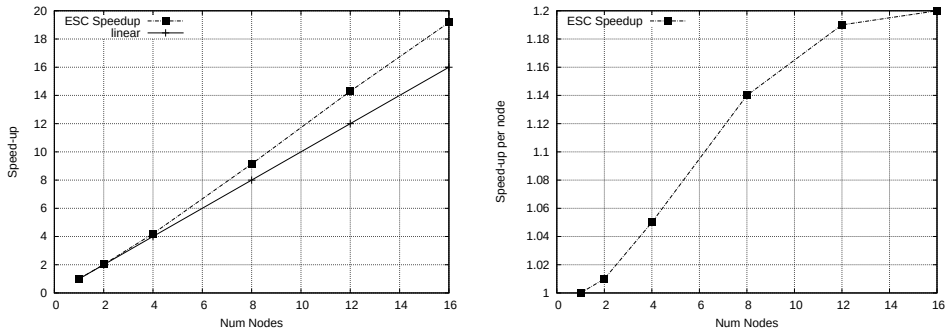


Figure 7.2: *Hit rate of the cooperative caching algorithms for different cache sizes*

other algorithms such as EA or BPR. We observe that for small caches it is more difficult to keep the balance between a good local hit rate and the cooperative cache, and here we observe that the ESC-summaries contain essential information to improve the data placement.

7.2.3 Speed up

Figure 7.3(a) shows the speedup of the QA system as the number of processors grows from one to sixteen computing nodes. The speedup is measured relatively to the system with a local cache in a single computer. The performance gain is above linear (which is shown as a solid line) and increases with the addition of new computers. Note that the speedup increase is better than linear due to the Zipf question distribution, which favors repeated questions, and due to our distributed caching algorithm, which efficiently manages these questions. In our experiments, the round robin algorithm balances the load in the cluster reasonably well because the number of queries is not too small, and all the queries were generated following the same probability distribution. So, even with just a local cache, a result not too far from a linear speedup would be expected. The right plot of Figure 7.3(b) shows the speedup per node for the left plot of the same figure. The benefit rates per added node are: 1.01 for two nodes, 1.14 for eight, 1.19 for twelve, and 1.2 for sixteen nodes. The reason for this increase is that a larger number of computers implies that more cooperative memory



(a) (b)

Figure 7.3: *Speedup of the ESC-placement, compared to a linear speedup (a). Speedup per node added (b). The question distribution follows a* $Zipf_{\alpha=0.59}$.

is available. Consequently, an effective cooperative caching algorithm can store more documents and increase the hit rates. In the case of 16 nodes, the progression of the benefit ratio (1.19 compared to 1.2) is smaller because we are getting closer assymptotically to the maximum hit rate achievable with an infinite cache.

7.2.4 Computational cost

Depending on the search task and the quality of the output, the modules that process the results of a search engines 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 search engines. In order to simulate the complexity of different search engines, we replace our passage analysis module with two alternatives. The first corresponds to a system that has a more complex analysis than our initial system. We implement a second natural language processing module with more complex features, based on support vector machines, which has a processing overhead approximately three times larger than our initial system [106]. This system is not adequate for interactive question answering, but simulates a system in which the user might sacrifice response time by a better answer quality. As a third testing system, we remove the natural language processing module. 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.

Figure 7.4 shows the normalized execution time 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. In other words, in a simple system, the cost to communicate the data through the network is similar to the cost of processing it. We observe that RSS has an even worse performance than a locally managed system because although RSS has better hit rate, the cost to pack, transfer and receive victims through the network is not overcome by the additional cooperative cache hits. BPR and EA are cooperative caching algorithms that do not forward the cache victims. Hence they save network time and their performance surpasses both RSS and the local system. ESC-placement forwards cache entries, like RSS, but ESC-placement queries fewer nodes than RSS: cache victims are usually related because they were requested by the same query, and they can be transferred to a node where a similar query was issued recently. This reduction in the number of connections plus the improved hit rate makes ESC-placement better than other algorithms. However, the benefit of cooperative caching for light systems is limited. This conclusion is in accordance with the models published by Wolman et al. [112] where they found that only applications with computationally expensive tasks significantly benefit from a better cooperative caching algorithm.

For computationally expensive systems, caching is fundamental to improve the throughput if network penalty is not severe. Out of the cooperative caching algorithms investigated, ESC-placement performs the best because it obtains the highest hit rate with low network overhead. Furthermore, the difference between ESC-placement and the local cache model 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.

7.2.5 Collection preprocessing

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

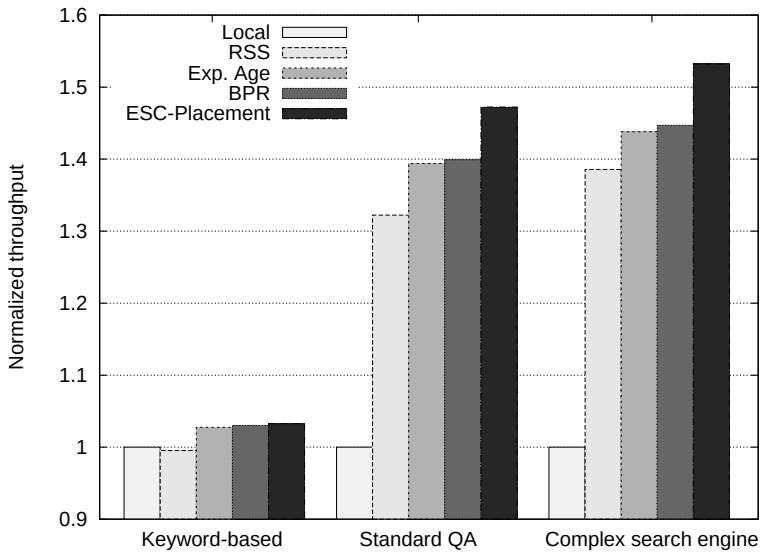


Figure 7.4: *Normalized throughput of search engines with different degree of complexity (Zipf$_{\alpha=1.0}$*

caching proposals for such an architecture.

In Figure 7.5, we compare the throughput of a system that stores a copy in disk against one that preprocesses the documents. We observe that for our hardware setup the throughput increases, if we are able to preprocess the data rather than to analyze it on the fly like in previous experiments. Anyway, we observe that ESC-placement is still faster than the local policy for preprocessed collections because ESC-placement is able to retrieve documents from remote memory, which reduces the traffic from disk. This result indicates that ESC-placement is also adequate in environments where we are able to preprocess the collection beforehand or even implement caches on disk that can be much larger than in memory caches.

The configuration in this experiment obtained the largest throughput among our tests. Nevertheless, we note that although preprocessing might be valid for some scenarios, it may not always be feasible or advisable for QA. Some of the main reasons are the following:

(a) Document collections are huge and are always growing. For example, Google recently quantified the number of different web sites crawled as more than one trillion, which makes it prohibitive to have it analyzed with NLP tools.

(b) Many queries appear only once in a real workload of a search engine because of the long tail of the Zipf distribution, and thus, it might not be worth to preprocesses the whole dataset given the amount of unique

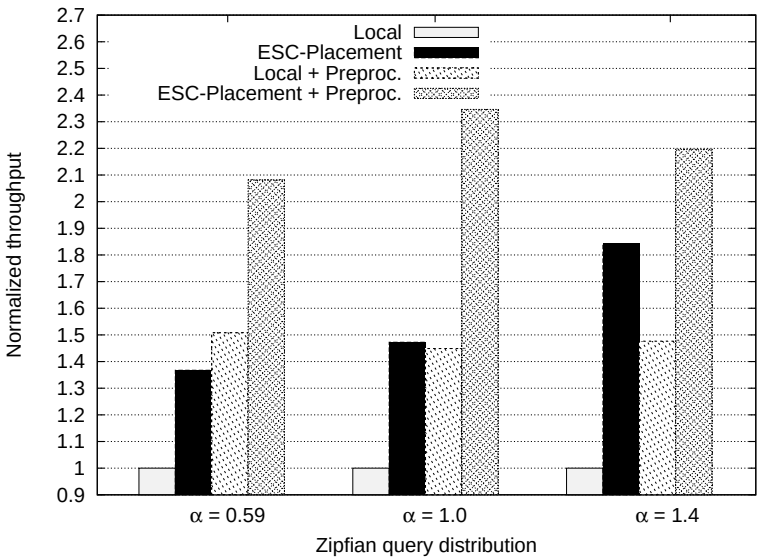


Figure 7.5: *Normalized throughput for system that has a preprocessed version of the NLP analysis of a QA system.*

queries and the cost to process each document (that is significantly more expensive than building current IR inverted indexes). For instance, Baeza et al. found that 50% of the queries received in the Yahoo UK web search engine during a whole year are unique [10].

(c) If the processed data is stored on the disks, then the bottleneck of the system is the I/O from disks in order to supply data for both PR and AE computing blocks. Although in our configuration, (which is able to compute two threads simultaneously) it is faster to preprocess documents and store them on disk, this might not hold in architectures with a larger degree of paralelism. Besides, the current trend in the last years is that the number of CPUs is growing at faster rate than the disks are. For example, a general purpouse processor, such as UltraSparc T2, can run 64 threads simultaneously [103], and a GPU, such as Nvidia GeForce GTX 295, has 480 CUDA cores [79]. This trend suggests that reanalyzing documents might be faster (except for the very complex processes) than reading preprocessed data (thus, more bytes) from a disk cache.

7.3 ESC-Search

The search algorithm is in charge of locating the node that is currently storing a certain document in its local cache. 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 create a bottleneck. As an alternative to reducing the number of communications, we compare two methods which take into account the frequencies stored in the ESC-summaries to reduce the number of communications. The first method is static and is the baseline for comparison; the second method is dynamic and adapts to a changing query distribution.

In ESC-search, once a document is not found locally, the system traverses the set of ESC-summaries and gets the frequency of the missing document in each node. Then, it generates a list of all the nodes in the system (L=$n_1, ..., n_N$) sorted by decreasing order of the frequencies. Up to this point the operations can be computed locally. Then, the search method requests the document to the first h nodes from the list. The compared methods differ in the procedure for choosing h:

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

Dynamic: h is determined every time a document is searched as a function of the probability that the document is in a certain group of nodes. This estimation is used to keep the probability of an avoidable miss below a defined threshold. Using this threshold, only the smallest number of nodes that keeps the probability below the threshold is queried.

After a local cache miss, the dynamic search estimates the probability of an avoidable miss ($P_{AvoidableMiss}(s,d)$), if the first s servers from L were queried. The probability is calculated for all possible values of s, from one node to all the available nodes in the network, using the following formula[3]:

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

where $ESCS(i,d)$ is the value in the ESC-summary received from the i^{th} server for document d; s is the number of servers that are queried to locate

[3]We assume that the probability of finding a document in one node is independent among nodes.

```
Input: Map<IpAddress, ESC> escMap, List<IpAddress> serversAvailable,
       int documentId
Output: List<IpAddress>
List<IpAddress> serversToQuery := { };
// Sort the list of servers by the frequency of the ESC
serversAvailable.sortByFrequency(escMap, documentId);
i := 0;
probAvoidableMiss := probAvoidableMiss(escMap, serversToQuery);
while (probAvoidableMiss > ϵ ) AND i < serversAvailable.size() ) do
    // Update the avoidable miss probability until we reduce it
       under ϵ
    currentIp := serversAvailable[i];
    serversToQuery.add(currentIp);
    probAvoidableMiss := probAvoidableMiss(escMap, serversToQuery);
    i++;
end
return serversToQuery;
```

Algorithm 4: Pseudocode of the ESC-search node selection with the dynamic policy.

document d; and $P_{freq(x)}$ is the probability that a document with frequency x is still available in the memory of the remote node. h is selected as the smallest number of servers such that $P_{AvoidableMiss}(s,d) < \epsilon$, where ϵ is an upper threshold on the allowed probability of an avoidable miss. We show the pseudocode for this algorithm in Algorithm 4.

In order to evaluate $P_{AvoidableMiss}(s,d)$, each server must keep 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, a node that has sent five requests for documents with frequency 2 and only found one out of the five, sets $P_{freq(2)}$ to $\frac{1}{5}$. After each search, this value is updated according to the success of the search. For instance, if the previous server requests another document with frequency 2 from a node and finds it, then $P_{freq(2)}$ will be set to $\frac{2}{6}$ because it found the documents in two occasions out of six attempts.

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 where n is the number of servers in the network because the algorithm checks the summaries from all nodes for a given document. Nevertheless, $P_{AvoidableMiss}(s,d)$ is a monotonic decreasing function so that as soon as it reaches a value below ϵ, this becomes the final result, and there is no need to compute further it for the rest of the nodes in the network.

ESC-search example: In this example, we show the steps that ESC-search (configured with $\epsilon = 0.10$) performs to locate documents in a network with four computers. The initial state of the system is depicted in Figure 7.6(a), in which *Node* 1 is computing a query that reads "Doc B", but it is not

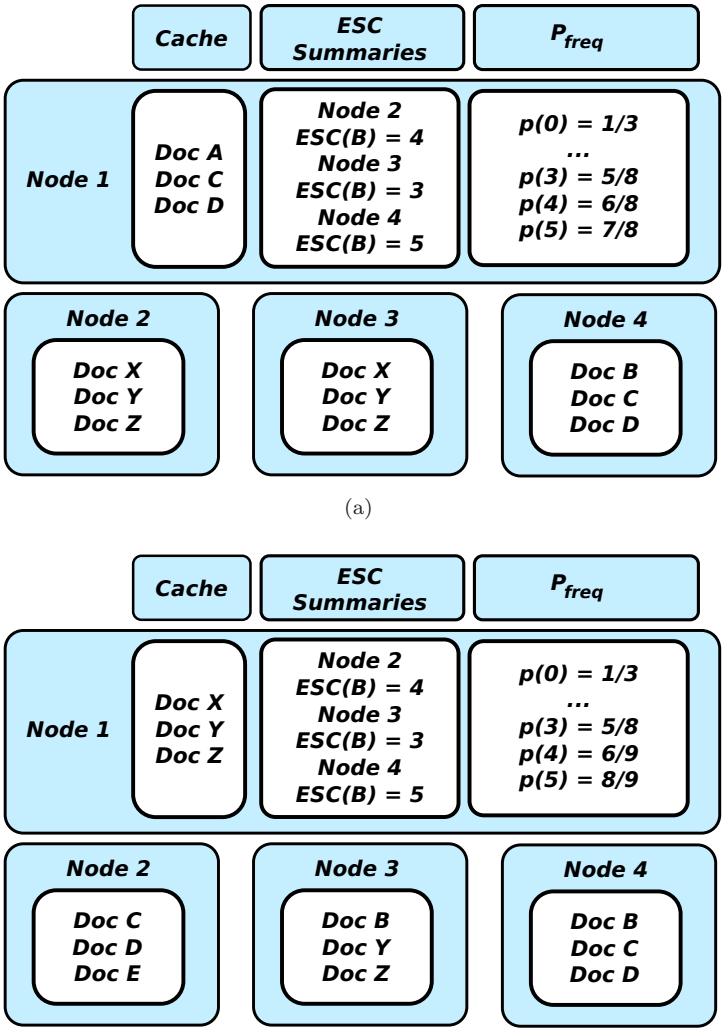


Figure 7.6: *Example of ESC-Search.*

cached locally. The node tries to locate the document in the cooperative cache. First, $Node\ 1$ sorts the server list according to the P_{freq}: L=$\{Node\ 4$, $Node\ 2$, $Node\ 3\}$, where $Node\ 4$ is the server with the highest probability of storing a copy of "Doc B". Then, $Node\ 1$ calculates the probability of an avoidable miss when no server is queried:

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

Since $P_{AvoidableMiss}(\{\}, \text{“Doc B”}) > 0.10$ (in other words, it is likely that the document is available in the cooperative cache), $Node$ 1 estimates the probabilities of an avoidable miss if more servers were queried. The first node that is included is the one which is more likely to store the document, which is $Node$ 4 because “Doc B” has been recently accessed 5 times and its estimated miss probability is the largest:

$$\begin{aligned} P_{AvoidableMiss}(\{Node\ 4\}, \text{“Doc B”}) &= \left[\prod_{i \in N4} \left(1 - P_{freq(ESCS(i,d))}\right)\right] \cdot \\ &\quad \left[1 - \prod_{i \in N2,N3} \left(1 - P_{freq(ESCS(i,d))}\right)\right] \\ &= \left[\frac{1}{8}\right] \cdot \left[1 - \frac{2}{8} \cdot \frac{3}{8}\right] = 0.11. \end{aligned}$$

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

$$P_{AvoidableMiss}(\{Node\ 4, Node\ 2\}, \text{“Doc B”}) = \left[\frac{1}{8} \cdot \frac{2}{8}\right] \cdot \left[1 - \frac{3}{8}\right] = 0.02$$

The algorithm finishes here because it estimates that the probability of an avoidable miss is 0.02, if nodes 2 and 4 are queried, which is sufficient to satisfy our probability miss requirements. Note that up to this point there has been no communication among nodes because the ESC-summaries from the other nodes are already stored in $Node$ 1. In order to retrieve “Doc B”, $Node$ 1 sends a request message to nodes 2 and 4 and discovers that the document is only available in $Node$ 4, which transfers it to $Node$ 1. Finally, $Node$ 1 updates the values of P_{freq} according to the final result: (a) it decresases $P_{freq(4)}$ to $\frac{6}{9}$ because the document was not found in $Node$ 2 (where “Doc B” was accessed four times; and (b) it increases $P_{freq(5)}$ to $\frac{8}{9}$ because the document was found in $Node$ 4 (where “Doc B” was accessed five times).

Later, $Node$ 1 computes another query that requests “Doc B”, which again is not available locally as depicted in Figure 7.6(b). The steps are similar to the one described for the previous request but the algorithm finishes after only two steps:

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

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

Therefore, ESC-search would only query $Node$ 4 but not $Node$ 2. We observe ESC-search is an algorithm that adapts to the previous experience of the

system. Given that in the previous search the document was only available in one node, it updated the corresponding $p_{freq(x)}$ and now it is able to reduce the number of severs queried.

7.4 Search Analysis

In this section, we analyze the cache search algorithms proposed in Section 7.3. We use the hardware setup and our QA system, as in the placement analysis, with the addition of the ESC-placement algorithm described in Section 7.1, which was shown to perform the best in the previous section.

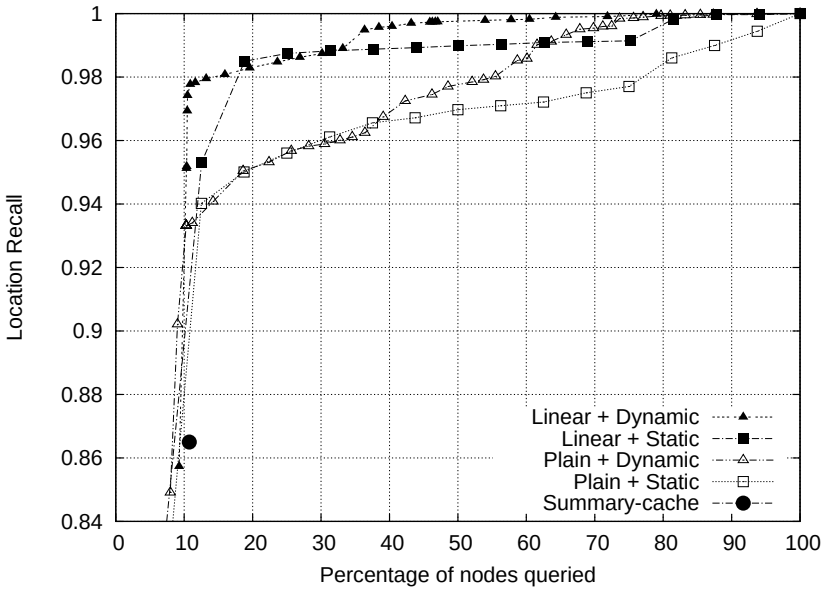
In the experiments, we compare the two possible summary methods, i.e., *plain* and *linear* (see Section 6.2) because they behave with significant difference for search, as opposed to what we did in ESC-placement where we only measured linear summary as they both behaved the same. We also combine the two possible search algorithms described in Section 7.3, *static* and *dynamic*. We compare the four combinations of these techniques with the broadcast protocol.

7.4.1 Location recall analysis

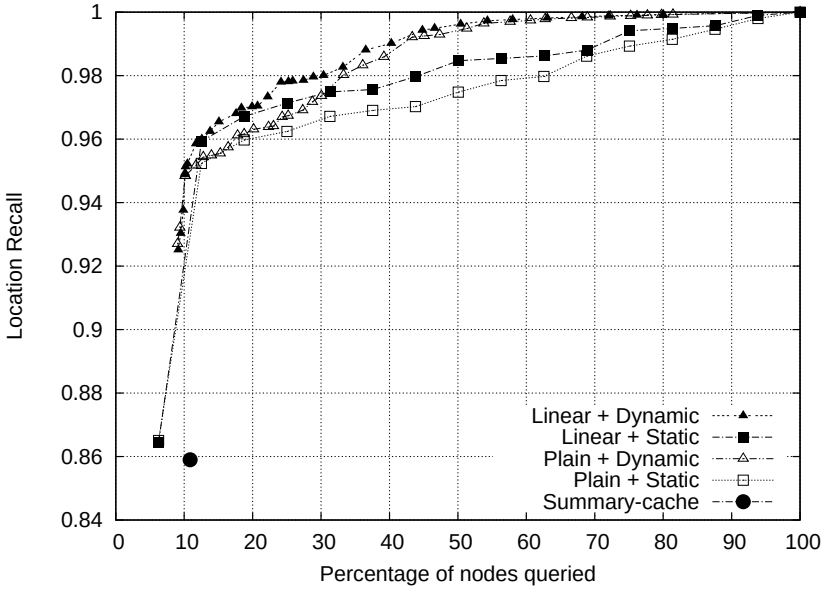
Intuitively, a good search algorithm must obtain a compromise between the number of servers requested, and the *location recall* of the system, where the location recall of a search algorithm is defined as the fraction of documents found remotely divided by the documents found if all the nodes were queried. In this first experiment we analyze the different configurations for ESC-search and compare them to summary caches, proposed by Cao et al [45].

Figure 7.7 shows the location recall for the two different summary procedures (the plain and the linear ESC-summary, see section 6.2) combined with the static and the dynamic location algorithms. The vertical axis is the location recall of the system. 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 servers to a constant. For dynamic policies, each point in the plot corresponds to the average number of servers requested when ϵ is fixed.

Zipf$_{\alpha=0.59}$: In contrast to the results for ESC-placement, ESC-search is very sensitive to the summary method. In Figure 7.7(a), we observe that the linear summaries score a significantly better location recall, specially for configurations that request documents to few nodes, and their recall is very close to requesting all the nodes. For example, the combination of linear summaries with the dynamic node selection (linear+dynamic) gets a 98% location recall with only 14% of the nodes (approximately two nodes) requested on average , and over 99.5% location recall with 36% of the nodes



(a)



(b)

Figure 7.7: *Location recall for the ESC-search algorithms. The query set send to the system follows a $Zipf_{\alpha=0.59}$ (a) and $Zipf_{\alpha=1.0}$ (b).*

(less than six nodes) requested. On the other hand, the plain summary must query more nodes to avoid a large number of avoidable misses: the best of the plain summary implementations (plain+dynamic) sends requests to over 55% of servers (almost nine nodes) and obtains a location recall of 98%, which corresponds to 4.5 times the number of request messages compared to the (linear+dynamic) policy. This difference shows that the temporal information, added by dynamic summaries, is crucial for the efficient search of documents in a distributed cache.

From the point of view of the dynamic and static selection of the number of nodes (h), we see that the dynamic policies converge faster to larger location recalls both for static and dynamic summaries. The adaptability is the reason of this improvement: the dynamic algorithm selects only very few nodes for very popular cache entries (often one is enough), and several nodes for rarer documents.

We also compare our proposal to summary caches, which we configure with the same parametrization as our ESC-summary, in Figure 7.7(a). It fits to remember that the summary cache policy queries all the nodes whose last summary indicates the document was cached. The summary cache policy requests the documents from approximately 1.5 nodes on average to reach a location recall of about 86%, which is far below the result of linear+dynamic that reach almost 98%.

Zipf$_{\alpha=1.0}$: We report the results of the same experiment with Zipf$_{\alpha=1.0}$ distribution in Figure 7.7(b). First, we observe that the previously described trends for Zipf$_{\alpha=0.59}$ are still visible in the plot: linear summaries are better, and dynamic policies converge faster. Nevertheless, we note that for this distribution the importance of the summarization procedure is smaller because the gap between linear and plain summaries is smaller. This makes the use of dynamic policies more relevant. As a result of these observations, the combination linear+dynamic is still the best policy overall, but, unlike for Zipf$_{\alpha=0.59}$, we observe that plain+dynamic is better than linear+static in general.

We see that the location recall for Zipf$_{\alpha=1.0}$ is smaller than for Zipf$_{\alpha=0.59}$ because the task is harder for the former distribution. The local cache policy stores the documents that have rencently been accessed, which approximately corresponds to the documents that are the most accessed according to the distribution. ESC-search is only triggered when the data is not available in the local cache, and thus, the search algorithm has to locate documents in the cooperative cache that typically belong to the tail of the zipf distribution. Since the documents that are unfrequently accessed (which make up the tail of the distribution) are the most difficult ones to locate because they stay a shorter time in the cooperative cache, then the Zipf$_{\alpha=1.0}$ distribution is harder.

The location recall for summary caches is similar for both Zipf distri-

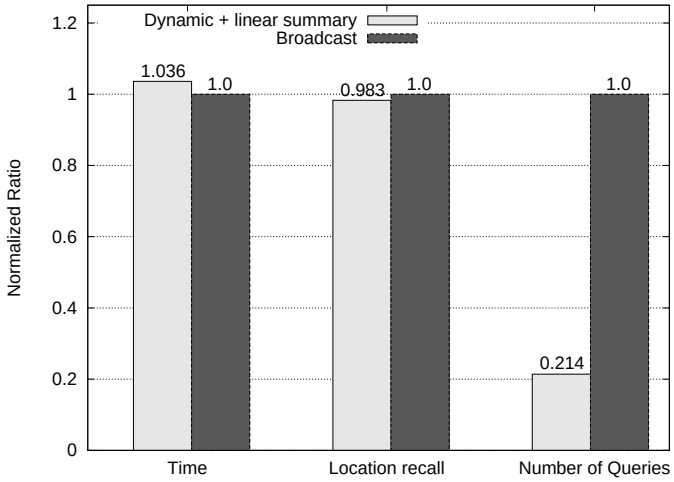


Figure 7.8: *Execution comparison between broadcast and ESC-search algorithm. The time, the hit rate and the number of remote requests are normalized with respect to the broadcast search protocol. ϵ is set to 0.05*

butions and is improved by the ESC-search alternatives. This indicates that ESC-search is taking advantage of the frequency stats provided by the ESC-summaries. Note that even though the summary cache sends smaller updates, because it only registers existence rather than frequencies, ESC-search shares the same information as ESC-placement. Hence, a system that implements ESC-placement can implement ESC-search with no additional communication cost.

ESC-search improves the recall of summary caches because: (i) it is able to detect the nodes that can potentially store a document, even though in the last update, the document was not cached; (ii) if a document is very popular in the network and appears in many nodes, then ESC-search requests it from a reduced set of nodes; and (iii) it benefits from ESC-placement decisions, as we analyze later in Experiment 7.4.3, because ESC-search sends the requests to the nodes targeted by the forwards of ESC-placement, which are the most accessed. Moreover, the ESC-search can increase, up to a virtually perfect recall, the location procedure increasing the ϵ for certain cache items. This is useful, for example, when some cached entries are particularly expensive to compute and we wish to improve their recall as much as possible. Summary caches use all the available information about the cache state because the system already requests all the nodes whose last summary indicated that the document was stored in the local cache. If the node wish to improve the location recall above its initial search, the node must broadcast the request to all the nodes, in contrast to ESC-search, which can improve its recall arbitrarily adjusting ϵ.

7.4.2 Comparison with a broadcast policy

In this experiment, we perform a detailed comparison of the ESC-search algorithm against the broadcast protocol. We note that if the network is not overloaded, the use of broadcasting achieves better performance than any other algorithm because its location recall is 1. Figure 7.8 plots the execution time of the search algorithm compared to the broadcast baseline. Even if all nodes are interconnected with a fast network that supports broadcast, which is our hardware setup, we see that the execution time of the best ESC-search configuration only increases the broadcast approach by about 3%. However, the probabilistic algorithm only queries approximately a fifth of the total number of servers. The ESC-search is more scalable because it reduces the stress on the network, and so it is possible to add many more computers than with the broadcast search implementation. Finally, the limited number of remote requests of the ESC-search make it a viable candidate if not all the nodes are in the same LAN.

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

7.4.3 Influence of ESC-placement in ESC-search

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

We observe in Figure 7.9 that both scenarios share a similar pattern: a steep increase of the slope that converges up to almost a perfect recall when more than one third of the nodes are queried. The location recall for a small number messages is also large enough for most applications: it is over 99% once the system queries approximately 20% of the nodes on average. We also observe that the recall when ESC-placement is activated is better than when no forwarding policy is applied. This happens because ESC-search together with ESC-placement, versus a system where ESC-search is combined with a placement strategy without forwarding, implement complementary policies on top of the same data structures: (a) ESC-placement prefers nodes with more local accesses to the corresponding document, and (b) ESC-search

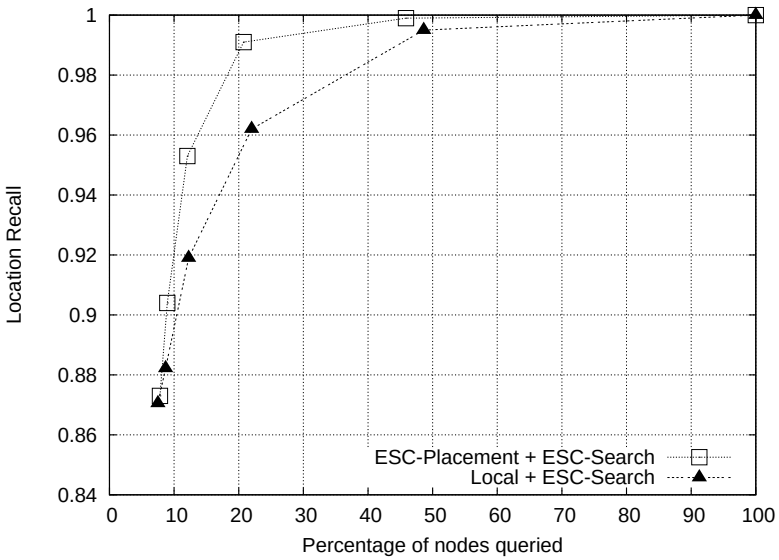


Figure 7.9: *Comparison between ESC-search and summary caches with ESC-placement activated (a) and no forwarding algorithm (b).*

queries first the same nodes with the largest number of accesses to the given document. In other words, ESC-placement tends to forward cache entries to the nodes that will be inspected first by ESC-search.

7.4.4 Search Communication Overhead

Now, we want to quantify the amount of communication introduced by our ESC-search policy, and we compare it to the broadcast policy. In this section, we include the cost of sending ESC-summaries, which is equivalent to the summary communication overhead of ESC-placement.

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

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

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

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

$$BitSend_{ESC} = h \cdot (nd \cdot fr \cdot S_k) + k \cdot (BFSize \cdot log_2(fr) \cdot N)\,.$$

The first term in the summation stands for the number of bits involved in the requests. Hence, we substitute N in the broadcast formula by h here, because the request is sent to a smaller set of servers. The rest of this first term is equal to the broadcast formula. The second term stands for the number of bits sent due to the broadcast of the ESC-summaries. Here, k is the number of CBF in ESC, and $BFSize$ is the number of bits per document in the CBF $\left(\frac{nd \cdot ln(ProbFalsePositive)}{ln(2) \cdot ln(1/2)}\right)$, which can be derived from [19]. Note that $ProbFalsePositive$ is the probability of a false positive in a CBF.

Parameter nd affects equally the two algorithms, thus, in our approach it is not relevant for the comparison. For ESC-search, fr determines the size of the counters of the CBF. In our implementation, we use the Dynamic Count Filters explained in [1]. In DCF, the counters grow logarithmically and dynamically with the size of the value they store.

In Figure 7.10, we plot our network communication model for different setups. We plot the number of bits sent, as a function of the average number of times that each document is requested in the network, fr. The setups shown are for the broadcast policy, and for an ESC-search configuration which queries between 10% and 50% of the nodes (this includes the requests as well as the transmission of the ESC-summaries). The solid area of the plot represents the broadcast of the ESC-summaries, which is the same for the two ESC-search scenarios.

Figure 7.10 confirms that if we increase the average number of requests for the same document, the search based on ESC-summaries is more beneficial than the broadcast policy, even for a considerable number of nodes queried (50%). The number of bits sent due to the requests grows linearly with fr for both scenarios. However, the slope for ESC-search (h) is smaller than for the broadcast policy (N). The number of additional bits sent because of the ESC-summary transmissions does not grow significantly, because the counters grow logarithmically with fr.

The results can be better understood by combining the plots in Figures 7.7 and 7.10. According to the experimental results from Figure 7.7 (where fr can be estimated to be close to 2 for $\text{Zipf}_{\alpha=1.0}$, the location recall is over 97.5% when we query 10% of the available nodes. In Figure 7.10, we see that for this configuration ESC-search achieves a four-fold reduction of the transmissions. If we wish to achieve even higher accuracy (according to Figure 7.7 more than a 99.7% location recall) it is possible to query 50% of the nodes and still save a 40% of the bandwidth (see Figure 7.10).

Furthermore, current network technologies only achieve the highest bandwidths with big network packets, and the ESC-summary is a compact structure that can be transferred in a sequence of packets that get the best performance from the network. ESC-search is also more adaptable when there is temporary congestion in the network because the refresh summary rate and the number of count filters (τ and k, respectively) can be adapted.

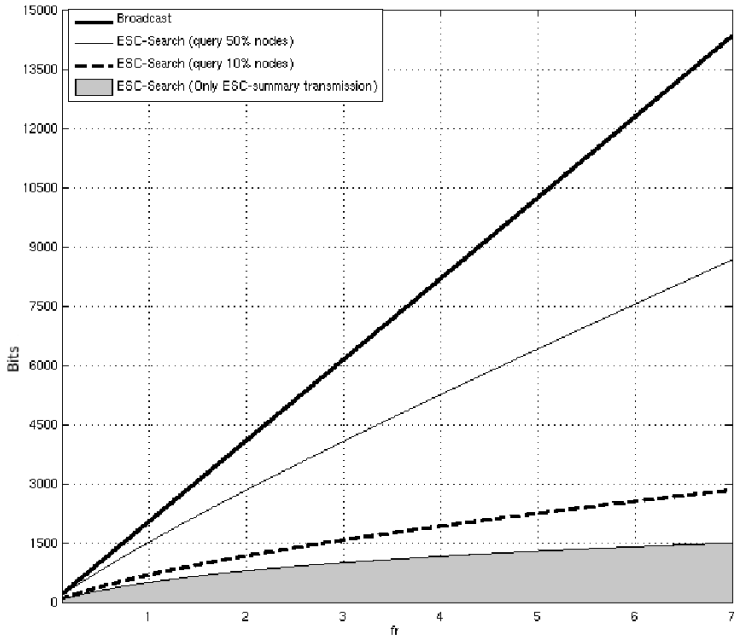


Figure 7.10: *Parametrization of the search model to calculate the bits sent through the network. The model uses the following parameters: N=16, k=5, ProbFalsePositive=0.05, S_k=16bytes.*

Considering the effect of the parameter changes in ESC (see Chapter 9.1), the ESC-search can temporarily delay the refresh of the summaries and/or reduce the number of requests until the peak is over without a large performance drop.

Last but not least, the size of the keys (S_k) also affects the amount of data sent through the network. The ESC-summary size is not affected by S_k. Nevertheless, the size of the request message grows proportionally to S_k. Distributed systems which can handle huge key sizes, such as the 64KB keys supported by Bigtable [25], will have even larger traffic reduction thanks to the decrease in the number of transmissions .

7.5 Summary and conclusions

In this chapter, we analyze the cache memory management in a distributed question answering system. We analyze several cooperative caching algorithms for distributed systems, and we obtain significant improvements for all of them over locally based policies. Even the worst cooperative algorithm tested in the chapter can achieve a reduction of over 35% of the execution time compared to a system that relies only on local caches.

We use ESC-summaries to design novel placement and search algorithms

for distributed caches. ESC-placement keeps at least one copy of the most accessed documents in some node of the network, and sends the data to the node where it is more frequently used according to the recent history. The placement algorithm outperforms other cooperative caching proposals as Expiration Age by up to 16% for web-search-engine like query distributions. Overall, ESC-placement provides a speedup between 1.33 and 1.81, compared to locally managed caches.

ESC-search utilizes the information stored in the summaries to compute which nodes have a higher probability of containing a document. In our experiments, the average size of the subset of nodes queried by ESC-search was reduced by 90% with respect to the broadcast protocol. Although the system queries a reduced number of nodes, it can still find the documents with a probability higher than 97.5%.

Chapter 8

Cache-Disk-CPU-Aware Load Balancing for Question Answering

Load balancing algorithms implemented in distributed systems assign the tasks to each node in such a way that all the resources available are used in an even way. In order to achieve the best performance, it is necessary to provide the load balancing algorithm with an estimation of the resources needed for each task that is as close as possible to the real needs of the task. 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 [54], or (b) migrating a task preemptively from one computer to another [89]. In both cases, the impact on the system is important: it produces processing overhead and additional network traffic.

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 [111], the disk I/O [27], or both [105], but none of them is aware of the cache contents in the system, the CPU load, and the I/O load. In this chapter, we propose two load balancing algorithms that take into account the cache contents in each node in order to improve the overall system performance. These algorithms rely on the Evolutive Summary Counters data structure, and hence they are a complement to the cooperative caching algorithms described in Chapter 7. Then, in Chapter 9, we study the interaction between the ESC-placement and the best load balancing algorithm from the ones described in this chapter.

We start the chapter with a description of a model to analyze the impact of caching on the load balancing. Then, we propose our new cache-aware algorithm: Probability Cost and Affinity. Finally, we report the experiments that measure the performance of our cache-aware load balancing proposals.

As already mentionened, the evaluation in this chapter does not implement the cooperative cache presented in the previous chapter but a neutral one (ICP). In the following chapter, we evaluate the system with both cooperative caching and load balancing techniques activated simultaneously.

8.1 Load balancing and caching

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 built a simple model to quantify the potential imbalance introduced by cached data. We model a system that does not take into account the caches in the global system. Then, we show the load balancing improvements, if the underlying system has a cache available. Therefore, this model illustrates that load balancing performs better when it is cache-aware.

We model a system with one type of resource (for example CPU), but it can be extended to other environments where more than one resource is necessary. We model a cluster of N nodes, with c cores per node, and where $N * j$ jobs are going to be scheduled and executed. Each job takes 1 unit of time, and cannot be further parallelized. If no caches are present, a perfect balance is obtained if j jobs are assigned to each node. However, if caches are available some tasks will not take 1 unit of time to be completed, but much less, because they will find the data cached. For simplicity, we will assume that a job that finds the data cached spends a negligible time for computation: `if` hit(j_i) `then` Load(j_i)=0; `else` Load(j_i)=1. We call M the probability that a job does not find the information in cache.

We can consider a system imbalanced when the queue of tasks in a node differs from the queue of the rest of nodes. However, we use a definition that is less restrictive: we consider that the node N_i is imbalanced when after the job scheduling, the load balancing algorithm assigns to N_i less than c jobs whose data is not in the cache (Load(N_i) $< c$), and there is at least a node in the cluster with more than c jobs whose data is not cached ($\exists i \in N$ Load(N_i) $> c$). The previous definition is restated in practice as: node N_i spends idle cycles because there are not enough tasks without the data cached to fulfill the cores in the node, whereas other nodes in the cluster are overloaded. We measure the imbalance in the cluster with a random variable X_i that counts the number of jobs whose data is not cached in the node N_i. X_i is a binomial variable following a distribution, whose probability is the probability of miss: $X_i \sim B(j, M)$. From our previous definitions, we count

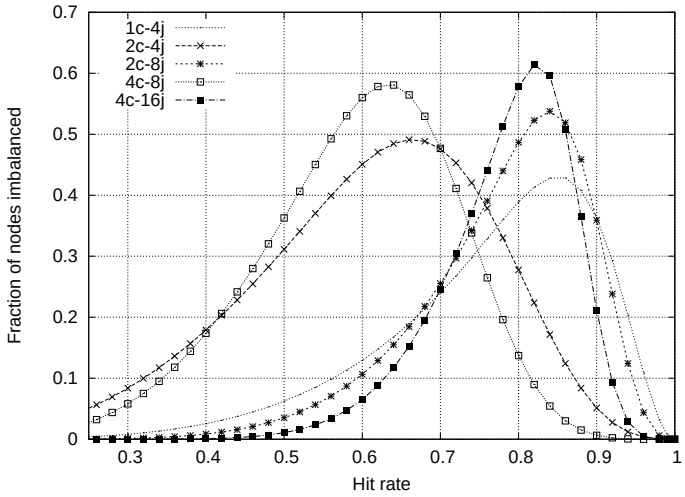


Figure 8.1: *Fraction of nodes imbalanced from a cluster of 16 nodes vs the hit rate of the jobs executed in the cluster for different node configurations. Each curve depicts a different node configuration where C is the number of cores per node, and J is the number of jobs assigned to each node.*

the number of imbalanced nodes as the sum of individual nodes imbalanced:

$$imbalanced\ nodes = \sum_{i=1}^{N} \left[prob(X_i < c) \cdot \left(1 - \left\{ 1 - prob(X_i > c) \right\}^{N-1} \right) \right]. \tag{8.1}$$

We plot in Figure 8.1 the parametrization of a model that estimates the imbalance in a cluster of 16 nodes. We measure the imbalance as the average number of nodes, with free resources, which could have been used to lighten the overloaded nodes. The model shows that the imbalance is mainly influenced by three factors: the probability of a cache hit (which is plotted in the horizontal axis), the number of cores per node, and the number of tasks assigned per node. It is important to see that for certain scenarios there is a large majority of imbalanced nodes. It is easy to see that the more jobs are available for scheduling, the harder it is to imbalance the system, because there is always exceeding work –compare the plots for 2 and 4 cores with a fixed number of jobs and twice this number. Although the saturation of the cluster with many jobs can avoid the underutilization of the processors, in a real computer two problems arise: (a) the average execution of each task increases because the processor is shared by more tasks, and (b) the system may run out of physical memory and require the use of swap memory, which reduces drastically the performance of any application.

All the curves in the plot have two sections: one ascending and other descending. The left increase occurs because, as hits grow, there are more

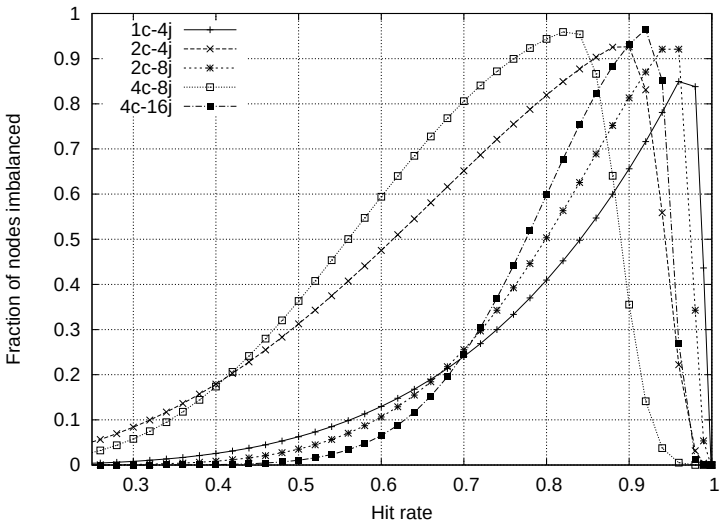


Figure 8.2: *Fraction of nodes imbalanced from a cluster of 1024 nodes.*

tasks with their data cached, and whose cost is consequently missestimated. If many tasks assigned to a node are overestimated, then this node will end its execution much faster than others, and its resources will be idle. Despite the fact that the right part of the graph is decreasing, this is not a good situation either: the decrease happens because the system is underloaded and we have a distributed system from where we are not obtaining its peak performance. The right part of the curves appears because most of the data requested is already available in cache, and it is not probable that many tasks whose data is not cached meet in the same node. Therefore, the average number of imbalanced nodes descends for very large probability hit rates.

If the number of tasks per core is kept constant, then we see that the more cores available the sharper the imbalance. This is a problematic behavior because the technology trend of processor design for the foreseeable future is to continue with the increase of the number of cores. Another troublesome effect is that with more cores available the imbalance conditions appear at lower hit rates. A single core system with a local cache may not reach hit rates large enough to be severely imbalanced. For example, the fraction of imbalanced nodes is below 0.1 for a distributed system of 16 computers (Figure 8.1) with a single core and 50% hit rate. However, the imbalance for current systems may affect a large fraction of the computers because each node typically has several cores and the cooperative cache can reach large hit rates since it is able to aggregate the memory available from multiple nodes.

The addition of more nodes to the system or cores to a node does not solve the problem, because the model becomes more skewed and the fraction of imbalanced nodes grows. In Figure 8.2, we plot the model for a large

distributed system composed of 1024 computing nodes. Here, the fraction of nodes imbalanced is larger than for a smaller network. Furthermore, the plot has negative skew because the maximum is towards very high hit rates. The negative skew, which is a near perfect hit rate, indicates that the more effort we put into improving the hit rate the more imbalanced nodes appear in the system.

All in all, according to the predictions of the model, cache awareness in load balancing algorithms was not very relevant for old application platforms (with one core, few tasks per node and local caches with low hit rates) because these parameters contribute to producing low imbalances originated by the caches. But, the increase of cores, interconnected nodes, and the use cooperative caches, which increase the hit rates, is a breeding ground for producing imbalances in the task distribution for large clusters of computers.

8.2 Load balancing algorithms with Evolutive Summary Counters

As we have seen in the previous section, caches have an impact on the computation cost of any query because they speed up the computation of queries. In a system with a cooperative cache, the queries can retrieve data not only from the local cache in the node, but can also 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. Otherwise, the load balancer algorithm has incomplete information and will lead to imbalanced assignations of tasks. Hence, we propose several algorithms that 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, which as we have seen in the previous chapter with ESC-search, is useful to predict the cache contents of a node. 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 thesis, we propose two algorithms: Probability Cost (PC) and Affinity (AF). Before introducing them, we review Weighted Average Load (WAL), which is the most relevant load balancing proposal for QA, and provides us a good start point to describe our proposals. The algorithms are described according to the notation introduced in Section 6.1. The load balancing algorithm selects a node, s, that will continue the execution of the

query. The selected node s belongs to the set of available computing nodes in the network N.

8.2.1 Weighted Averaged Load (WAL)

This algorithm, described in [105], 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:

$$s_{\text{WAL}} = \arg\min_{i \in N} \left(W_{(q)}^{CPU} \cdot Load_{(i)}^{CPU} + W_{(q)}^{I/O} \cdot Load_{(i)}^{I/O} + \zeta_{(i)} \right), \qquad (8.2)$$

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 $W_{(q)}^{I/O}$ is the analogous value for the time spent doing 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, 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 from among them.

8.2.2 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 $C_{(task(q))}^{CPU}$, which we now refer to as $C'^{CPU}_{(task(q),i)}$. $C'^{CPU}_{(task(q),i)}$ defines the CPU cost of executing the task q, or in other words, the additional load added to the system if q is executed in 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 ($C_{HIT(task(q))}^{CPU}$, $C_{RHIT(task(q))}^{CPU}$, $C_{MISS(task(q))}^{CPU}$, respectively):

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

The probability of finding 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 on the assumption that the cache contents in each node are mutually independent: $P_{(d \notin N)} = \prod_{i \in N} (1 - P_{(d \in i)})$. 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 whether the query will find the documents locally or 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 query q in node 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. $C'^{(y,i)}_{x}$ 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.

8.2.3 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, we calculate ϑ with a popular relevance formula used in information retrieval, the $tf \cdot idf$ [96]. Information retrieval focuses on finding the set of the most relevant documents 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 are best fitted 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 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:

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

In information retrieval $tf \cdot idf$, is used because some terms are more relevant for the query than others. $tf \cdot idf$ is a compromise between two heuristics: it gives a bonus to the terms that appear many times in one document and penalizes the terms that appear very often in the collection. Nevertheless, in our case the intuition to use $tf \cdot idf$ is slightly different.

On the one hand, in our proposal, we compute tf as the number of times that a document has been recently read, according to the ESC-summary stats. The higher the tf, the larger the probability of the document being in the cache because it is more likely to have been recently accessed. Therefore, tf is promoting a locality policy: the queries that request a document which is available in a subset of nodes will be sent preferable to these nodes.

If we had only used tf, the server selection process would be dominated by the most popular documents because they have the largest frequency. And consequently, the most popular documents would be accessed locally and the infrequent documents would be accessed remotely. However, this is against a good cache policy because it would replicate infrequent documents and would put additional contention on the nodes that store the popular documents. The use of *idf* is aimed at solving this problem by giving an estimate of whether a document is rare or popular in the context of the network and reduce the cache contention. It reduces the load of nodes that store the documents with multiple copies in order to substitute these replicas with other contents.

In conclusion, the selection of $tf \cdot idf$ as $\vartheta_{(i,q)}$, ensures the previously

described requirements for a good affinity function: (a) local cache accesses are prefered rather than remote because of tf, and (b) documents which are not widespread in the cooperative cache have more weight in the calculation of the affinity score because of idf, and so the rare documents tend to have fewer replicas. This behavior reduces the global load of the system because the data is available locally, thereby obtaining a more efficient use of the available resources.

8.3 Comparison example

We illustrate the differences between the three main load balancing algorithms WAL, and the proposed PC and AF with an example of a distributed QA system composed of four nodes. Figure 8.3 shows the system state at a given time: each node has information about the load and the ESC-summary of the rest of nodes. We color in green the queries whose documents are cached, and in red the queries whose documents are not available in cache. The WAL algorithm estimates the load of a node indepedently of the cache state. For example, in Figure 8.3, the load in node 4 is 40 time units because it has four queries queued (each one with cost 10 time units). On the other hand, cache-aware algorithms estimate the load of a node by considering the state of the cache, and by reducing the computational cost if necessary. For example, PC and AF estimate the load in node 4 as 31 time units, because it has three non cached queries queued, which account for 30 time units, plus a query with its contents available in cache, which has a cost of one time unit.

In this example, node 4 is overloaded and is going to forward the execution of 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 a 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 "least" loaded node according to its information. However, this is not a good choice because node 3 has to compute query Q_3, which 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 in each node: both PC and AF send a lower load for nodes 1 and 2 because

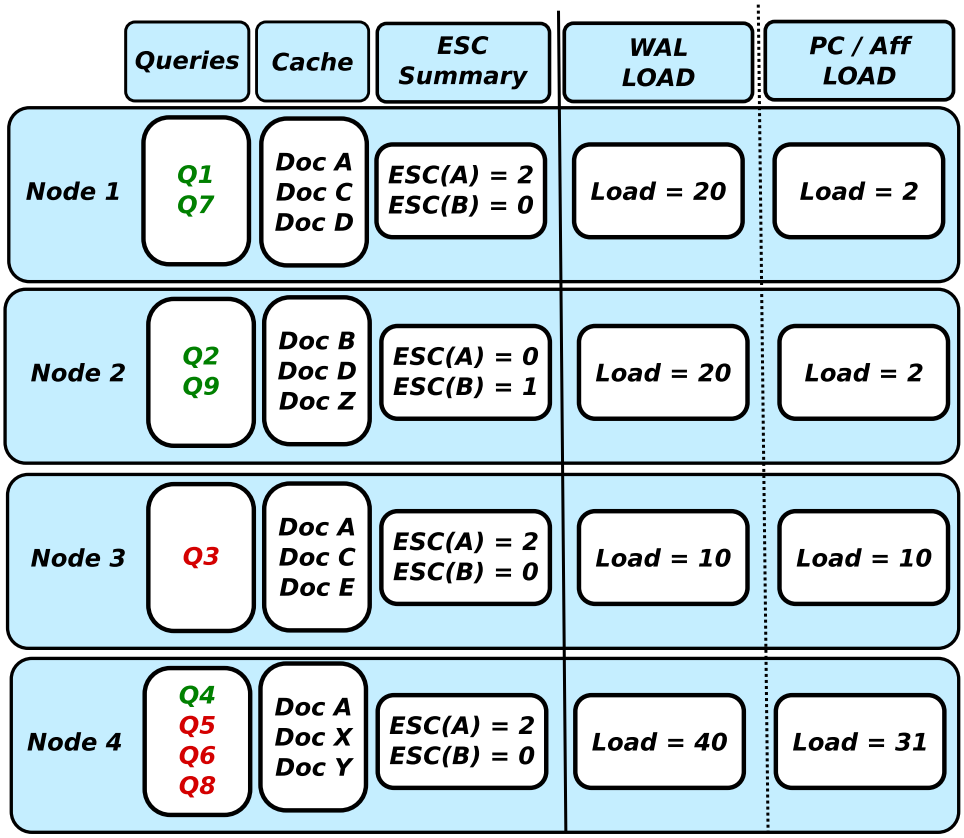


Figure 8.3: *Document A is cached in nodes 1, 3 and 4. Document B is cached in node 2. Node 4 is overloaded and is forwarding Q_8, which will access A and B, to a different node in the network. $C^{CPU}_{(x)} = 10$ for the non cache-aware example, and $C^{CPU}_{HIT(x,i)} = 1$ and $C^{CPU}_{MISS(x,i)} = 10$ for the cache-aware.*

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 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 will request document A more often than document B. For example, suppose the next query that arrives, Q_{11}, only

reads document A. If 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. AF ensures that Q_{11} finds all the information cached locally independently of which node is underloaded. However, PC may need remote accesses if Q_{11} is assigned to node 2 and document A is not already there.

8.4 Experimental Results

As already mentioned in Section 6.3, in this experimental section, we limit our analysis to the evaluation of the cache-aware load balancing algorithms presented in this chapter. Therefore, we implement an algorithm similar to ICP [110] in our cooperative cache manager: a node broadcasts a request to the rest of nodes in the network in order to retrieve the data associated to a document identifier; and if any node has the contents available in its cache, it sends the requested data to the querying node. 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.

We use the experimental setup described in Section 6.3 with query sets compound of 3000 queries using different Zipf distributions and cache sizes. In addition to WAL, PC and AF, we also compare the algorithms against two more baselines:

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 [24], which 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.

8.4.1 Comparison of load balancing algorithms

Distribution Zipf$_{\alpha=0.59}$: In Figures 8.4-8.6, we plot the outcome of several variables that we measured for an execution of a query set following a Zipf$_{\alpha=0.59}$ distribution. The horizontal axis in all the plots is the maximum number of documents that can be stored in the cache. The vertical axis measures the average throughput (Figure 8.4, the hit rate (Figure 8.5) and the probability of forwarding (8.6).

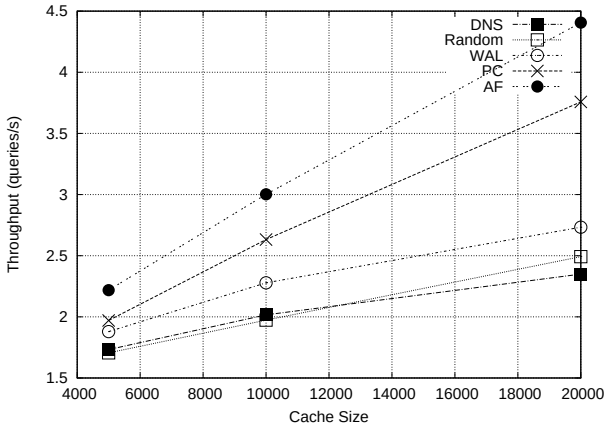


Figure 8.4: *Throughput for Zipf*$_{\alpha=0.59}$.

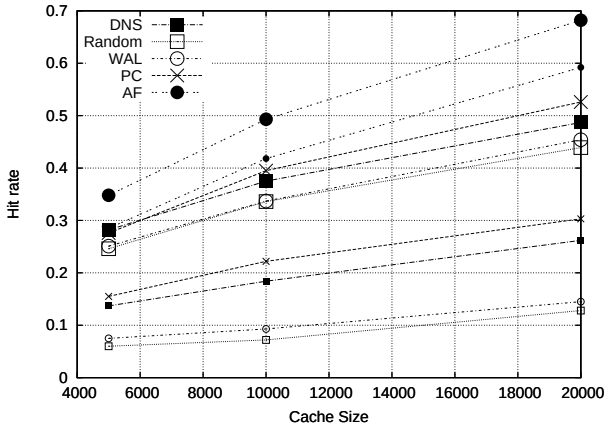


Figure 8.5: *Hit rate for Zipf*$_{\alpha=0.59}$.

The first observation is that the simpler policies have poor results: a DNS based approach is not advisable, and neither is Random competitive with policies that are aware of the execution costs. In some cases, Random 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: they improve the throughput over WAL

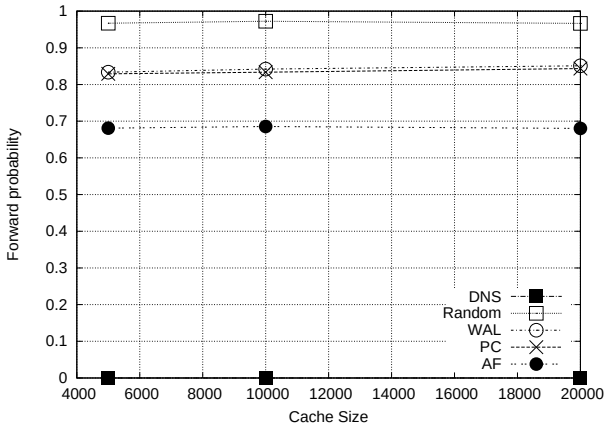


Figure 8.6: *Fraction of questions that change the node in which they execute after one step. Query distribution follows* $Zipf_{\alpha=0.59}$.

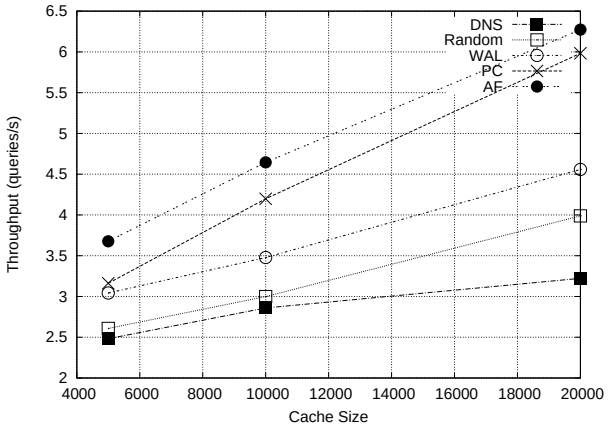


Figure 8.7: *Throughput for* $Zipf_{\alpha=1.0}$

by 61%, and DNS by 88%.

We depict the hit rate obtained by each algorithm in Figure 8.5. 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 8.4: 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 increasing the throughput of the system. Furthermore, Figure 8.5 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.

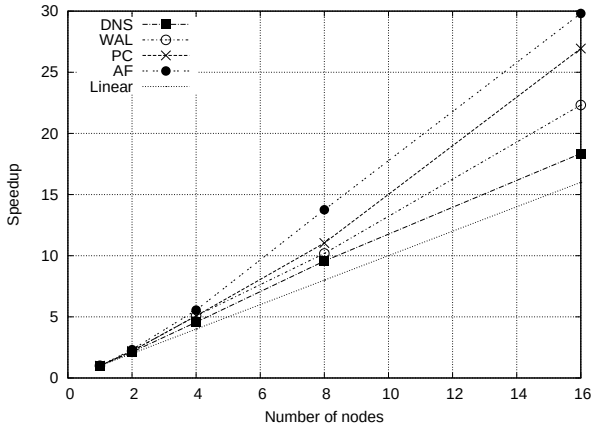


Figure 8.8: *Speedup for Zipf*$_{\alpha=1.0}$

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 the cache locality better than non cache-aware algorithms. So, the throughput of PC is better because the data is more often accessed locally.

As the cache size grows, the trend for all the load balancing algorithms 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 the results with an infinite cache.

We also record 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 8.6. 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 distributing 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 8.7 shows the results for the same experiment but with a more skewed query set. The shape of the results is similar to Figure 8.4: 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 increases the throughput 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 $Zipf_{\alpha=1.0}$ distribution, is similar to that shown in Figure 8.6.

We have also experimented with the system performance varying the number of nodes in the distributed system. In Figure 8.8, we plotted the experiments as the system speed-up. All the tested algorithms behave consistently for the different number of processors and achieve a superlinear speedup, 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 in the plot that even if we use no load balancing algorithm (DNS) we obtain a superlinear speedup because of the larger efficiency of the cooperative cache.

Although DNS reaches a very good speedup, it creates important imbalances that are stressed when the number of nodes increases. 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. This effect is also derivable from the imbalance model described in Section 8.1. 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 throughput 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.

Distribution Zipf$_{\alpha=1.40}$: Since this configuration is the most skewed distribution in our tests, the caches are even more effective than in previous experiments. We present the throughput achieved by the different algorithms in Figure 8.9. We observe that the algorithms have a similar trend to previous throughput measurements. Cache-aware load balancing algorithms get the best throughput and AF is the best algorithm overall. We note that the throughput progression towards the 20k cache is less linear than for previous experiments. This effect is produced by a progressive approach of the cache to the ideal hit rate, which is visible in Figure 8.10. Here, we see that the hit rate of all the algorithms converge to approximately 90%. Nevertheless, for this particular configuration, the throughput of the algorithms is still different because of the load balancing policy itself. WAL and PC divide the computation more evenly than a random or DNS, and thus, get a better throughput. And, AF is better than any of the other alternatives because

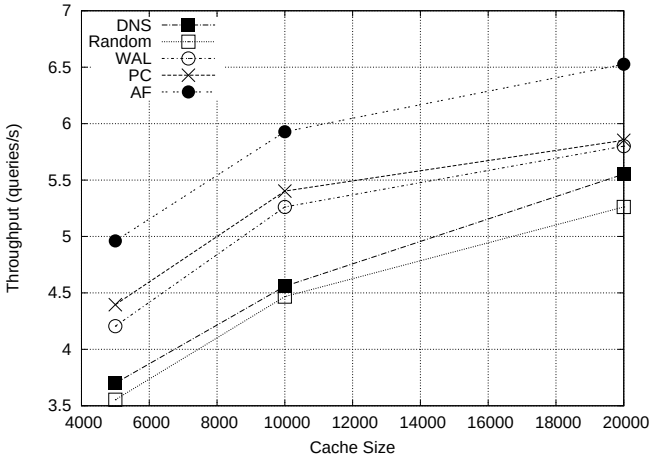


Figure 8.9: *Throughput for Zipf*$_{\alpha=1.40}$

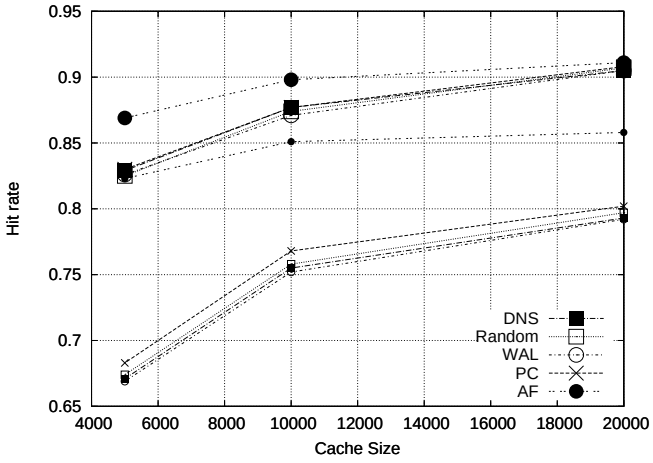


Figure 8.10: *Hit rate for Zipf*$_{\alpha=1.40}$

of the high local hit rate, which yields to a better final throughput.

For this query distribution, 16 nodes combined with the biggest cache configuration tested, we reach the highest throughput in the experiments, which is more than 6.50 queries per second. This corresponds to a speedup of more than 100 times over the original system, without a cache system, in a single computer (whose throughput is 0.06 queries per second); and we get a speedup over 25 if we consider the system with cache in a single computer as baseline (0.26 queries per second).

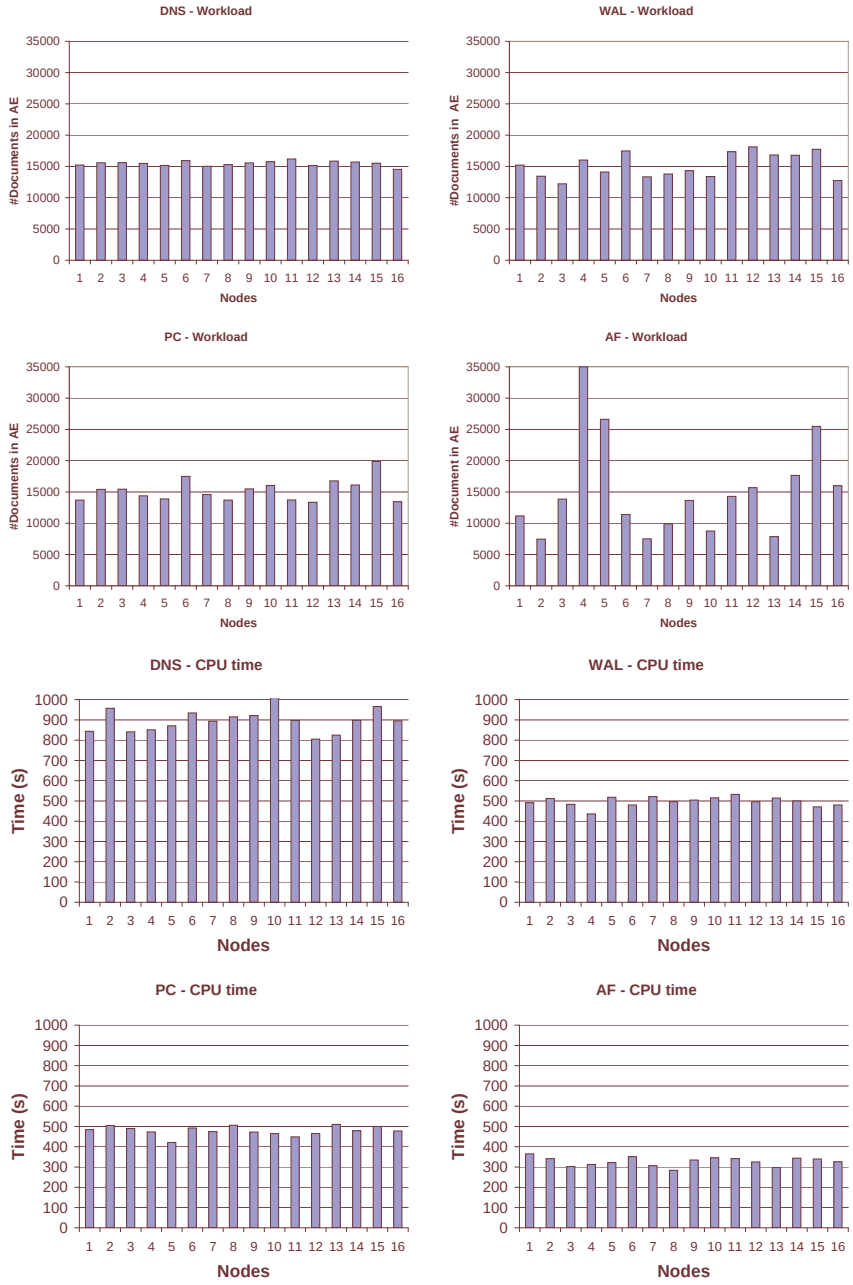


Figure 8.11: *Workload and CPU time consumption for each node in the system. Query set follows a Zipf$_{\alpha=1.0}$.*

8.4.2 Imbalance vs. performance:

In Figure 8.11 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 presents similar patterns. For simplicity, we do not show the random algorithm, because as seen 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 measure an average CPU usage of 0.56, 0.63 and 0.65 for WAL, PC and AF respectively).

These results confirm that cache-aware algorithms, although introducing a significant imbalance of the workload in the system, 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.

8.5 Summary and conclusions

In this chapter, we have built a model showing that the overestimation of the query cost originated by the cache hits can produce large imbalances in computationally intensive distributed systems, such as QA.

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 load. *Probability Cost* reestimates the computational cost of the queries according to the cache contents in the computing node and the state of the global cache. It shows a significant and consistent performance improvement in all our configurations –for different query sets, number of nodes, cache sizes, query distributions and QA configurations.

Nevertheless, we find that a good query assignation for a distributed question answering system is not sufficient for a balanced workload. The cache hit rate in the system is very important, and an imbalanced workload achieves better performance if it takes advantege of a better hit rate. We propose the *Affinity* algorithm, which includes a parameter that considers the affinity between the query and the cached contents. Although Affinity drives to imbalanced workloads, the computing time in each node is balanced due to the better hit rates, which turn into a better throughput than that of the PC algorithm.

Chapter 9
Distributed System Analysis

In this chapter, we statistically model the performance of the distributed question answering system. The statistical analysis carried out in this chapter applies ANOVA models, which were described previously in Chapter 5. We analyze two different problems arising in the previous chapters from a global perspective: (i) How to configure properly the ESC to obtain the maximum performance in our distributed system? (ii) Given that ESC-placement and cache-aware load balancing improve the hit rate and the system performance, how important is the interaction between both techniques? And hence, is either of the two alternatives preferable over the other? The results obtained in this chapter yield the conclusions reported in the final chapter of the thesis.

9.1 Analysis of the Evolutive Summary Counters

The Evolutive Summary Counter is the data structure that provides the recent history information to the system. The cache management algorithms are conditioned by the information in the ESC. Nevertheless, the ESC is a probabilistic data structure that can be configured with several parameters that may influence the system performance. Here, we study the influence of each parameter on the system configuration when both load balancing and cooperative caching techniques are enabled in a system.

9.1.1 Factors

Our aim is to understand the configuration of the ESC data structure, described in Chapter 6. We select several variables that in our experience are relevant to the system outcome, and we design a factorial experiment for all of them. The four factors under study are the following:

- *Update*: This parameter accounts for how often is the ESC-summary recalculated. We test five levels that indicate the update time in seconds: $Update = \{ 5, 10, 30, 60, 120 \}$.

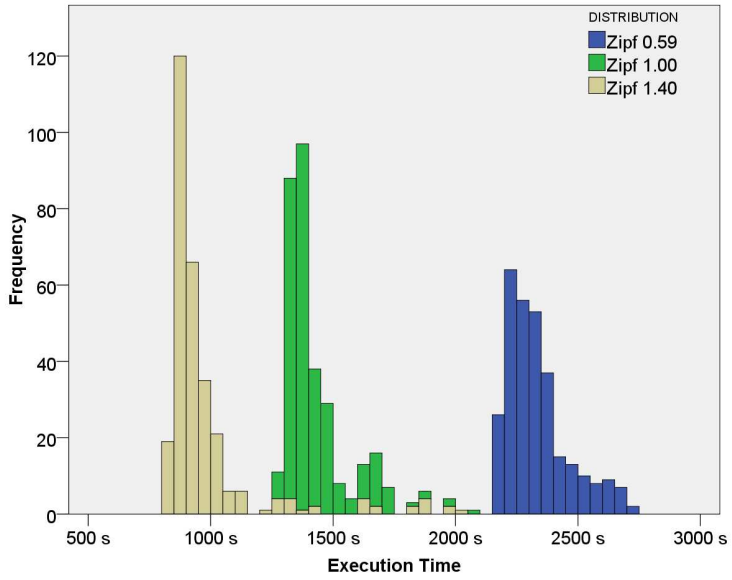


Figure 9.1: *Histogram of the system execution time, detailed by distribution*

- *ListLength*: This configures the number of CBF that are kept in the ESC. A value of 10 means that the system keeps the 10 most recent CBFs in the ESC list. We test the following levels: $ListLength = \{$ 1, 5, 10, 25, 50 $\}$.

- $FalsePositive$: This parameter measures the expected false positive rate of the CBFs in the ESC. A larger value indicates that the CBF size is reduced, which means that a larger number of collisions in the hashes of the CBF occur. We test the following probabilities: $FalsePositive = \{$ 0.30, 0.10, 0.01, 0.001 $\}$.

- *Distribution*: This determines the distribution of the query input set. As already stated, query logs from search engines follow power law distributions and thus we consider three different zipf distributions with varying skewness. We generate three query sets from the questions of TREC-QA, following Zipf distributions with different parameter: $Distribution = \{$ $\text{Zipf}_{\alpha=0.59}$, $\text{Zipf}_{\alpha=1.0}$, $\text{Zipf}_{\alpha=1.4}\}$.

This yields a factorial design of 300 different configurations. For each of these configurations, we perform 3 observations, which adds up to a total of 900 observations, which approximately accounts for 17 computing days in a fully dedicated 16 node cluster).

9.1.2 Exploratory data analysis

We start our data exploration with a visual analysis of the observations in the sample. We observe that the query distribution has a very strong influence on the execution time of the system. We illustrate this effect in the histogram of execution time detailed by query distribution (Figure 9.1). The observations for each distribution depict three different groups, with most observations clustered, and hence the histograms exhibit a large kurtosis. This is a consequence of a differentiated mean hit rate for each distribution (0.90 for $\text{Zipf}_{\alpha=1.4}$, 0.73 for $\text{Zipf}_{\alpha=1.0}$, for 0.51 for $\text{Zipf}_{\alpha=0.59}$) and the small standard deviation observed in each of these observation sets (less than 0.04 for each group). We also see that there are no observations below a certain execution time for each distribution. This barrier indicates the best possible result achieved with our techniques. Nevertheless, we note that most of the configurations are very close to the minimum time observed. This indicates that the ESC policies are robust and not difficult to configure.

In Figure 9.1, we observe that all histograms are assymetric, with more observations on the left side of the histogram. The configurations on the right side, which belong to the tail, lead to poor performance because they take a longer execution time than the optimal. Furthermore, these bad configurations exist for all the distributions under test, but we notice that the more skewed the distribution the longer tail towards long execution times. In other words, the parametrization of the ESC is more influential for the most skewed distributions. We also check if these large execution times correspond to outliers. However, we find that for all the configurations that exhibit bad system throughput, all the observations show this bad performance. Thus, these observations cannot be considered outliers, but bad ESC configurations.

According to Figure 9.1 and the large differences between the three distributions, we consider it convenient to perform separate models for each distribution in order to obtain a very exact model of the system behavior. We report these separate models in Appendix A of this thesis. Nevertheless, although separate models are more precise in the parametrization details of ESC, they require an advanced knowledge of the query distribution. For that reason, we also build a unified model with all the factors included (the number of DCFs, the update time, the false hit probability and the query distribution). This more general model, which we present in this section, gives a more general overview, and leads to conclusions similar to those obtained from the separated models.

9.1.3 Model description and validation

We report the levels under study for each factor in Table 9.1. We built our model incrementally. First, we considered the model with only the main

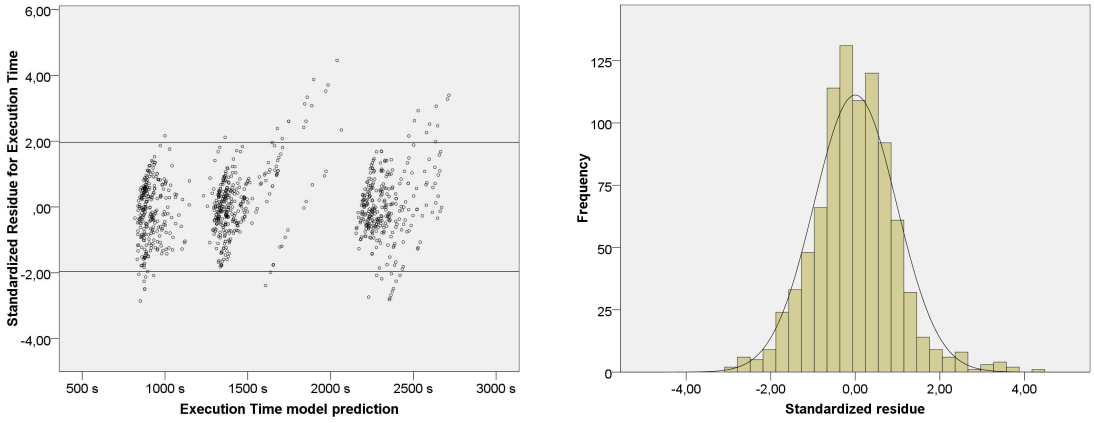
Factor	Level	Description
α_i Update	i = 1	ESC-summaries updated each 5 s.
	i = 2	ESC-summaries updated each 10 s.
	i = 3	ESC-summaries updated each 30 s.
	i = 4	ESC-summaries updated each 60 s.
	i = 5	ESC-summaries updated each 120 s.
β_j ListLength	j = 1	The ESC has 1 DCF
	j = 2	The ESC has 5 DCFs
	j = 3	The ESC has 10 DCFs
	j = 4	The ESC has 25 DCFs
	j = 5	The ESC has 50 DCFs
γ_k FalsePositive	k = 1	The false positive probability of a DCF is 0.30
	k = 2	The false positive probability of a DCF is 0.10
	k = 3	The false positive probability of a DCF is 0.01
	k = 4	The false positive probability of a DCF is 0.001
π_k Distribution	l = 1	Query distribution is $\text{Zipf}_{\alpha=0.59}$
	l = 2	Query distribution is $\text{Zipf}_{\alpha=1.0}$
	l = 3	Query distribution is $\text{Zipf}_{\alpha=1.4}$

Table 9.1: *Description of the factors in the analysis of the ESC performance.*

effects for each factor, but the distribution of the residuals for this model was not adequate. Then, we considered the inclusion of all the first order interactions in the model, which all proved significant. The latter model exhibited a good correlation $\mathcal{R}^2 > 0.99$ and an adequate distribution of the residuals, which proved the validity of the model. Nevertheless, we tested if we could remove some of the interactions in order to have a simpler model, in accordance with the principle of parsimony, and we found that the following model with three interactions was adequate too:

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \pi_k + \gamma_l + (\beta\gamma)_{jl} + (\beta\pi)_{jk} + (\gamma\pi)_{kl} + \epsilon_{ijkl} \qquad (9.1)$$

This model yields a very good accuracy, because it explains 98.5% of the variability in the observation set and has a reduced coefficient of variation, equal to 4.67%. In Figure 9.2(a), we observe that most of the observations (97.3%) fall in the central region. We also observe that the residuals are grouped in three blocks in the horizontal axis. This is the effect of the query distribution, which produces larger execution times for the least skewed distributions due to a more reduced hit rate. Each of the groups shows no internal pattern, and hence we can consider this distribution of residuals acceptable. In Figure 9.2(b), we plot the histogram of the normalized residuals. This histogram is very close to the standardized normal probability:



(a) Residues (b) Histogram

Figure 9.2: *Residue analysis for the ANOVA model of ESC configuration*

it is centered in zero and with a standard deviation of 0.97, very close to 1. This leads us to accept the model as valid because it verifies the ANOVA assumptions.

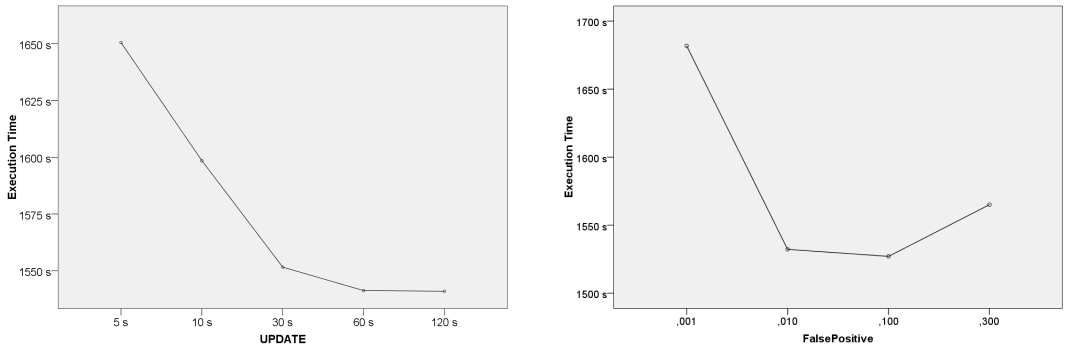
Our selected model indicates that all the factors under consideration are significant, as reported in Table 9.2. We also confirm the importance of the distribution (detected previously in the exploratory data analysis) because the F value for the distribution is larger than the one obtained for any of the other factors. The model also draws three interactions relevant for the system outcome. Two of them relate the query distribution (*Distribution*) to the configuration parameters of the ESC: *ListLength* and $FalsePositive$. Besides, we observe a significant interaction between the *ListLength* and $FalsePositive$.

The rate of updates in the ESC (*Update*) indicates the length of the ESC history. When we increase the refresh time of the ESC, we increase the number of elements that are monitored, and hence, we are picking a more representative sample of the document access distribution in that node. In Figure 9.3(a), we plot the average execution time for different refresh times, which confirms the intuition that long summaries are more representative of the activity in a node. The benefit is large up to 30 seconds, where we find that the execution time of the system stabilizes. We compute a Tukey pairwise test over the different levels of this factor, and we conclude that above 30 seconds all the levels are not statistically distinguishable. Therefore, we conclude that over 30 seconds is a good refresh rate for a system with a similar query throughput to ours.

Our next step is to establish what a good probability of hit is for each count filter in the ESC. The system tends to improve its performance when

Factor	SS	D.F.	MSS.	F	Significance	Power
Distribution	2,86E14	2	1,43E14	26425	0.000	1.000
ListLength	2.80E12	4	7.01E11	129.67	0.000	1.000
FalsePositive	3.52E12	3	1.17E12	216.67	0.000	1.000
Update	1.64E12	4	4.09E11	75.65	0.000	1.000
ListLength* FalsePositive	5.93E12	12	4.94E11	91.52	0.000	1.000
Distribution* FalsePositive	2.21E12	6	3.68E11	68.21	0.000	1.000
Distribution* ListLength	2.93E12	8	3.66E11	67.74	0.000	1.000
Error	4.65E12	860	5.41E9			
$\mathcal{R}^2 = 0.985$		$\sigma = 73.55E3$		$CV = 4.67\%$		

Table 9.2: *ANOVA model generated for the parametrization of the ESC when query distribution is* $Zipf_{\alpha=1.0}$



(a) (b)

Figure 9.3: *Execution time for different factors.*

ListLength / Distrib.	$\text{Zipf}_{\alpha=0.59}$	$\text{Zipf}_{\alpha=1.0}$	$\text{Zipf}_{\alpha=1.4}$
1	**0.639**	**0.653**	**0.718**
5	0.026	**0.600**	**0.705**
10	0.026	**0.437**	**0.756**
25	**0.071**	0.000	0.016
50	**0.693**	0.022	0.000

Table 9.3: *Pairwise Tukey test between the levels of $FalseProbability$ 0.1 and 0.01, with respect to the factors interacting in the model. Tests that did not show a significant difference are in bold.*

we reduce the probability of false positives in the DCFs because the count filters exhibit fewer collisions and the summaries are more precise (Figure 9.3(b)). Nevertheless, we observe an abrupt increase of the execution time for some configurations with very small hit probabilities. In order to reduce the hit probability, the DCF must be overdimensioned and more memory and network bandwidth is necessary, which increases the costs to maintain the ESC-summary. In our implementation, this steep increase in the cost is a consequence of our simple network coherence protocol, described previously in Chapter 6, which retransmits the DCF several times if the summaries are not properly received. We believe that a smarter broadcasting protocol than ours could send larger ESCs without this drawback. Fortunately, we observe that it is not necessary to have extremely low false positive probabilities in order to a have good performance and a false probability of 0.1 is precise enough for the configurations tested.

We run a Tukey test between the pairs of levels $FalseProbability$ and we find that all the levels are statistically different among them, except for some configurations with $FalseProbability$ 0.1 and 0.01, where the difference between γ_2 and γ_3 is not found statistically significant. Given that the model includes interactions of $FalseProbability$ with $ListLength$ and $Distribution$, we proceed to do a set of pairwise Tukey tests on the interaction, which we report in Table 9.3, fixing the two interacting factors. These tests show that for most of the configurations the two levels of $FalseProbability$ are indistinguisible. Furthermore, the levels that show significant differences correspond to configurations with very long lists, with 25 or more count filters, which correspond to configurations that may consume too many resources, and are not advisable. Therefore, we recommend setting the probability of false positive to 0.1, which yields a good performance and does not show the side effects of the interactions.

In order to complete the analysis, we show the model predictions graphically in Figures 9.4- 9.6, for any level of the three interacting factors. We observe that the average trend is that the more recent and more detailed the

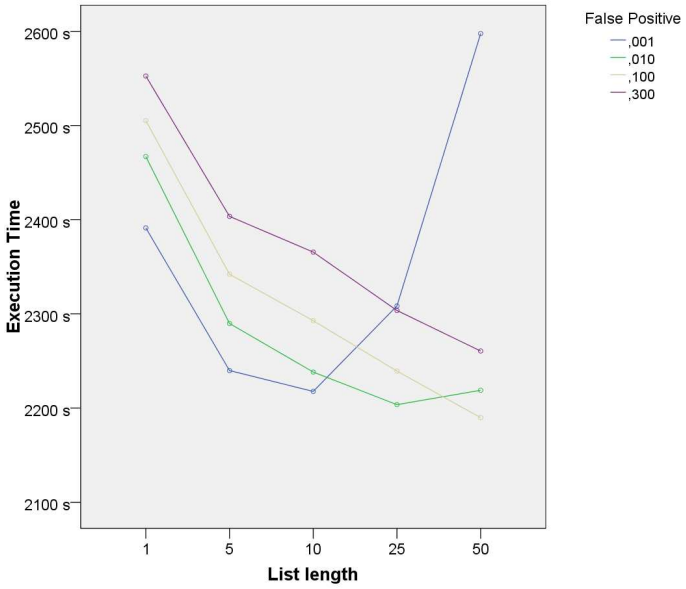


Figure 9.4: *Model predictions for* $Zipf_{\alpha=0.59}$

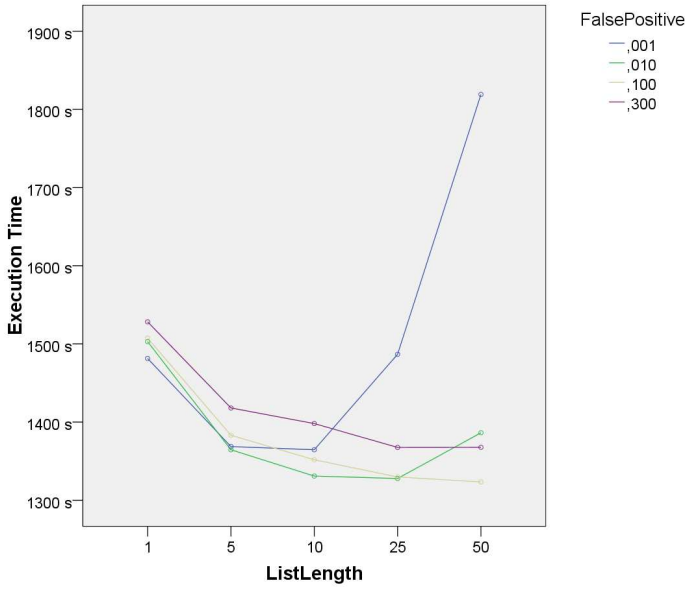


Figure 9.5: *Model predictions for* $Zipf_{\alpha=1.0}$

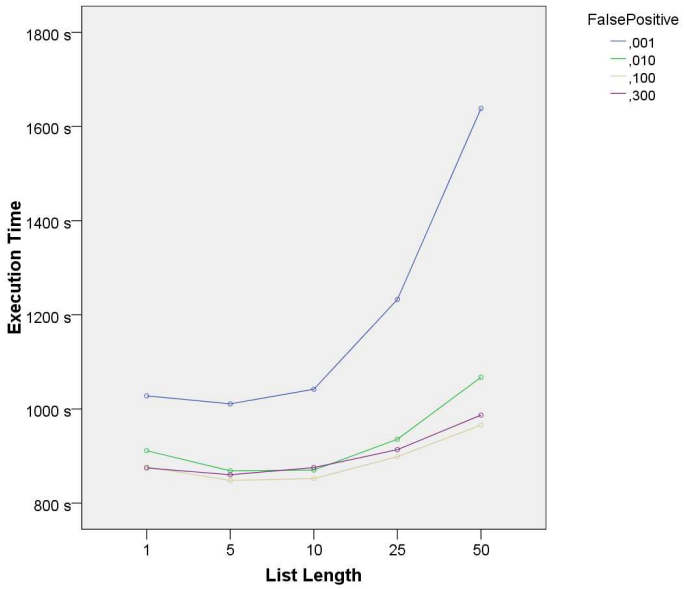
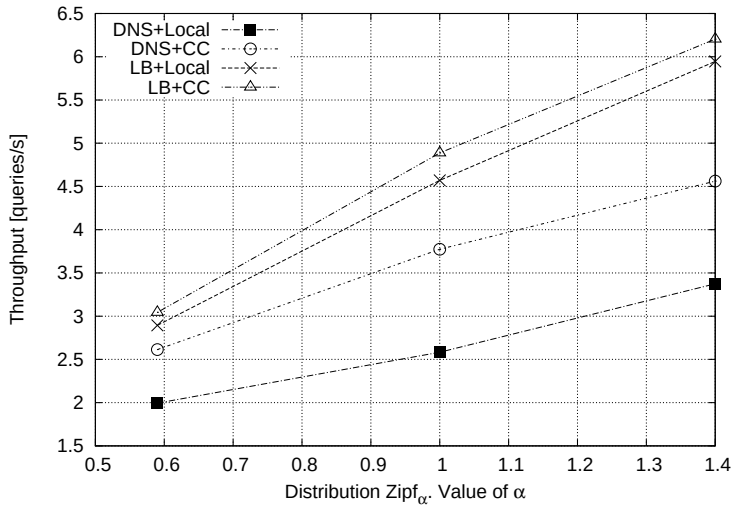


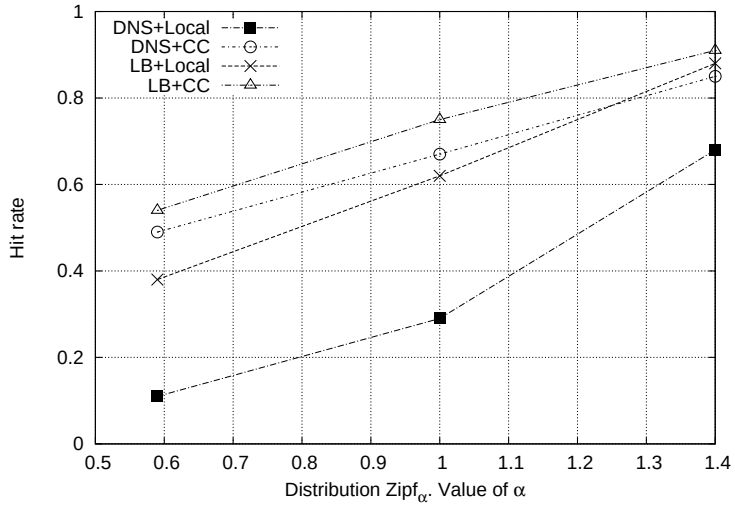
Figure 9.6: *Model predictions for Zipf*$_{\alpha=1.40}$

information included in the ESC, the better is the performance: a longer history record (*ListLength*) and reduced hit rate (*FalsePositive*) produce better throughput. We see that for all distributions the use of one count bloom filter in the ESC is insufficient and thus our proposal to build a list of count filters means a performance improvement. This recommendation is specially important for Zipf$_{\alpha=0.59}$ distributions because of the interaction $Distribution * ListLength$. It indicates that for less skewed distributions it is very important to keep longer history records because the dataset accessed is more diverse.

An apropriate number of count filters in the ESC (*ListLength*) depends on the query distribution. The best results are obtained with lists of several count filters and the number reduces with the skewness of the distribution. For a skewed distribution (Zipf$_{\alpha=1.4}$), only 5 DCFs are sufficient, whereas for less skewed distributions (Zipf$_{\alpha=0.59}$) the best results obtained compared to 25-50 DCFs. In general, the longer the list, the more information is provided, and the best performance is obtained. However, for some very long lists of DCF, the interaction $Distribution * ListLength$ affects the performance and we find that for some configurations of more than 25 DCFs the performance degrades. Therefore, we consider that 10 DCFs suffices in general, because it obtains the best results for most of the configurations and is close to the best for the remaining configurations.



(a)



(b)

Figure 9.7: *(a) Throughput of a system with different configurations of cache and load balancing. (b) Hit rate of a system with different configurations of cache and load balancing*

9.2 Load balancing and Cooperative Caching

In this experiment, we study the importance of the cooperative cache and the load balancer in the system performance. We quantify whether the improvement of each technique is statistically significant and to see if they exhibit interaction. Cooperative caching (CC) corresponds to a system where data is transferred to the node processing the query (with the ESC-placement al-

Factor	Level	Description
α_i Distribution	i = 1	Distribution Zipf$_{\alpha=0.59}$
	i = 2	Distribution Zipf$_{\alpha=1.0}$
	i = 3	Distribution Zipf$_{\alpha=1.4}$
β_j Cache policy	j = 1	Local cache
	j = 2	Cooperative Cache
γ_k Load balance	k = 1	DNS
	k = 2	Cache-aware Load Balancing

Table 9.4: *Description of the factors in Equation (9.2).*

gorithm), and the cache-aware load balancer (LB) corresponds to a system where queries are sent to the node with the cached content (with the Affinity algorithm). When the cooperative caching is disabled we keep local caching in each node activated (Local), and when we disable the load balancer we assign the queries following a round robin policy (DNS).

We perform factorial analysis to analyze the system throughput: we pick the set of factors (independent variables) to study, and for each possible configuration of the factors and distribution we obtain three observations. Thus, we test the resulting twelve configurations of a complete factorial design compound by two binary variables and a variable with three levels: the cache policy (CC or Local), the load balancing algorithm (LB or DNS) and the distribution ($\alpha = 0.59$, 1.0, 1.4). For each configuration, we repeat the experiment three times, which adds up to 36 observations.

In Figure 9.7(a), we plot the average throughput for each of the configurations with respect to the query distribution. We observe that the activation of either cooperative or load balancing improves the system significantly, over the system without any of these techniques (DNS + Local). The improvement from the addition of a cooperative cache is up to 46%, and the cache aware load balancing increases the system throughput up to 77%. However, the best system is when both techniques are combined with an increase in the throughput of 90%. We also observe that in any situation, the improvement is greater for the largest values of α.

Figure 9.7(b) shows that both techniques improve the hit rate significantly and by a similar amount. However, cooperative caching and cache-aware load balancing increase the system hit rate from two different perspectives. Thus, when we combine them, the hit rate is better than with either of them individually, as can be observed in Figure 9.7(b).

We test different models in order to fulfill the parsimony principle, and pick as our final model the one that takes into account all the three variables (the distribution, the cooperative cache, and the load balancing) plus the interaction between the cooperative cache and the load balancing. Al-

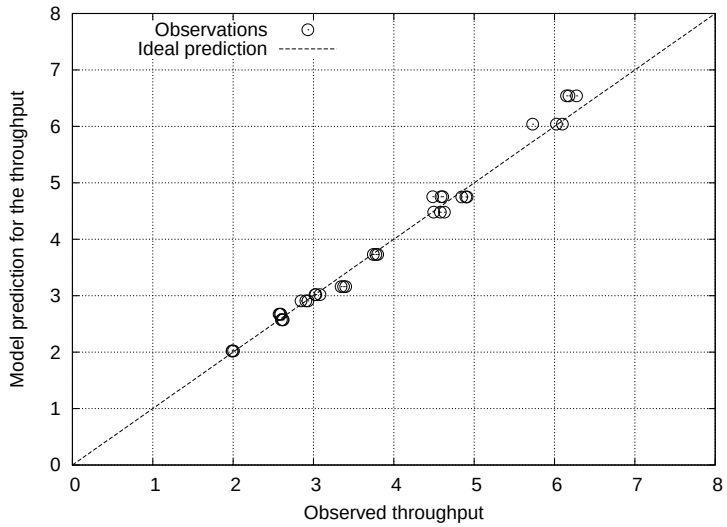


Figure 9.8: *Factorial analysis for three factors: cache-aware load balancing, cooperative caching and query distribution. Predicted throughput by the model vs. observed throughput (Equation (9.2)).*

Factor	SS	D.F.	MSS.	F	Significance	Power
Distribution	4969076	2	2484538	1034	0.000	1.000
Coop.Cache	793287	1	793287	330	0.000	1.000
Load Bal.	2440885	1	2440885	1016	0.000	1.000
CoopCache* LoadBal	490466	1	490466	204	0.000	1.000
Error	72035	30	2401			
$\mathcal{R}^2 = 0.989$		$\sigma = 49$		$CV = 3.38\%$		

Table 9.5: *ANOVA model generated for the distributed system analysis. Output variable is the total execution time of the system in ms.*

though the interactions between the distribution and the rest of the factors are significant, they have a much smaller F value and make no important contribution to the precision of the model. Thus, we discard these two interactions and the final model is the following:

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + (\beta\gamma)_{jk} + \epsilon_{ijkl}, \tag{9.2}$$

where μ is the overall mean of the observations, and α_i, β_j and γ_k are the factors under consideration, as shown in Table 9.4.

The statistical tests for the model indicate that all the included terms are statistically significant, and the response variable (i.e. the overall system performance) strongly depends on the independent variables (i.e. the cache, the load balancer and the distribution.). This confirms that both cooperative caching and cache-aware load balancing significantly improve the system

throughput. The summary of the model is reported in Table 9.5. The estimated model is very precise: $\mathcal{R}^2 > 0.98$, which means that only less than 2% of the variability is not explained by the model. As a consequence, the correspondence between predictions and observations lies very close to the identity function, which is a perfect fit, as we observe in Figure 9.8. This plot also confirms visually that the the residuals are very small compared to the system throughput, as indicated by the coefficient of variance, which is only 3.38%.

We observe in Table 9.5 that the distribution is the factor with the greatest influence, because for each distribution the cache hit rate differs, and hence the throughput of the system varies. We also observe that the cache-aware load balancer has a larger contribution than the cooperative cache. This indicates that the cache-aware load balancer is more effective than the cooperative cache for improving the throughput.

The model indicates that the best configuration activates both the cooperative caching and the load balancer and is statistically better than the other configurations. The reason for this result is the data size: a query is much smaller than a document. In other words, it is faster to transfer the queries through the network than the data requested by a query. However, cooperative caching is still valuable because it introduces a global management of the cache contents that turns into better hit rates as we observe in Figure 9.7(b).

In summary, we find that the combination of the techniques presented in previous chapters, ESC-cache and Affinity, improve the system througput. The model proves that both techniques improve the hit rate from different perspectives, and hence our two proposals are complementary.

9.3 Scheduling points analysis

We have proven that load balancing plays an important role in query execution. We now proceed to analyze if the addition of more scheduling points improves the system performance. We introduce up to four scheduling points in the system before each of the computational blocks in the system: (a) when a query is received, before QP; (b) before accessing the indexes of the collection in PR; (c) before reading the documents from the collection in PR; (d) before processing the received documents in AE. We test five configurations with an increasing number of scheduling points. The first configuration only enables the scheduling point for the most expensive computing blocks, and we sequentially add more scheduling points according to the following most expensive computing block. We test the following combinations: only (c) or (d) enabled, (c+d) enabled, (b+c+d) enabled, and finally (a+b+c+d) enabled.

The execution time for each configuration is plotted in Figure 9.9. We

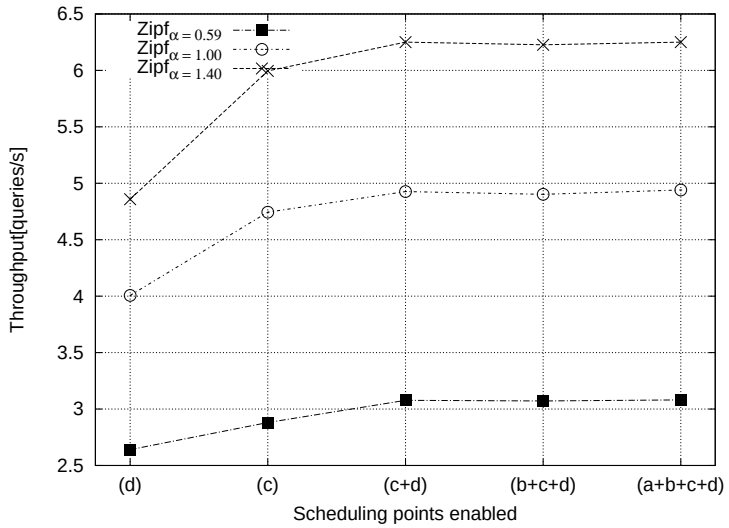


Figure 9.9: *Throughput of the system with different scheduling points enabled.*

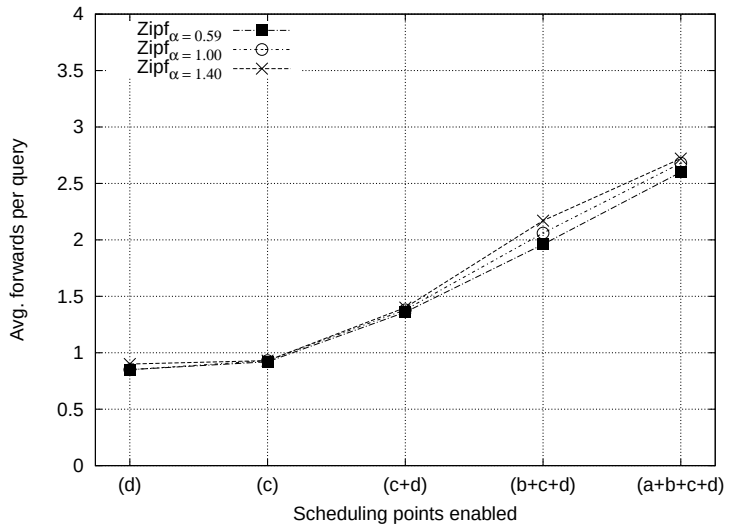


Figure 9.10: *Number of forwards per query with different scheduling points enabled.*

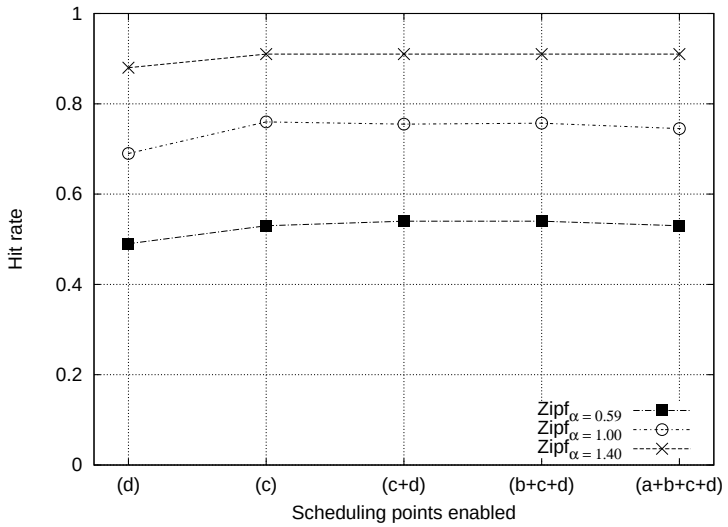


Figure 9.11: *Hit rate for different combinations of scheduling points.*

analyze our results with a General Linear Model [76], which includes only the main factors, without any interaction between the distribution and the scheduling points. The model is statistically valid for all levels of the tested variables with a high $\mathcal{R}^2 = 0.99$, which means the model predictions correlate the performance and the scheduling points significantly.

In Figure 9.9, we observe that the number of scheduling points influence the system throughput: the more scheduling points the better the performance. Nevertheless, the addition of some scheduling points (a and b) does not affect the system throughput significantly. We confirm this intuition by computing a set of contrasts among the different configurations of scheduling points [76]. The contrasts show that configurations (a+b+c+d), (b+c+d) and (c+d) present no statistical difference in the system throughput, and all of them are better than the single scheduling point configurations. We also record the number of forwards for each configuration, which are reported in Figure 9.10. The plot shows that (c+d) requires the smallest number of forwards among all the multiple scheduling point configurations, and consequently takes less network traffic.

The hit rate does not depend on all scheduling points equally (Figure 9.11). The PR scheduling point is the most relevant from a hit rate perspective. Once we enable PR as scheduling point (c), the hit rate is the same as with all the scheduling points enabled (a+b+c+d). However, considering the final performance of the system, (c+d) is the preferable option. If we compare system (c) with system (c+d), we observe that most queries are forwarded when they reach PR because they are transferred to a node with cache contents affine to the query. Almost all queries change their execution node after (c). The probability of reassigning a query is smaller

when they reach the AE scheduling point. Only a few queries (about a third of the total) change their executing node, because the query is already in a good node from the cache perspective. We note that the source of forwards in AE come from a different source than from PR: queries are forwarded due to load unbalances in the cluster. Thus, it seems plausible that cache-aware algorithms may be improved if they become flexible: first node assignments should be more cache oriented, and then the query can be transferred to an underloaded node if severe unbalance is detected.

Although the distribution modifies the throughput of the system, the model shows that the best set of scheduling points does not depend on the distribution because there is no significant interaction between the distribution and the scheduling points. According to our experiments and the statistical model generated, a load balancer for a Question Answering system should include two scheduling points in PR and AE (c+d), because this is the simplest among the best possible configurations, generates a small number of forwards, and achieves the best or close to the best performance.

9.4 Summary and conclusions

The performance of a search engine is closely related to its in-memory data management. We have compared two different approaches to improve the system performance: (a) cooperative caching, i.e, send the cached contents from a node to the node that is currently computing a query; (b) cache-aware load balancing, i.e. send a query to the node whose cached contents are related and whose computing load is small.

On the one hand, cooperative caching creates the illusion that computers share a large virtual cache pool created by the merging of the available memory in each computing node. In our tests, the speedup of this approach was up to 1.46 and always over 1.32, which is significantly better than the original system with local caches. The source of the improvement is a better hit rate, which surpasses the local policy by more than 25 percentile points.

On the other hand, cache-aware load balancing sends queries to nodes with similar cache contents. Our results show that both policies achieve a similar improvement of hit rates, but the throughput of cache-aware load balancing is 1.77 times larger due to reduced network traffic. Queries are smaller than data contents and only need a single connection, whereas the document size is larger and may require contacting many nodes. Nevertheless, each technique copes with cache management from complementary perspectives and the throughput of the combined system is up to 1.90. Our statistical model proves that although the absence of caching and load balancing penalizes the system throughput severely, both techniques do not collide when they are simultaneously enabled and can be activated simultaneously for a better performance.

All in all, our statistical model provides us with several simple guidelines to configure the ESC data structure. In general, the best throughput is obtained when we increase the history record of the ESC, because it contains more historical information and/or makes it more precise. Nevertheless, we note that if the ESC is very large we may have scalability problems due to the network transmissions. As a rule of thumb, the configuration should use a few count filters (about 10), a moderate probability of false positive (0.1 is sufficient), and a refresh time of at least 30 seconds in order to capture the data access trends in the system.

Part IV

Conclusions and Future Work

Chapter 10
Conclusions

This chapter concludes the work presented in this thesis. It summarizes the main achievements described in the previous chapters and draws some different lines for future research.

10.1 Summary and conclusions of the thesis

This thesis analyzes how to improve the performance of a distributed Question Answering (QA) system. QA is an example of advanced search engines, which supplies precise answers to queries expressed in natural language. Although current research on QA has enabled them to achieve a reasonable precision in their answers, their computational costs make them unsuitable for large scale deployment. This thesis focuses on this problem, and proposes several cache oriented enhancements, ranging from single computers to distributed systems, to improve the performance of QA systems.

The first part of this work describes the architecture of a QA system and how to implement each of its computing blocks. This implementation is a compromise between precision and complexity, yet it is qualitative enough to have an answer-finding performance comparable with the state of the art. Although some systems in QA evaluations, such as TREC, have proposed several techniques with better answer-finding precision, their basic architecture is similar to ours, and hence our performance proposals can be adapted to these more complex implementations. Additionally, our implementation is very modular and will be publicly available for research purposes. We consider that our system is a good starting point to implement new research proposals related to QA. A proof of this flexibility is its usage for different research studies related to question answering, which are not related to this thesis: testing new architectures for question answering [47], analyzing spoken document retrieval algorithms [104], or testing answer extraction procedures [40].

The second part of this work analyzes the cache requirements of a single computer QA search engine. According to the pipelined architecture of a

question answering system, we set up a multi-layer cache architecture. We study this architecture and prove that both layers are necessary to improve the system performance. We analyze the problem from two approaches, one analytical and the other statistical. Both these models agree in that the combination of two layers improves the system performance over single layer configurations. Therefore, as a rule of thumb a system administrator should avoid single layer caches and allocate some memory for each layer in the system. For a better configuration, we recommend setting the memory allocation with the aid of our analytical model for multi-layer caches.

Generalizing our results, multilayer caching is not a question answering problem specific to the problem of QA, because there are other applications with skewed workloads and several expensive computing operations. For example, the predictions of our models explain the cache duality among caching answers and posting lists in IR, in which multiple caches are also preferable [10]. Therefore, we consider that our model is a general good decision tool to optimize the cache allocation of complex applications, such as search engines.

In the third part of the thesis, we present a new data structure, called the Evolutive Summary Counters (ESC), which stores an approximate record of the documents accessed in a node. This data structure is adapted to distributed computing because the ESC can be summarized efficiently into ESC-summaries, which have a reduced memory footprint and can be sent in a fast way through the network. All the ESC summarization procedure is a background process that does not interfere with the computation of a QA system. Moreover, a look up in the ESC-summary is a very fast operation because it only requires a few hashes and does not require network communication. Thus, the deployment of ESC does not interfere with the query computation and does not increase the query response time. This is very important because the forthcoming technology is providing processors with an increasing number of cores, some of which can be used to improve the data locality of applications with the analysis of the data access trends in real time [83].

We apply the ESC to improve the cache locality of a cooperative cache, to reduce the number of nodes contacted during a search and improve the load balancing of a QA system. Nevertheless, the application of ESC is not limited to these activities. The ESC infrastructure provides statistics that can be shared and checked simultaneously by several services. We believe that our proposals related to ESC are the tip of the iceberg, because our ideas can be combined with many other high performance techniques, which exploit the data locality and adapt to the changes in the data access trends. For example, it is possible to combine our infrastructure with a new prefetching algorithm that reads data from disk according to the data access trends; or with a concurrency protocol for a database that applies pessimistic techniques (e.g. locks) for the most accessed pages, and optimistic

techniques (e.g. timestamps), which require weaker synchronization, for the less accessed pages [15]. Besides, applications that require few writes can also apply our cooperative cache proposals, by implementing a broadcast protocol to invalidate cached contents.

Our experiments show that cooperative caching achieves a significant throughput improvement over local policies for QA systems. This super-linear speedup is only possible if the caches collaborate to take placement decisions based on the global cluster benefit, like ESC-placement does, and not just on local information.

In order to have an efficient management of a cooperative caching, it is necessary to have procedures that locate the information efficiently. We present ESC-search, which, in addition to allowing a better data location in the network, also provides a probabilistic map of the data contents available in the network. This information is very valuable because the probabilistic nature of the location procedure is able to decide which nodes potentially hold a document, without additional connections. In comparison to other approaches such as DHTs, which require a cascade of communications, ESC-search contacts all nodes in parallel, which is important in high performance applications where a sequence of connections introduce latency in the search and degrades the system performance. Furthermore, ESC-search allows for the connection or disconnection of servers without any extra cost.

We also find that the assignation of a balanced workload to all the nodes in a QA system does not guarantee a good performance. In applications where caches substantially reduce the computational time, it is important to consider the cached contents available. Our load balancing algorithm *Affinity* does not only take into account the system load but also the similarity between the query and the cached contents. This policy sends the queries to the nodes that are not overloaded and have similar cache contents, which leads to a better hit rate and system throughput.

All in all, we propose two types of data distribution optimizations for QA: (a) sending the cached contents to the computing nodes that request them (ESC-placement), and (b) sending the queries to the node that has the most adequate cache contents available (cache-aware load balancing). We model the two approaches and compare them quantitatively. Our model shows that they obtain similar hit rate improvements, leading to a significantly better performance. However, we note that the reassignment of queries requires less computing resources than forwarding cached contents, and hence the cache-aware load balancing, on its own, is preferable. We also demonstrate that the two techniques are not two excluding alternatives but complementary optimizations: when both are activated, the hit rate and the performance increase over the use of each of them separately. This prompts us to recommend implementating both data placement algorithms and cache-aware query distribution in question answering systems.

From a quantitative perspective, we believe that the techniques intro-

duced in this thesis constitute a step forward in the adoption of question answering systems in real world applications. Our final system, with 16 computers, achieves an average throughput of more than 6 queries per second, which corresponds to a throughput of more than half a million queries per day. This constitutes a significant improvement over our initial system, a single computer QA system without caching, whose throughput is just 0.05 queries per second (and fewer than five thousand queries per day). Furthermore, our final system also achieves an almost two-fold improvement over a distributed QA system, with 16 computers, which does not implement our cooperative cache policies and achieves 3.37 queries per second. Our performance gain is superlinear, which proves that cache-aware techniques are a powerful tool that scales with the available computing resources.

Although the algorithms presented in this thesis are designed for question answering systems, we believe that our proposals can be applied generally to different scenarios such as other search engines. Documents from current digital document collections contain more information than the bag of words model provides: words build sentences, which in turn are ordered logically in paragraphs, and come with images, audio and multimedia content that, as a whole, describe the information provided by the document. Then, in order to get human-like search precision, search engines need to analyze the documents more and more to achieve an interpretation of the data closer to that done by a human, which means they are becoming more computationally expensive. In these environments, the impact of a good cache hierarchy management is at least as relevant as for QA. We take, for instance, advanced multimedia search, which requires deep analysis of multimedia content (e.g., music, images, movies), instead of just a simple match of the query keywords to the caption of the multimedia object. In this case, an advanced media search architecture will include not only a tag based retrieval but image and sound analysis to obtain a high quality answer. This architecture, which associates analyzed data to multimedia objects, ressembles the one presented in this thesis for QA. Therefore, we believe our proposals are not limited to the actual case of question answering, but can also applied to other future search engines or resource intensive applications.

We believe that the work presented helps the development of large scale question answering systems as well as motivates the research into performance for future search engines. This will enable us to retrieve more precise information from the giant digital data repositories available in our day to day life.

10.2 Future work

We conclude by describing some research trends which we believe can be interesting to explore according to the results presented throughout the the-

sis.

Replacement algorithms for local caching: In the thesis, we explore document placement, document search and query load balancing in a question answering system. The last component in a cooperative cache is the replacement algorithm, which we implement with the LRU policy that is considered a good enough algorithm for search engines [44, 71]. However, most previous studies consider the replacement cache policy from a local perspective, and do not consider cooperative caching. We believe that an interesting direction to explore is to test replacement algorithms that exploit the data access trends available in the ESC-summaries, and thus, take replacement decisions considering the whole distributed system state.

Application to other search engines: Even though we find many types of search engines that are specialized for certain searches (e.g. image retrieval, multimedia retrieval, search adapted to the user profile, exploration of the relations among documents...), they share many drawbacks related to their performance and scalability. Most of these search engines have to perform expensive computing operations to filter out the most relevant results for a query. This problem is similar to that presented for QA, where we read many documents from disk and analyze them with natural language tools. We consider that a promising research trend is the generalization of our techniques to such systems in order to improve their throughput.

Evolutive Summary Counters scalability: The architecture described in this thesis assumes that the ESC-summaries can be distributed to all the nodes in the network. However, this might become a bottleneck in configurations with limited network bandwidth (for example in WAN environments) or for huge clusters running thousands of computers. For this configurations, we believe that an alternative is an organization based on a hierarchy of multiple levels. In the first level, all the available nodes are partitioned in several groups that work like the architecture described in this thesis. Each of these groups works at this level independently of the others, and a node only exchanges ESC-summaries among the members of its group. Then, for each group a representative of this group is selected, which exchange ESC-summaries that represent all the group. The representatives are in charge of interacting between groups. For example, one node from one group can perform any placement, search or load balance operation through the representative of its group, which will act like a tunnel to the destination group. For even larger configurations, this architecture can be scaled to more levels or to more group representatives, thereby setting up a whole world of research.

Other ESC based algorithms: Many tasks of a distributed system can be optimized if additional information about the workload trends is provided. We consider that ESC is a versatile data structure to create not only additional cache related algorithms but also other high performance techniques. We exemplify this with a concurrence control for database that adapts to

the number of accesses to each table. There are several techniques to implement the concurrence control of a database [14]. On the one hand, we find optimistic techniques, such as timestamp ordering [14], that do not lock the pages and hence compute queries faster. Note that if the optimistic protocol finds a concurrence risk at the end of the query computation, then the query must be rolled back and be reexecuted. On the other hand, in tables with very frequent updates, the pessimistic concurrence protocols (such as locking a disk page or table) perform better because, although locking penalizes the computational time, they do not recompute any step of the query. We propose, for example, the application of ESC to build a dynamic protocol that, depending on the number of accesses to a table, applies a different concurrence protocol.

Question answering implementation at large scale: Although the system implemented in this thesis is fully functional, it corresponds to a research prototype of a QA system. We simplify some of the issues that must be considered in a production environment. Many additional performance enhancements have been published inside the research community on search engines, which can be combined with the techniques presented in the thesis. For example, query parallelization improves the response time of question answering queries [105], precomputed answers in a front end server reduce the query traffic to the search engine [13]; collection partitioning reduces the effective collection size [87]; or even more advanced document analysis which has been introduced over the years in question answering or information retrieval conferences, such as TREC, in order to retrieve higher quality results. We aim to combine the ideas presented in this thesis with those already published to make large scale question answering systems a reality.

Appendix A

Evolutime Summary Counters. Detailed performance analysis

In Section 9.1, we studied how to configure the performance of the Evolutive Summary Counters data structure. In the previous analysis, we saw that the query distribution had a very important impact on the system throughput and was the dominant factor because caches achieve higher hit rates when the distribution is more skewed. In this section, we provide the more detailed models for each distribution. If the system administrator knows the exact query distribution that the system is going to receive, it is preferable to check these more detailed models, which are more precise.

The models are generated from the same set of observations that we used in Section 9.1. We separate the observation set in three groups according to the query distribution. In general, the three individual models are similar to those obtained previously for the global system. The model for $\text{Zipf}_{\alpha=0.59}$ has the same factors as the one presented previously, and its results are very similar too. In the case of $\text{Zipf}_{\alpha=1.0}$ and $\text{Zipf}_{\alpha=1.40}$, it is necessary to include in the model all the first level interactions. Despite these additions, we observe that the most relevant interaction is $ListLength{*}FalsePositive$, with an F value of 109 and 341 respectively, like in the model described in previous chapters.

The two additional interactions, $FalsePositive{*}Update$ and $ListLength{*}Update$ correspond to warning signs of configurations with slow performance. The former says that if we set a small false positive probability to the count filters (which increase the ESC-summary size because it allocates more entries per count filter) and we wish very frequent updates, then the update cost of ESC-summaries is larger than the benefit that we get. This effect is visible in Figure A.4 and A.8, where the overhead is very large for a false positive probility of 0.001 and a number of count filters of 50. For the latter interaction, $ListLength * Update$, the interpretation is similar (see

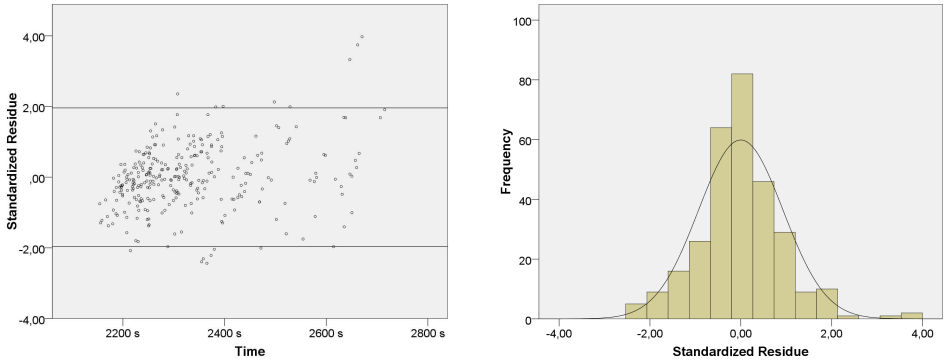
Figure A.6 and A.10). It says that if we set very long lists (which increase ESC-summary size because the counters monitor more document accesses and allocates more bits per counter) and we wish very frequent updates, then the management cost is excessive.

Note that the main factors included in the model of Section 9.1 already indicated that very frequent updates, long lists of count filters and tiny false positive probabilities were not advisable. In addition to this information, the models presented in this appendix identify which are the problematic combinations of the factors (interactions). Therefore, if we already know the query distribution that the QA system receives, the system administrator can apply the models in this appendix to achieve an optimal throughput.

Following, we include a short description of the three models. For each model, we include the ANOVA summary table that indicates the significant factors and interactions for the model, and a set of plots that depict the model predictions with respect to the interactions of each model.

Factor	SS	D.F.	MSS.	F	Significance	Power
ListLength	1.77E12	4	4.43E11	315.14	0.000	1.000
FalsePositive	3.81E11	3	1.27E11	90.31	0.000	1.000
Update	1.05E12	4	2.62E11	186.16	0.000	1.000
ListLength* FalsePositive	5.19E11	12	4.32E10	30.75	0.000	1.000
ListLength* Update	6.16E11	16	3.85E10	27.39	0.000	1.000
Error	3.65E11	260	1.41E9			
$\mathcal{R}^2 = 0.922$		$\sigma = 37.55E3$		$CV = 1.61\%$		

Table A.1: *ANOVA model generated for the parametrization of the ESC when query distribution is* $Zipf_{\alpha=0.59}$



(a) Residues (b) Histogram

Figure A.1: *Residue analysis for the ANOVA model of ESC configuration for* $Zipf_{\alpha=0.59}$.

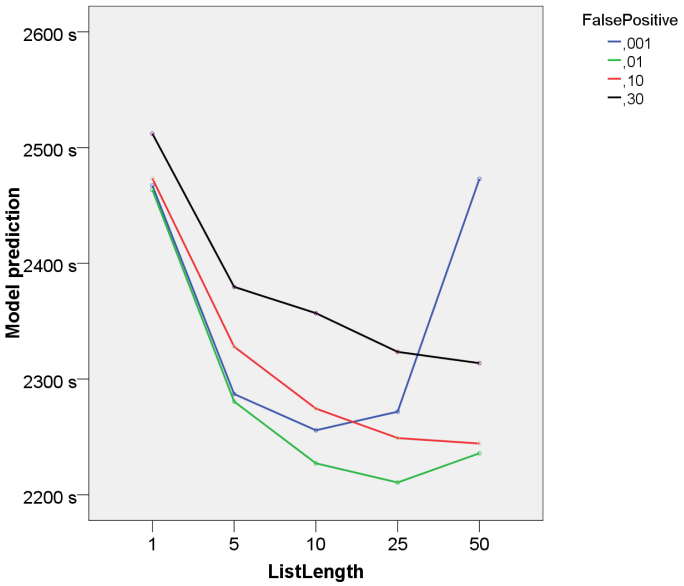
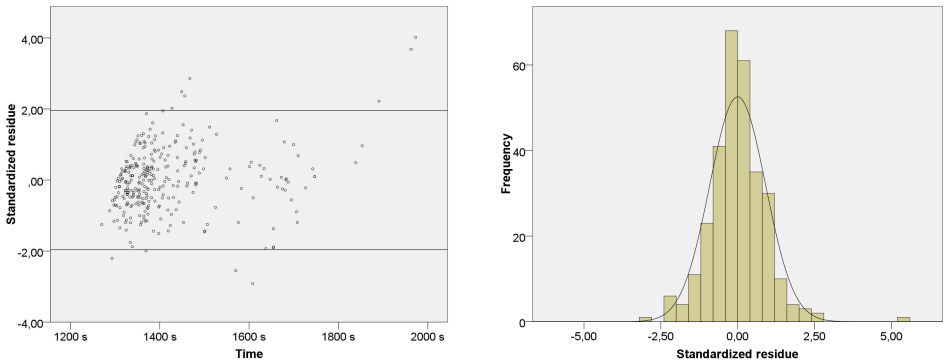


Figure A.2: *Model predictions for* $Zipf_{\alpha=0.59}$

Factor	SS	D.F.	MSS.	F	Significance	Power
ListLength	1.00E12	4	2.50E11	239.67	0.000	1.000
FalsePositive	7.63E11	3	2.54E11	243.35	0.000	1.000
Update	7.16E11	4	1.79E11	171.47	0.000	1.000
ListLength* FalsePositive	1.37E12	12	1.14E11	109.21	0.000	1.000
ListLength* Update	6.41E11	16	4.00E10	38.33	0.000	1.000
FalsePositive* Update	1.54E11	16	1.29E10	12.32	0.000	1.000
Error	2.59E11	248	1.04E9			
$\mathcal{R}^2 = 0.947$		$\sigma = 32.25E3$		$CV = 2.27\%$		

Table A.2: *ANOVA model generated for the parametrization of the ESC when query distribution is* $Zipf_{\alpha=1.0}$



(a) Residues (b) Histogram

Figure A.3: *Residue analysis for the ANOVA model of ESC configuration for* $Zipf_{\alpha=1.0}$.

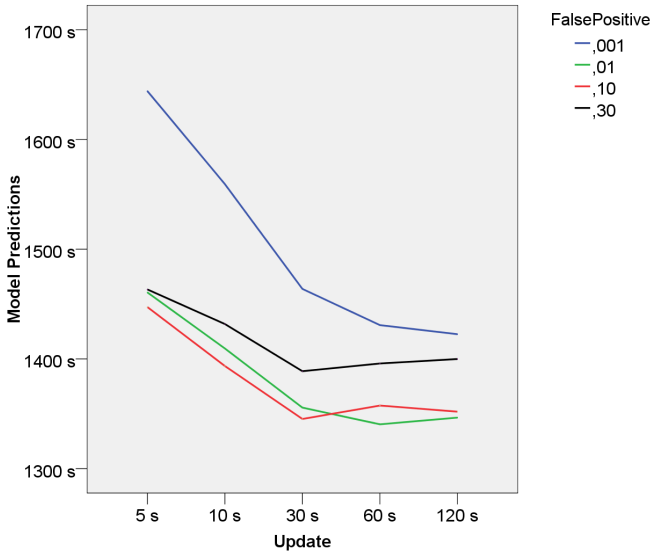


Figure A.4: *Model predictions for* $Zipf_{\alpha=1.0}$

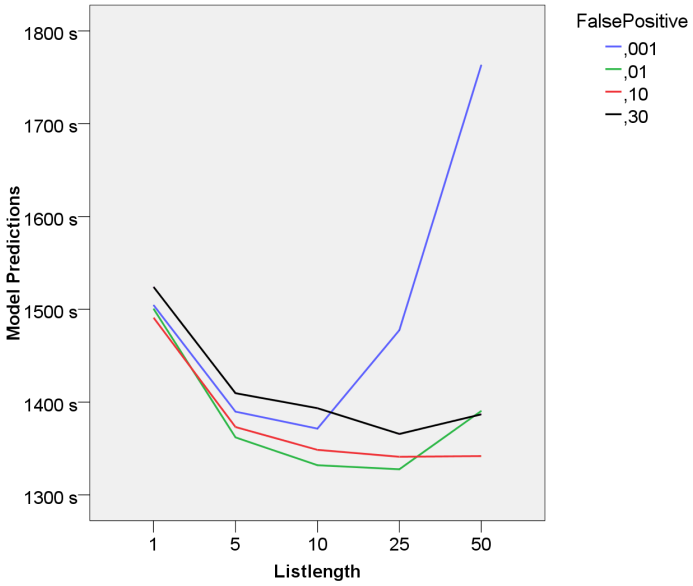


Figure A.5: *Model predictions for* $Zipf_{\alpha=1.0}$

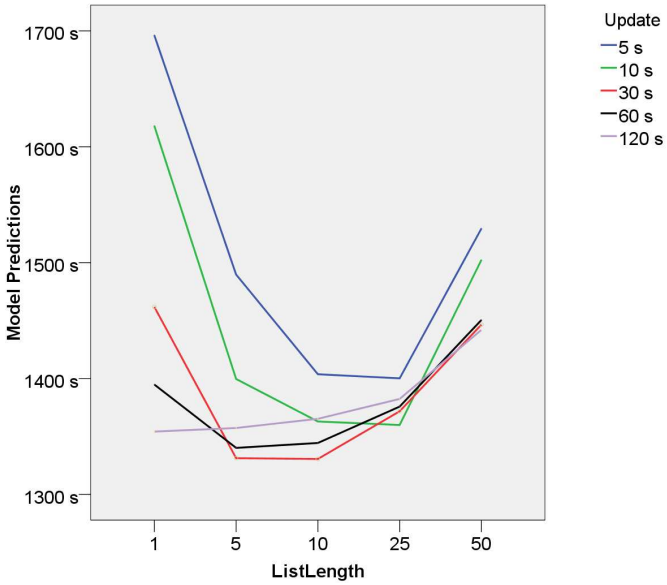
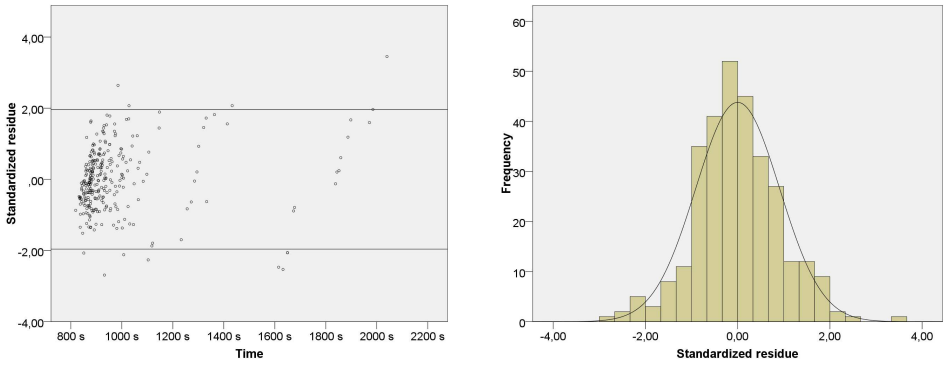


Figure A.6: *Model predictions for* $Zipf_{\alpha=1.0}$

Factor	SS	D.F.	MSS.	F	Significance	Power
ListLength	2.96E12	4	7.40E11	538.42	0.000	1.000
FalsePositive	4.58E11	3	1.52E12	1111.27	0.000	1.000
Update	1.26E11	4	3.15E10	22.89	0.000	1.000
ListLength* FalsePositive	5.64E12	12	4.70E11	341.96	0.000	1.000
ListLength* Update	2.06E11	16	1.29E10	9.37	0.000	1.000
FalsePositive* Update	2.23E11	12	1.85E10	13.51	0.000	1.000
Error	3.41E11	248	1.37E7			
$\mathcal{R}^2 = 0.976$		$\sigma = 37.03$		$CV = 3.78\%$		

Table A.3: *ANOVA model generated for the parametrization of the ESC when query distribution is* $Zipf_{\alpha=1.4}$



(a) Residues (b) Histogram

Figure A.7: *Residue analysis for the ANOVA model of ESC configuration for* $Zipf_{\alpha=1.4}$.

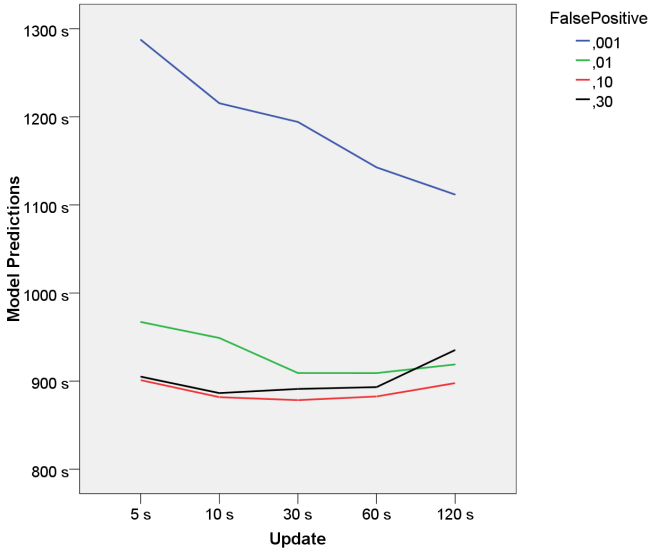


Figure A.8: *Model predictions for* $Zipf_{\alpha=1.4}$

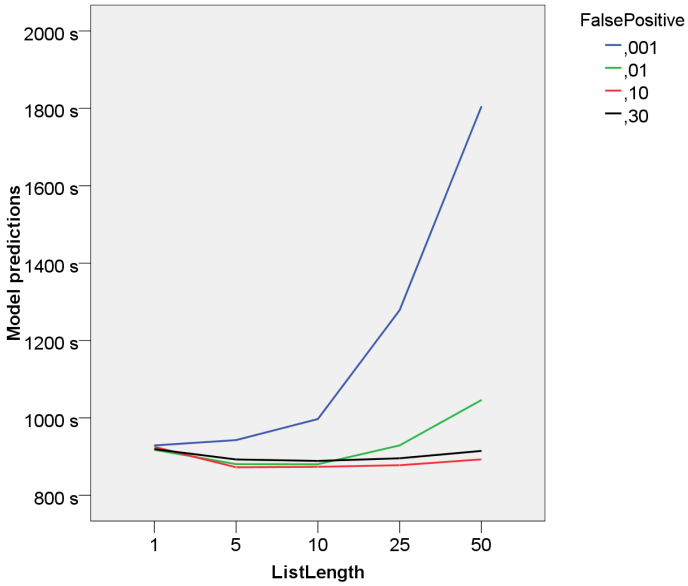


Figure A.9: *Model predictions for* $Zipf_{\alpha=1.4}$

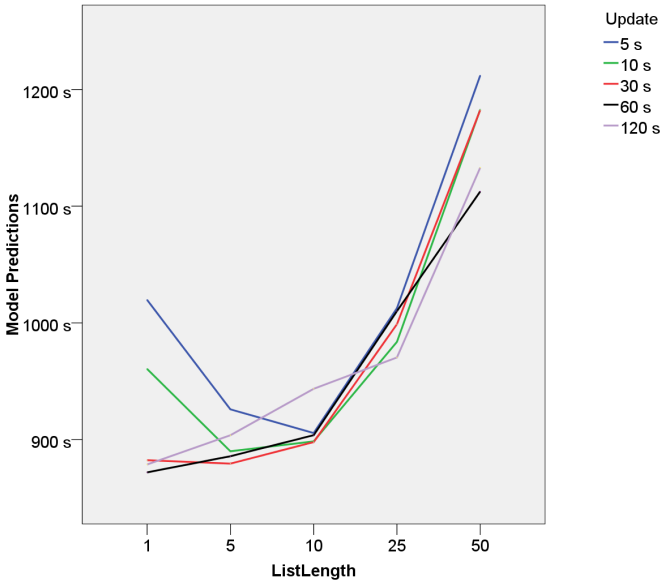


Figure A.10: *Model predictions for* $Zipf_{\alpha=1.4}$

Appendix B
Code of the Disk Direct I/O library

In the experiments of this thesis, we have implemented a small library to read the documents from the file system using direct I/O. Direct I/O bypasses the operating system buffer cache, and does not store a copy of the data read from disk in the main memory of the computer. This technique is usually applied by software that knows its workload, such as databases [56], because the own program can implement a cache policy more adapted to its usual workload instead of the general operating system policies. In our case, direct I/O ensures that the operating system cache does not interfere with our measures of the cache performance.

In our implementation, all the documents of the collection are stored sequentially in a large file. The position of a document is given by an offset in bytes from the beggining of the file. We activate direct I/O with the addition of the flag "`O_DIRECT`" to the `open` operating system function. Note that, in order to use direct I/O, the data blocks read from the disk must be aligned to the logic block size.

B.1 Source code

```
#include <jni.h>
#include "Utils_Writers_FileStorageDirectIO.h" //Jni stub

#define _FILE_OFFSET_BITS 64
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
```

```
/*
 * Class:     Utils_Writers_FileStorageDirectIO
 * Method:    readData
 * Signature: ([BIJ)V
 * Parameters:
 * j_fileName: Filename in which the document is stored.
 * dataArray: Buffer where the document is returned
 * j_size: Number of bytes of the document
 * j_offset: Position of the document in the file (in bytes)
 */
JNIEXPORT void JNICALL Java_Utils_Writers_FileStorageDirectIO_readData(
                  JNIEnv *env, jobject obj, jstring j_fileName,
                  jbyteArray dataArray, jint j_size, jlong j_offset){

    long long error;
    jboolean  isCopy = 0;

    const char *fileName = env->GetStringUTFChars(j_fileName, &isCopy);

    //Get from the JVM the original buffer, not a copy
    char*     data   = (char*)(env)->GetPrimitiveArrayCritical(dataArray, &isCopy);
    int       fd     = open(fileName, O_RDONLY | O_LARGEFILE |O_DIRECT);
    env->ReleaseStringUTFChars(j_fileName, fileName);

    long long offset      = (long long) j_offset;
    int pageSize          = getpagesize();
    offset                = pageSize*(j_offset/pageSize);
    char* diskAlignedData = (char*) valloc(pageSize);


#ifdef _FILE_OFFSET_BITS
    error = lseek64(fd, offset, SEEK_SET);
#else
    error = lseek  (fd, offset, SEEK_SET);
#endif

    if (error<0 ){
        perror("Lseek error (misaligned read?)");
        std::cerr << "fd     : "         << fd               << std::endl;
        std::cerr << "offset : "         << offset           << std::endl;
        std::cerr << "sizeof(offset):"   << sizeof(offset)   << std::endl;
        std::cerr << "j_offset: "        << j_offset         << std::endl;
        std::cerr << "sizeof(j_offset):" << sizeof(j_offset) << std::endl;
        std::cerr << "j_size : "         << j_size           << std::endl;
    }

    long long bytesToRead   = (size_t) j_size;
    char*     readPosition  = diskAlignedData + j_offset - offset;
    char*     writePosition = data;
    while(bytesToRead > 0){
        error = read( fd, diskAlignedData,  pageSize);
        if (error<0){
```

```
            perror("Read error (misaligned read?)");
            std::cerr << "fd              : " << fd              << std::endl;
            std::cerr << "diskAlignedData : " << diskAlignedData << std::endl;
            std::cerr << "pageSize        : " << pageSize        << std::endl;
        }

        while(bytesToRead > 0 && readPosition < (diskAlignedData+pageSize) ){
            *writePosition = *readPosition;
            readPosition++;
            writePosition++;
            bytesToRead--;
        }
        readPosition = diskAlignedData;
    }
    error = close(fd);
    (env)->ReleasePrimitiveArrayCritical(dataArray, data, 0);

    free(diskAlignedData);
}
```

Appendix C
Published papers

C.1 Publications related to this thesis

- D. Dominguez-Sal and M. Surdeanu. A Machine Learning Approach for Factoid Question Answering. *SEPNL*, 2006
- D. Dominguez-Sal, J.L. Larriba-Pey and M. Surdeanu: A Multi-layer Collaborative Cache for Question Answering. *Euro-Par*, 2007.
- D. Dominguez-Sal, M. Surdeanu, J. Aguilar-Saborit, J.L. Larriba-Pey. Cache-aware load balancing for question answering. *CIKM*, 2008.
- D. Dominguez-Sal, M. Perez-Casany, J.L. Larriba-Pey. Cache-aware load balancing vs cooperative caching for distributed search engines. *HPCC*, 2009.
- D. Dominguez-Sal, M. Surdeanu, J. Aguilar-Saborit, J.L. Larriba-Pey. Cooperative caching based on the recent data access history. *Submitted to TPDS*.

C.2 Other publications

- M. Surdeanu, D. Dominguez-Sal, and P. Comas. Design and Performance Analysis of a Factoid Question answering System for Spontaneous Speech Transcriptions. *Interspeech*. 2006
- D.Ferres, S.Kanaan, D.Dominguez-Sal, E.Gonzalez, A. Ageno, M. Fuentes, H. Rodrguez, M. Surdeanu, J. Turmo. TALP-UPC at TREC 2005: Experiments Using a Voting Scheme Among Three Heterogeneous QA Systems. *TREC*, 2005.

Bibliography

[1] J. Aguilar-Saborit, V. Muntes-Mulero, P. Trancoso, and J. Larriba-Pey. Dynamic Count Filters. *SIGMOD Rec.*, 35(1):26–32, 2006.

[2] J. Allan, J. Aslam, N. J. Belkin, C. Buckley, J. Callan, W. Croft, S. Dumais, N. Fuhr, D. Harman, D. Harper, D. Hiemstra, T. Hofmann, E. Hovy, W. Kraaij, J. Lafferty, V. Lavrenko, D. Lewis, L. Liddy, R. Manmatha, A. McCallum, J. Ponte, J. Prager, D. Radev, P. Resnik, S. Robertson, R. Rosenfeld, S. Roukos, M. Sanderson, R. Schwartz, A. Singhal, A. Smeaton, H. Turtle, E. Voorhees, R. Weischedel, J. Xu, and C. Zhai. Challenges in information retrieval and language modeling. *SIGIR Forum*, 37(1):31–47, 2003.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, 1990.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 109–126, 1995.

[5] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, 1996.

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

[7] M. Atallah. *Algorithms and theory of computation handbook*. CRC Press, 1999.

[8] R. Baeza-Yates. Challenges in the interaction of information retrieval and natural language processing. In *CICLing*, pages 445–456, 2004.

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

[10] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190, 2007.

[11] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *TWEB*, 2(4), 2008.

[12] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*. ACM press New York, 1999.

[13] R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, pages 56–65, 2003.

[14] N. Barghouti and K. Gail. Concurrency control in advanced database applications. *ACM Comput. Surv.*, 23(3):269–317, 1991.

[15] N. Barghouti and G. Kaiser. Concurrency control in advanced database applications. *ACM Comput. Surv.*, 23(3):269–317, 1991.

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

[17] S. Beitzel, E. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. In *SIGIR*, pages 321–328, 2004.

[18] L. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

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

[20] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. pages 126–134, 1999.

[21] A. Z. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2003.

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

[23] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, 193, 1997.

[24] 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.

[25] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

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

[27] 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.

[28] J. Chu-Carroll, K. Czuba, P. Duboue, and J. Prager. Ibm's piquant ii in trec2005. In *Proceedings of the Fourthteen Text REtrieval Conference (TREC)*, 2005.

[29] CLEF. CLEF website. *http://www.clef-campaign.org/*, 2000-2009.

[30] S. Cohen and Y. Matias. Spectral Bloom Filters. *SIGMOD'03: In Proceedings of the ACM SIGMOD international conference on Management of data*, pages 241–252, 2003.

[31] T. Cortes, S. Girona, and J. Labarta. PACA: A Cooperative File System Cache for Parallel Machines. *Euro-Par*, pages 477–486, 1996.

[32] T. Cortes, S. Girona, and J. Labarta. Avoiding the cache-coherence problem in a parallel/distributed filesystem. *Proceedings of the High-Performace Computing and Networking*, pages 860–869, 1997.

[33] T. Cortes, S. Girona, and J. Labarta. Design issues of a cooperative cache with no coherence problems. *Workshop on I/O in parallel and distributed systems*, pages 37–46, 1997.

[34] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *OSDI*, pages 267–280, 1994.

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

[36] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[37] D. Dominguez-Sal, J. Aguilar-Saborit, M. Surdeanu, and J. Larriba-Pey. Cooperative caching based on the recent data access history. *Submitted to IEEE Transaction on Parallel and Distributed Systems*, 2009.

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

[39] D. Dominguez-Sal, M. Perez-Casany, and J. Larriba-Pey. Cooperative caching vs cache-aware load balancing. In *HPCC*, 2009.

[40] D. Dominguez-Sal and M. Surdeanu. A Machine Learning Approach for Factoid Question Answering. SPNL, 2006.

[41] D. Dominguez-Sal, M. Surdeanu, J. Aguilar-Saborit, and J. Larriba-Pey. Cache-aware load balancing for question answering. In *CIKM*, pages 1271–1280, 2008.

[42] A. Echihabi, U. Hermjakob, E. Hovy, D. Marcu, E. Melz, and D. Ravichandran. Multiple-engine question answering in textmap. In *TREC*, pages 772–781, 2003.

[43] B. S. Everitt. *The Cambridge Dictionary of Statistics*. Cambridge University Press, 3rd edition edition, 2006.

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

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

[46] M. Feeley, W. Morgan, E. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 201–212, 1995.

[47] D. Ferres, S. Kanaan, D. Dominguez-Sal, E. Gonzalez, A. Ageno, M. Fuentes, H. Rodriguez, M. Surdeanu, and J. Turmo. TALP-UPC at TREC 2005: Experiments Using a Voting Scheme Among Three Heterogeneous QA Systems. *Proceedings TREC*, 2005.

[48] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[49] Google blog website. We knew the web was big... *http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html*, Retrieved in March 2009.

[50] Google Trends website. Trends on 25th june 2009. *http://www.google.com/trends/hottrends?q=michael+jackson+dead+2009\&date=2009-6-25\&sa=X*, Retrieved in July 2009.

[51] Google Trends website. Trends on the query: Champions league. *http://www.google.com/\-trends?q=champions+league\-\&ctab=0\&geo=all\&date=all\-\&sort=0*, Retrieved in July 2009.

[52] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, 1997.

[53] J. Hartman and J. Ousterhout. The zebra striped network file system. *ACM Trans. Comput. Syst.*, 13(3):274–310, 1995.

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

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

[56] IBM DB2 website. DB2 UDB Version 8.2 update considerations. *http://www-01.ibm.com/support/docview.wss?uid=swg21223978*.

[57] S. Jiang, K. Davis, F. Petrini, X. Ding, and X. Zhang. A Locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance. *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, 2006.

[58] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a):364–373, 1990.

[59] Juniper website. Juniper Networks Introduces Breakthrough 100 Gigabit Ethernet Interface for T Series Routers. *http://www.juniper.net/us/en/company/press-center/press-releases/2009/*, Retrieved in June 2009.

[60] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *WWW8 / Computer Networks*, 31(11-16):1203–1213, 1999.

[61] M. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. *IEEE Transactions on Knowledge and Data Engineering*, 14(6):1317–1329, 2002.

[62] M. Korupolu, C. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *SODA*, pages 586–595, 1999.

[63] P. Krueger and M. Livny. A comparison of preemptive and non-preemptive load distributing. In *ICDCS*, pages 123–130, 1988.

[64] T. Kudo and Y. Matsumoto. Fast methods for kernel-based text analysis. In *ACL*, pages 24–31, 2003.

[65] S. Lam and M. Ozu. Querying Web data-the WebQA approach. *Web Information Systems Engineering, 2002. WISE 2002.*, pages 139–148, 2002.

[66] K. Lillis and E. Pitoura. Cooperative xpath caching. In *SIGMOD Conference*, pages 327–338, 2008.

[67] Lin, D. Proximity based Thesaurus. *http://www.cs.ualberta.ca/~lindek/downloads.htm*, 2006.

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

[69] Lustre website. Lustre file system (whitepaper). *http://www.lustre.org/docs/whitepaper.pdf* .

[70] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. *International World Wide Web Conference*, pages 107–110, 1995.

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

[72] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*. USENIX, 2003.

[73] H. Meuer, E. Strohmaier, J. Dongarra, and S. H. Top 500 supercomputers. *http://www.top500.org/lists*.

[74] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, 2007.

[75] D. Moldovan, M. Pasca, S. Harabagiu, and M. Surdeanu. Performance issues and error analysis in an open-domain question answering system. *ACM Trans. Inf. Syst.*, 21(2):133–154, 2003.

[76] D. Mongomery. *Design and Analysis of Experiments*. Wiley, 5th edition edition, 2000.

[77] N. B. N. Johnson, S. Kotz. *Continuous Univariate Distributions*. Wiley, 1995.

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

[79] Nvidia. Geforce gtx 295 specifications. *http://www.nvidia.com/object/product_geforce_gtx_295_us.html*, Retrieved on Nov. 2009.

[80] E. Nyberg, R. Frederking, T. Mitamura, M. Bilotti, K. Hannan, L. Hiyakumoto, J. Ko, F. Lin, L. Lita, V. Pedro, et al. JAVELIN I and II Systems at TREC 2005. *Proceedings of Text REtrieval Conference*, 2005.

[81] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In P. Buneman and S. Jajodia, editors, *ACM SIGMOD*, pages 297–306. ACM Press, 1993.

[82] 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.

[83] K. Papadopoulos, K. Stavrou, and P. Trancoso. Helpercoredb: Exploiting multicore technology to improve database performance. In *IPDPS*, pages 1–11, 2008.

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

[85] R. Porkess. *Collins dictionary of Statistics*. Collins, 2nd edition edition, 2005.

[86] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[87] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *Infoscale*, page 34, 2006.

[88] X. Qin, H. Jiang, Y. Zhu, and D. Swanson. Towards load balancing support for i/o-intensive parallel jobs in a cluster of workstations. In *CLUSTER*, pages 100–, 2003.

[89] 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.

[90] L. Ramaswamy, A. Iyengar, and J. Chen. Cache Clouds: Cooperative Caching of Dynamic Documents in Edge Networks. *2nd International Conference on Collaborative Computing (CollaborateCom-2006)*, 2006.

[91] L. Ramaswamy and L. Liu. An expiration age-based document placement scheme for cooperative web caching. *Knowledge and Data Engineering, IEEE Transactions on*, 16(5):585–600, 2004.

[92] L. Ramaswamy, L. Liu, A. Iyengar, and G. Tech. Cache Clouds: Cooperative Caching of Dynamic Documents in Edge Networks. *ICDCS*, pages 229–238, 2005.

[93] S. Ratsanamy, P. Francis, M. Handley, and R. Karp. A Scalable Content-Addressable Network. *ACM SIGCOMM Conference*, pages 161–172, 2001.

[94] D. Roussinov, W. Fan, and J. Robles-Flores. Beyond keywords: Automated question answering on the web. *Commun. ACM*, 51(9):60–65, 2008.

[95] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218(0):329–350, 2001.

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

[97] 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.

[98] P. Sarkar and J. H. Hartman. Efficient cooperative caching using hints. In *OSDI*, pages 35–46, 1996.

[99] C. Silverstein, M. R. Henzinger, H. Marais, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[100] A. Spink, D. Wolfram, M. Jansen, and T. Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science and Technology*, 52(3), 2001.

[101] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.

[102] A. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *VLDB*, pages 1081–1092, 2006.

[103] Sun Microsystems. Ultrasparc t2 and t2 plus processors. specifications. *http://www.sun.com/processors/UltraSPARC-T2/specs.xml*, Retrieved on Nov. 2009.

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

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

[106] M. Surdeanu, J. Turmo, and E. Comelles. Named entity recognition from spontaneous open-domain speech. *Interspeech-2005*, 2005.

[107] S. T. System. Switchboard corpus. (LDC catalog number LDC97S62). *http://www.icsi.berkeley.edu/real/stp/*.

[108] E. Voorhees. The TREC-8 question answering track report. *NIST SPECIAL PUBLICATION SP*, pages 77–82, 2000.

[109] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. TnT A Statistical Part-of-Speech Tagger. *Applied NLP Conference, 2000.*, 2000.

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

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

[112] 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*, pages 16–31, 1999.

[113] Y. Xie and D. R. O'Hallaron. Locality in search engine queries and its implications for caching. In *INFOCOM*, 2002.

[114] L. Xin and D. Roth. Learning question classifiers: the role of semantic information. *Natural Language Engineering*, 12(03):229–249, 2005.

[115] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. *Computer*, 74, 2001.

[116] Z. Zheng. AnswerBus question answering system. *Human Language Technology Conference*, pages 24–27, 2002.

List of Figures

List of Tables

List of acronyms

AE: Answer extraction.

AF: Affinity.

ANOVA: Analysis of variance.

BPR: Broadcast petition recently.

CBF: Count bloom filter.

CPU: Central processing unit.

DCF: Dynamic count filter.

EA: Expiration age.

ESC: Evolutive summary counters.

ICP: Internet cache protocol.

I/O: Input/output.

IR: Information retrieval.

LAN: Local area network.

LRU: Least recently used.

GLM: General linear model.

MLS: Minimum least squares.

MLE: Maximum likelihood estimation.

MSS: Mean sum of squares.

NLP: Natural language processing.

NERC: Named entity recognition and classification.

PC: Probability cost.

QA: Question answering.

QP: Question processing.

PR: Passage retrieval.

SC: Summary cache.

SS: Sum of squares.

TREC: Text retrieval conference.

SVM: Support vector machine.

WAL: Weighted Average Load.

WAN: Wide area network.

WLS: Weighted least squares.