Università Ca' Foscari di Venezia

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: number

Query Log Based Techniques to Improve the Performance of a Web Search Engine

Daniele Broccolo

Supervisor

Salvatore Orlando

PhD Coordinator

Riccardo Focardi

November, 2013

Author's e-mail: daniele.broccolo@unive.it

Author's address:

Dipartimento di Informatica

Università Ca' Foscari di Venezia

Via Torino, 155

30172 Venezia Mestre – Italia

tel. +39 041 2348411

fax. +39 041 2348419

web: `http://www.dsi.unive.it`

Abstract

Every user leaves traces of her/his behaviour when she/he surfs the Web. All the usage data generated by users is stored in logs of several web applications, and such logs can be used to extract useful knowledge for enhancing and improving performance of online services. Also Search Engines (SEs) store usage information in so-called query logs, which can be used in different ways to improve the SE user experience. In this thesis we focus on improving the performance of a SE, in particular its effectiveness and efficiency, through query log mining.

We propose to enhance the performance of SEs by discussing a novel Query Recommender System. We prove that is possible to decrease the length of a user's query session by unloading the SE of part of the queries that the user submits in order to refine his initial search. This approach helps the user find what she/he is searching in a shorter period of time, while at the same time decreasing the number of queries that the SE must process, and thus decreasing the overall server load.

We also discuss how to enhance the SE efficiency by optimizing the use of its computational resources. The knowledge extracted from a query log is used to dynamically adjust the query processing method by adapting the pruning strategy to the SE load. In particular query logs permit to build a regressive model used to predict the response time for any query, when different pruning strategies are applied during query processing. The prediction is used to ensure a minimum quality of service when the system is heavily loaded, by trying to process the various enqueued queries by a given deadline. Our study also addresses the problem of the effectiveness

of query results by comparing their quality when dynamic pruning is adopted to reduce the query processing times. Finally, we also study how response times and results vary when, in presence of high loads, processing is either interrupted after a fixed time threshold elapses or dropped completely. Moreover, we introduce a novel query dropping strategy based on the same query performance predictors discussed above.

Acknowledgments

I would like to express my sincere gratitude to my thesis advisor, Professor Salvatore Orlando, who has consistently inspired me in this study and provided me precious suggestions and advice. Without his attentive guidance, endless patience and encouragement through the past year, this thesis would not have been possible to accomplish. My sincere thanks also go to Fabrizio Silvestri, Nicola Tonellotto and Raffaele Perego. They have provided me lots of constructive suggestions and comments on the content of the thesis with their professional experience and extensive knowledge in research conducting.

Besides, I want to thank present and past members of the HPC lab for sharing their knowledge, for the hundreds of coffee breaks and for providing a great work environment.

Thanks also to members of the Terrier Lab, Craig Macdonald, Iadh Ounis and Dyaa AlBakour for the for their help and company during my stay in Glasgow.

I also thank my friends (too many to list here but you know who you are!) for providing support and friendship that I needed.

I especially thank my flatmates Diego Marcheggiani, Stefano Baccianella, Paulo Cintia and Lalaura Giglioni, for tolerating me every day during these years.

A particular thank also go to Cristina Muntean for the lots of constructive suggestions and comments on the structures of this theses.

Last but not least, a special thank to my mom, dad, and sister, that are my first supporters.

Contents

List of Figures

List of Tables

Introduction

Query log generated by the interaction between users and web search engine (SE) is a great source of information to enhance user experience and to improve the performance of the SE itself. The information stored in the query log is collected from the activity of users that make searches on SE. The number of users and the variety of searched topics make the analysis of the query log complex. For these reasons query log mining is the subject of many research papers.

In this thesis we use query log mining techniques to enhance the performance of the SE both to improve the user experience and to optimise the usage of the SE resources.

In this thesis we propose to improve the performance of a SE from different point of views. Firstly, we propose a novel query recommender system to help users shorten their query sessions. The idea is to find shortcuts to speed up the user interaction with the SE and decrease the number of queries submitted. The proposed model, based on the pseudo-relevance feedback, formalizes a way of exploiting the knowledge mined from query logs in order to help users rapidly satisfy their information need. We tested the algorithm proposed exploiting different metrics both to prove the goodness of recommendations provided and to prove that our algorithm is a valid solution to the *search shortcuts problem*.

Secondly, the satisfaction of a user can be enhanced also from a different perspective. To this extent, we propose to optimize the query processing phase to obtain an improved trade-off between efficiency and accuracy. In particular, commercial SEs have to deal with a lot of simultaneous queries from numerous different users. At the same time the query stream is characterized of a discontinuous arrival rate that make difficult for the SE to deal with high query load periods without oversizing the architecture. In this scenario we exploit the past queries submitted by users to build a machine learning model for query efficiency prediction. This model is then used to dynamically select a suitable query processing (retrieval) strategy to handle the queries and satisfy a time threshold deadline, thus reducing the query processing time during intense query traffic. In particular, when deciding between processing strategies, our approach considers the time necessary to satisfy the per-query deadlines for all queries in queue for being dealt with by search servers. In this way the system can fairly allocate the available resources for enqueued queries. The proposed techniques are validated and tested in a distributed SE deployed over a multi-node cluster. At the same time we study also the cases when the query arrival rate is unsustainable for the system and we propose a method to decrease the number of query dropped or stopped.

In the Section I.1 we present the contributions brought on from the proposed solutions.

I.1 Contributions

In this thesis we describe three main contributions as follows.

Search Shortcuts for User Sessions: We describe a method to recommend queries helping the users to easily find informative queries. This work aims to resolve the query shortcut problem and is firstly introduced in 2012 in the IPM Journal [23]. The query shortcuts problem is formally defined as a problem related to the recommendation of queries in search engines and the potential reductions obtained in the users session length. By exploiting a weak function for assessing the similarity between the current query and the knowledge base built from historical user sessions, we re-conduct the suggestion generation phase to the processing of a full-text query over an inverted index. The resulting query recommendation technique is highly efficient and scalable, and it is less affected by the data-sparsity problem than most state-of-the-art proposals. Thus, it is particularly effective in generating suggestions for rare queries occurring in the long tail of the query popularity distribution. The quality of suggestions generated is assessed by evaluating the effectiveness in forecasting the users' behaviour recorded in historical query logs, and on the basis of the results of a reproducible user study conducted on publicly-available, human-assessed data. The experimental evaluation conducted shows that our proposal remarkably outperforms two other state-of-the-art solutions, and that it can generate useful suggestions even for rare and unseen queries.

Load-Sensitive Selective Pruning: We explore different techniques of query processing and we analyse their behaviour in a high load environment, this work is published at CIKM'13 [21]. Our idea is to improve the performance of a SE in a heavy loaded environment using the query log mining. In particular we introduce a novel method to dynamically adapt the pruning strategy of each query to the instant workload of the search sever. With the goal of answering a query within a given upper bound in terms of query response time, the method uses machine learning models to predict the running time of each pruning strategy, selects the one that fits into the time bound and maximizes the effectiveness of results. The proposed method, by adapting the query processing to the workload, is able to face peak load periods reducing sensitively the processing time and keeping the quality of the results at acceptable levels. For each search server there is a queue of queries to be processed, and the number of queued queries depends on the query arrival rate. Our framework is able to optimise the queue management strategies so that not only the currently processed query is optimized but also the queries that are already in the processing queue of each search server. We validate our approach on $10,000$ queries from a standard TREC dataset with over 50 million documents, and compare it with several baselines. These experiments encompass testing the system under different query loads and different maximum tolerated query response times. Our results show that, at the cost of a marginal loss in terms of response quality, our search system is able to answer 90% of queries within half a second during times of high query volume.

Query Processing in Highly-Loaded Search Engines: The idea of exploiting a predicting framework to dynamically adapt the processing strategy is used and extended on another contribution, published at SPIRE'13 [22], where we analyse query dropping strategy in high load environment. By using predictors to estimate the query response time, we propose to exploit the estimated response time to pro-actively control the query dropping behaviour. Our experiments show that our proposed dropping strategy is able to decrease the number of queries dropped, improving overall effectiveness whilst attaining query response times within the time threshold. For instance, for a query arrival rate of 100 queries per second, our strategy is able to answer up to 40% of the queries without degrading effectiveness, while for our baseline strategies this happens for only 10% of queries.

I.2 Outline

This thesis is structured as follows.

In Chapter 1 there is a brief overview of the structure of SE and its components while also discussing the state-of-the-art solutions. After a general discussion about current problems and goals of a SE, the main phases of a search engine are described: crawling, indexing and querying. We focus our presentation mainly on describing the query processing phase that is one of the central point of this thesis.

In Chapter 2 we analyse the web search topic by taking into consideration the problems referring to the distributed environment like query routing and index partitioning strategies. Based on those observations we configure a distributed search engine for our experiments and we describe the resulting framework.

Since all of our works exploit query log mining techniques, in Chapter 3.1 we analyse the state-of-the-art on this topic including typical aspect of a query log, common features, and data mining techniques such as session extraction.

The following three Chapters 4, 5 and 6 elaborate on the contributions mentioned in Section I.1. Chapter 4 describes: a novel algorithm to efficiently and effectively generate query suggestions that is robust to data sparsity, a novel evaluation methodology with which we can compare the effectiveness of suggestion mechanisms and an extensive evaluation comparing on the same basis the proposed solution with two state-of-the-art algorithms.

Chapter 5 describes: a load-sensitive selective pruning framework for bounding the permitted processing time of a query and an accurate approach for query efficiency prediction of term-at- a-time dynamic pruning strategies.

Chapter 6 describes a novel query dropping strategy by using predictors to estimate the query response time for document-at-a-time.

Conclusions and future work are discussed in Chapter 7.

1 Web Search

The World Wide Web (WWW) is a huge and complex network that consists of an extensive collection of inter-linked hypertext documents. SEs aim to discover, map and index the web in order to enable easy information search. The main problems of this process is that the web is continuously growing and changing, especially if we think at news websites and social networks. Some studies [3] estimate that the number of pages in the WWW, using the documents indexed by the main commercial web search engines, to at least 14.79 billion pages. To effectively search through this enormous, and often dynamic, quantity of information, SEs must exploit complex and efficient algorithms, in general distributed over multiple servers and often located in different sites. Different servers are also used to accomplish different tasks that the SEs have to fulfil. A SE consists of several precesses and phases: *exploration* meaning crawling the web by following the links in the pages; *indexing* which allows to organize the data collected from the crawling and storing it in a special data structure, such as the inverted index; *query processing* which exploits the indices in order to retrieve information in reply to queries. The distributed environment, typical for SE architecture, makes all these phases more complex. In fact, each algorithm must be designed to interact with several machines and to avoid flooding web servers.

1.1 Search engines

SEs supply results to a large number of users in a timely fashion. A typical distributed architecture of a commercial SE for query processing consists of a broker and a pool of query servers. The broker receives queries from users and forwards them to all (or a subset of) query servers. Each query server exploits the posting lists of its local index to compute partial results. These results are returned to the broker, which produces the final ranking of documents that is presented to the user. Each local index is a partition of the global index, which is in turn distributed over different servers.

Cambazoglu et al. [34] have presented a comprehensive survey of the distributed architecture and scalability challenges in modern Web search engines. According to their architecture, query processing can be classified in four types of granularity: single node, multi-node cluster, multi-cluster site and multi-site engine. In this thesis we deal with query processing in a multi-node cluster, where typically off-the-shelf commodity computers are employed to parallelise the query processing procedure (see Chapters 5 and 6). In the multi-node cluster architecture, a user query is issued to a broker node, which dispatches the query to the search nodes (Figure 1.1). The broker node is also responsible for merging the results retrieved from the search nodes and returning a final results set to the user. Query processing in a multi-node search cluster depends on the index partitioning techniques used to distribute the index among the search nodes (Figure 1.2). In case of *document-based partitioning* ([29, 50]), the broker issues the query to all search nodes in the cluster. Results are concurrently computed and returned to the broker. In case of *term-based partitioning* [76], the query is issued to only the search nodes that contain the posting lists associated with query terms. The contacted nodes compute results sets, where document scores are partial, by using their posting lists that are related to the query. These result sets are then transferred to the broker, which merges them into a global result set.

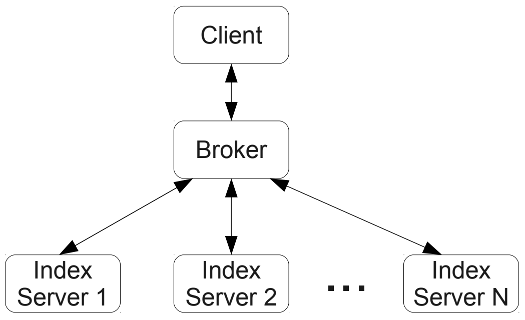

Figure 1.1: Classical architecture of a distributed search engine (based on [33]).

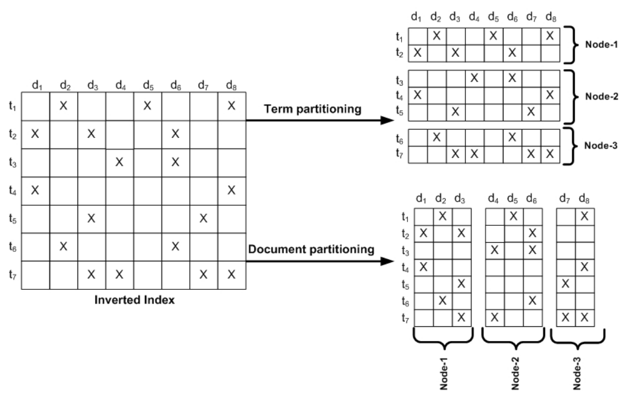

Figure 1.2: Differences between Document and Term Partitioning.

In the rest of this chapter we describe in more detail the different phases of the SEs in a distributed settings. We give particular emphasis to the query processing phase, that is the main phase analysed in this thesis.

1.2 Crawling

The crawling module is a software that downloads and collects relevant objects from the web. The implementation of such a module is very simple: it starts from an initial list of URLs to visit. The crawler stores these pages and identifies hyperlinks in the content which allow to discover new pages to visit, as represented on Figure

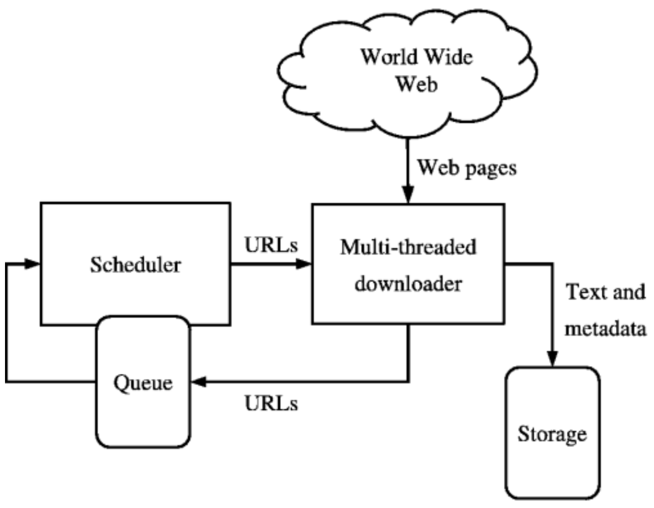

Figure 1.3: Single Node Crawling module.

1.3.

The main challenge of this module is to avoid multiple visits to the same page and, in the case of distributed crawling, is to avoid that multiple agents visit the same server simultaneously, thus overloading it. Large-scale web crawlers should be distributed because of the intensive network and CPU usage due to the hundreds of simultaneous connections to the web servers.

An important restriction for distributed web crawlers is to avoid overloading web servers. This is made possible by using a good policy for the partitioning of the web graph to be discovered. The partitioning should take into account also the load balancing between crawling servers, in order to optimize network and power capabilities [19]. A partitioning strategy can be organized to reduce the data exchange between agents: a possible solution is assigning all pages deployed by the same web server to the same agent [6].

Regarding crawling, another interesting topic is the periodicity of the crawling. In fact, lots of web users are actually interested on searching news or trending topics. To satisfy these users, SEs must scan the web frequently to collect the last news or update information from dynamic social network pages.

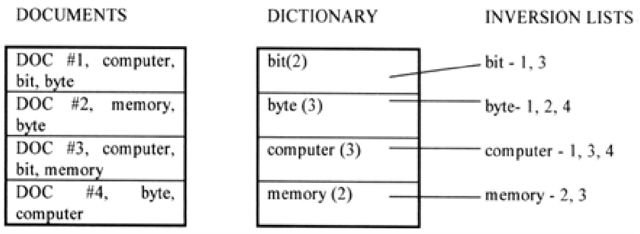

Figure 1.4: Inverted Index Structure

1.3 Indexing

During the crawling phase, the SE gathers a wide collection of documents. Each document in the collection has to be preprocessed; this operation implies removing HTML tags, stopwords, duplicates and performing tokenising, stemming and lowercasing operations. After this cleaning phase the data is ready to be indexed.

When speaking about indexing in SE, we mainly refer to building an inverted index [46, 118, 78]. The inverted index is a data structure that permits to easily find terms occurrences in documents. The inverted index is composed of a dictionary that holds all the terms found in the collection and a set of corresponding inverted lists, one for each term (Figure 1.4). Each inverted list is a list of postings where each posting consists of docids and statistics. The information stored in the inverted index are useful for the retrieval and ranking of all the documents matching the query terms.

As an example, if we want to compute `tf-idf` [90, 91] to rank the retrieved documents, we need to store, for each term, its frequency in the whole collection. We also need to know which documents the term occurs in and how often it occurs in each document. These last two pieces of information can be stored inside the inverted list associated with the term. Additional information, like the position of the term in the document, can also be useful for phrase query processing and can be stored directly in the index.

Due to the fact that the index must retrieve the documents on a short period of time it should not be charged with too much information. For more detailed statistics and data related to documents a structure called *docmap* can be used. It associates the document ID (a.k.a. docid) with the full document information: full text, URL, length (for length-normalized similarity metrics like BM25 [56, 88]).

Apart from the data stored in the inverted index, another important feature is the ordering of documents in the inverted lists [39]. The easiest way to design the inverted lists is to order the documents in ascending docid-order (docid-sorted). This structure is commonly used because it permits to easily update the inverted lists. Indeed, since new documents will have a greater docid, we can add them at the end of the respective posting list. The docid-sorted lists are also preferred in the case of Boolean queries. Docid sorted indices allow a better compression of the index because is possible to exploit gap-encoded compressor [116, 100]. In the modern context, saving space is still a very important issue. The aim is to reduce the number of machines needed to hold the index, their use of energy, and the amount of communication. Furthermore, in the last years, to improve the query response time, often SEs that store the index on the main memory to avoid disks latencies. Compressing the index is then indispensable for this type of applications [107, 44, 38]. To improve the performance of a document-ordered index, several strategies have been employed in the past. One of these is the use of the skipping index [109], which avoids reading all the postings, thus skipping part of the lists. This method does not always offer better performance, but it behaves well in the case of AND queries. More detailed description of the characteristics can be found in Section 1.4.

Other index structures exploit the fact that users do not need to examine all the documents matching their query. For this reason it is possible to avoid reading and processing all the postings by ordering the documents for highest-relevance. For example if we are using `tf-idf`, we can use a frequency ordering, as documents that contain more occurrences of the terms are in general more relevant than the others. Using this ordering we can stop reading the posting lists when we understand that the contribution is no longer sufficient to change the document ranking [83]. Another example is the impact factor ordering [50, 5] that computes the impact of each posting during the index building. Adding documents to this type of index is a more challenging task because documents are not added at the end of the posting lists, as in the case of the docid-sorded index, in fact they must be inserted at the corresponding position.

Other solutions aim to combine the advantages of the two techniques above. The Impact-Layered indices reassign the docid to documents after having sorted them by impact-factor.

The advantage is that while the impact factor ordering permits to use fast retrieval strategy, it is possible to easily compute operations like intersection thanks to the docid ordering. However, this index structure makes the updating of the index a more complex operation.

In a commercial SE or in a general large scale Information Retrieval (*IR*) system the index phase must also address the problem of distributing the index across multiple query servers. Due to the big dimension of the index, it is preferable to not build a single global index. Accordingly, as explained in Section 1.1, the architecture of such a system is in general distributed over several servers and located in different geographical sites (multi-site engine).

In each site, there are a set of query servers. The collection is typically split over part of these servers, while others are often used as a replicas. The common ways to split the index are in general for documents or for terms. The document partitioning is the simplest way to partition the index: in this case the collection is split before indexing by assigning different documents to different indices. The different indices can be build independently and each index can be stored in one or more servers. The other approach is to split the index by terms. In this case each shard contains only a subset of terms. This type of partitioning needs to have a global index and strategies to find, given a query, the servers that contain the query terms.

The index partitioning strategy affects the query processing phase, aspects we are going to describe in Section 1.4. Also the query routing is dependent of the index partitioning. This problem will be addressed in Section 2.

1.4 Query Processing

The strategies used to match documents to a query fall into two categories [77]: term-at-a-time (`TAAT`) [109, 59] where the posting lists of query terms are processed and scored sequentially, or document-at-a-time (`DAAT`) where posting lists are processed in parallel for all query terms. The `DAAT` strategy is obviously not possible when we do not have the posting list ordered by docid without a prior sorting phase, that anyway nullifies any possible advantages of an impact factor sorted posting list.

In `DAAT` processing, the final score of any document is computed immediately. This allowed to remove, from the list of candidates, the documents that do not have a high-enough score to stay in the top-n results. The disadvantage of the `DAAT` processing is that all the posting list must be in main memory.

In `TAAT` processing, the inverted list of each term is processed in turn. Each document receives a partial score and the couple document-score is stored in one accumulator. The set of accumulators is updated until the last term is processed. At the end of the process, the accumulators must be sorted to retrieve the top-n results.

Unlike `DAAT`, `TAAT` can maintain in main memory only one posting list at a time, but it also has to maintain the list of accumulators with the partial scores.

The choice between `DAAT` and `TAAT` is not trivial and depends on a lot of factors: index structure, collection features, type of the queries, etc.

As mentioned above, query processing is also affected by query type. Common methods to interpret a query is to use operators, AND or OR, between terms. The AND queries return only the results that contain all the query terms. They are faster to process than the OR ones, because we are able to avoid scoring all postings when noticing that for one of the terms a corresponding document is not present in the associated posting list. In the AND settings is also possible to exploit `skip indices` [77]. Skip indices are additional information that must be included in the index during the indexing phase.

A possible disadvantage for AND queries is that they may return a limited number of results, especially for long queries or when the query is expanded. OR queries, on the other hand, can return more results which can be eventually re-ranked using machine learning techniques (ML-rank) in order to improve the quality of the results [108]. OR queries, in fact, return any document that contains at least one term of the query. For this reason, the OR queries are slower, especially when the terms are very frequent in the collection and the posting lists are long. To improve the query response time of OR queries, pruning strategies have been developed. Pruning strategies allow skipping part of the posting list thus speeding up the processing. In literature, both lossless and lossful strategies have been studied for both for `TAAT` and `DAAT`.

1.4.1 Pruning Strategies

The dimension of the collection impacts on the length of the posting lists which in turn can worsen the performances of the SE in terms of query processing time. In literature there are a lot of different techniques to speed-up query processing by leveraging the previously mentioned pruning strategies. Various techniques have been proposed both for `TAAT` and `DAAT` processing methods. They can be classified as follows [109]:

safe optimizations guarantee that the ranking of the retrieved results is correct. This implies optimizing the way in which the evaluation technique is implemented, depending on the appropriate data structures or file access methods. The implementation highly influences the cost of the evaluation.

safe-up-to-rank-n optimizations guarantee that the ranking of the top-n retrieved results is correct (top-n documents appear in the same order as in the full ranking).

approximate optimizations do not guarantee the correctness of the ranking, but however, with a high degree of certainty, part of the top-n documents are present in the top-n positions of the full ranking.

One popular technique for `TAAT` is the `Quit-Strategy` and its variant `Continue-Strategy` described by Moffat and Zobel in [77]. The idea of `Quit-Strategy` is to limit the number of accumulators used to store the partial information of documents. `Quit-Strategy` fixes a maximum number of accumulators and when this number is reached, the computation is interrupted. `Continue-Strategy`, instead of interrupting the processing, continues to process updating only the accumulators already created. `Continue-Strategy` improves the quality of the results by paying in terms of processing time. Improved versions of `Continue-Strategy` add an adapting pruning [62] that estimates a threshold number of accumulators for each term. These techniques are *approximate* optimizations.

Dynamic pruning strategies have been also proposed for `DAAT`. Two of the most popular are WAND [26] and MAXSCORE[109], proposed to reduce the query processing time by avoiding to score a subset of documents (usually those likely to not be present in the final list of results). These two methods can be configured to be safe-up-to-rank-n.

WAND is a Boolean predicate standing for Weak AND. It takes as argument a list of Boolean variables $X_1, X_2, ..., X_k$, a list of associated positive weights, $w_1, w_2, ..., w_k$, and a threshold θ. By definition $WAND(X_1, w_1, ...X_k, w_k, \theta)$ is true iff $\sum_{1 \leq i \leq k} x_i w_i \geq \theta$ where x_i is the indicator variable for X_i, that is 1 if X_i is true and 0 otherwise. It can be observed that with WAND it is possible to implement both the AND and the OR operators. The tuning of the threshold θ lets WAND behave more like OR or more like AND according to preference.

MaxScore algorithm estimates the maximal score that a document can achieve to *early terminate* the retrieval step. This method requires evaluating query terms in ascending order of the global frequencies to place less influential terms at the end. MaxScore works estimating the maximal score for each document-term pair. Consequently, at any time of the computation, it is possible to compute the maximal achievable score (max-score). Based on this, the computation can be interrupted if the sum of the computed score and the remaining estimated score is not sufficient for a placement in top-k.

In this thesis (Chapters 5 and 6) we consider the Continue `TAAT` dynamic pruning strategy [77], which we denote by `TAAT-CS`. Within this strategy, as a posting list is processed, the partial score contributions of documents are held in temporary accumulators, that contain the final scores of documents once all posting lists are processed. To limit the processing time, `TAAT-CS` limits the number of accumulators. Once this limit is reached, no new accumulators are created, and the postings of further query terms can only update the existing accumulator values. In this stage of `TAAT-CS`, the use of skip pointers within the inverted file posting lists allows `TAAT-CS` to skip the decompression of postings which it does not need to examine, thus reducing actual IO and increasing efficiency. Although further improvements for the `TAAT-CS` strategy have been proposed [63], we leave their consideration to future work. Our choice of the `TAAT-CS` strategy is motivated by the fact that its overall efficiency is directly proportional to the number of accumulators to create in the first phase [77]. Indeed, the fine tuning of the number of accumulators gives us the flexibility to directly control the efficiency of the pruning strategy.

1.4.2 Selective Pruning

Dynamic pruning strategies, such as Wand and `TAAT-CS` can all be configured to be made more *aggressive*. In doing so, the strategy becomes more efficient, but at a possible loss of effectiveness [26]. For instance, reducing the maximum number of accumulators in the `TAAT-CS` strategy results in less documents being examined before the second stage of the algorithm commences, when no new accumulators can be added. Hence, reducing the number of accumulators increases efficiency, but can result in relevant documents not being identified within the set of accumulators, thereby hindering effectiveness [77].

Typically, the aggressiveness is selected a priori to any retrieval, independent of the query to be processed and its characteristics. However, in [106], Tonellotto et al. show how the WAND pruning strategy can be configured to prune more or less aggressively, on a per-query basis, depending on the expected duration of the query. They call this approach *selective pruning*.

Our thesis (Chapters 5 and 6) makes an important improvement to selective pruning, in observing that the appropriate aggressiveness for a query should be determined not just by considering the current query. Instead, our proposed *load-sensitive* selective pruning framework also accounts for the other queries waiting to be processed, and their predicted response times, together with their positions in the waiting queue. These are used to select the dynamic pruning aggressiveness in order to process the queries with a fixed time threshold, when possible, or to process it more efficiently, when the time constraint cannot be respected.

2

Distributed Architectures

2.1 Typical Structure

In Chapter 1 we have already mentioned about the general distributed structure of SEs. According to [34] SEs can be classified in different types of granularity. From a coarse-grained point of view, the SEs are structured as multi-cluster site engines (Figure 2.1). It means that a SE is geographically distributed to permit users all over the world access without significant network latencies.

Distributing a SE over multiple sites is necessary to overcome lack of services caused by regional problems, to exploit a bigger overall network capabilities and decrease network latencies. Another advantage is that this infrastructure allows an improved reliability whenever the data is replicated over geographical distributed servers. Indeed, to improve the reliability, sites must have an overlapped portion of the index to compensate eventual unreachable sites. Finding the optimal way to

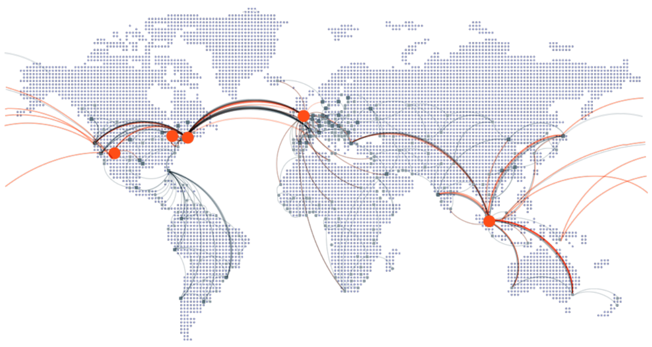

Figure 2.1: Classical architecture of a distributed search engine node (based on [33]).

distribute the index and which parts are better to replicate is not a trivial problem.

As an example, a solution is proposed in [31]. The authors suggest to store in one site only the documents crawled from a certain region. Local documents are in general more interesting for local queries. For example, different sites could store documents written in different languages, to which a user may not be interested. This approach is useful also for decreasing the communication between sites, this representing an advantage from the latency point of view. It has obviously some drawbacks. The more obvious is that not all the queries can be answered with the local data. Different alternatives can be explored: replicating popular documents over different sites and/or forwarding the query to different sites [32]. To obtain good overall performances one must find a trade off between the number of the site contacted, the dimension of the overlapping index and the quality of the results returned to users.

In this thesis we do not analyse in detail the multi-site SE problems, but we do take into consideration this level of granularity and problems related to multi-node cluster. Each site of the SE is in fact made of several components as represented in Figure 2.2. The front end of the architecture is called Query Broker. It is designed to broadcast the query to the query servers, collect any partial results and merge them into the final results presented to the user. Depending on the index partitioning strategy used, the Query Broker selects the query servers to query. In the case of shard replication the broker must implement also strategies to select replicas and balance the load over the different shards. Different strategies have been proposed to manage shard replication: from simple round-robin [70] to more complex solutions based on machine learning [48, 47].

When a query reaches a query server, it is processed immediately if the server is idle. Each query server comprises a *query processor*, which is responsible for tokenising the query and ranking the documents of its index shard according to a scoring function. Strategies such as dynamic pruning [26, 77, 110] can be used to process queries in an efficient manner on each query server.

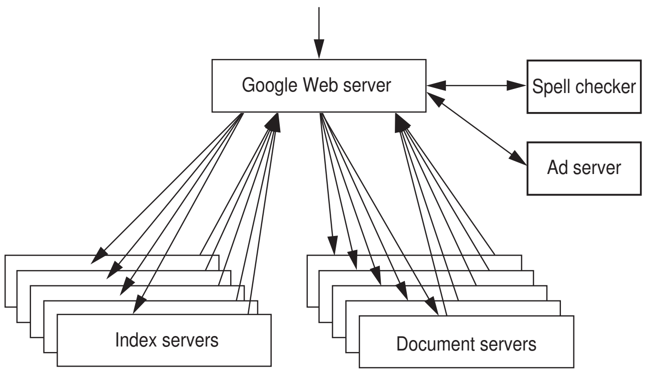

Figure 2.2: Google query serving architecture [13]

The SE architecture comprises also a set of document servers that hold full documents, snippets and other information about documents. In certain architectures it is possible to have some servers dedicated to caching, spell checking and advertising. [13].

2.2 Distributed Query Processing

In a distributed environment, for a given query, several servers are involved for the computation of results. The different techniques used to contact query servers are influenced by factors like the index partitioning strategy or the presence of shard replication. In this paragraph we describe the common ways to process a query in a distributed environment taking into account the common ways to distribute an index. Without loss of generality we do not deal about replication. As we see in the next chapters, the following discussion can be easily extended in a distributed environment with shard replication.

When the index is partitioned by documents, the broker has to query all the query servers to obtain all the matching documents required. Each query server can process the query autonomously, the only information that is globally required regards statistics (like global IDF) which help to properly score and rank the documents. The top-k documents of each query server are returned to the broker that collects and merges the $n*k$ results obtained, where n is the number of query servers. From a query processing point of view, the document partitioning has the advantage of being simple, and permits to automatically balance the load, since all the servers are involved in all the queries.

To involve less servers, some works [30, 95, 96, 94, 105] exploit collection selection strategies. These techniques could affect negatively the balancing of the load, but can decrease the computational resources needed for a query, improving the inter-query parallelism.

The query processing is more complex when the index is partitioned by terms. As described in 1.3 each server contains a subset of the terms and the broker has to know where the query terms are located. This approach takes its advantages from the fact that the query is computed by a limited group of servers, but in general posting lists are longer that those contained in the document partitioned index that limits the intra-query parallelism. This approach permits to process more queries at a time since only few servers are involved in the processing of a query. Actually, even though (the average length of a query is around 2 terms (see Chapter 3.1)), commercial SEs often use techniques of query expansion that could waste the advantage of term partitioning. Term partitioning is anyway used in at least one commercial SE [87].

When using a term partitioned index, query servers do not have full information about documents and they cannot compute the final scores. Partial scores must be sent to the broker for the final computation. Unfortunately to obtain the optimal solution all the partial score should be sent back to the broker. Since the list of partially scored documents is too long, sending it to the broker, can lead to bad performance due to an excessive use of the network. Furthermore is also costly for the broker to perform the merge of the lists. For this reason, some studies [86] suggest that a good choice is to return $k \times p \times c$ documents where p is the number of query server contacted and c is a constant. The bigger c, the better the effectiveness of results.

A possible problem of term partitioning is that the approach is quite inefficient for phrase queries. In fact, different terms can be located in different servers and the position of terms in documents needs to be communicated to the broker, thus increasing the communication cost. Lastly, the balancing of the load can be a problem due to the fact that the terms have different frequencies in the collection and some of these can be referred too frequently.

In order to resolve the above mentioned problems of term partitioning, over time several techniques have been proposed in order to balance the load or to decrease communication cost with a targeted assignment of terms [53, 71, 74, 117]. Other approaches, instead, are based on the distribution of the merging phase, freeing the broker from this duty. This is the case of the method called pipelining [75, 54].

Pipeline Strategy permits to distribute the merging phase contacting one after the other, the query servers involved. Each query server is in charge of computing the partial results taking into account the results computed from the previous servers. The last query server sends to the broker the final list of results. Adopting this method, the broker has only to create a query bundle containing the query, the routing information and a set of empty accumulators. Figure 2.3 illustrates the routing of a query through a pipelined system. In this example, the query contains four terms: t_1, t_2, t_3 and t_4. Since the index is term-partitioned, we do not have to contact all the servers, but only a subset of them.

2.3 Our framework

In Chapters 5 and 6 we assume a distributed search engine where data are distributed according to a document partitioning strategy. The index is thus partitioned into shards each one relative to a particular partition of the documents. To increase query throughput, each index shard is typically replicated into several replicas and a query received by the search front-end is routed to one of the available replicas. Our architecture works over a multi-node search engine without replicas, because

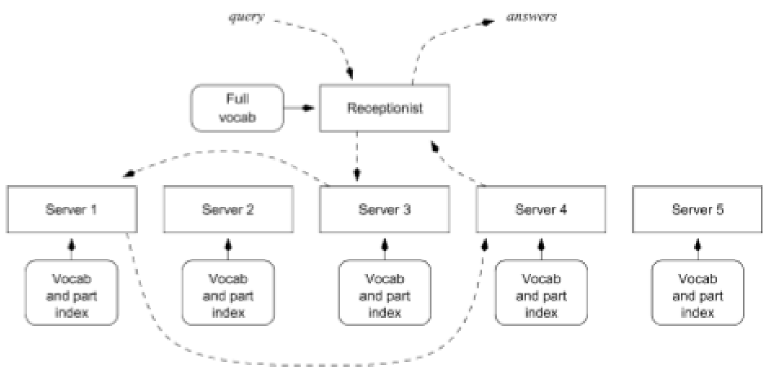

Figure 2.3: Routing of a query through a pipelined system.[75].

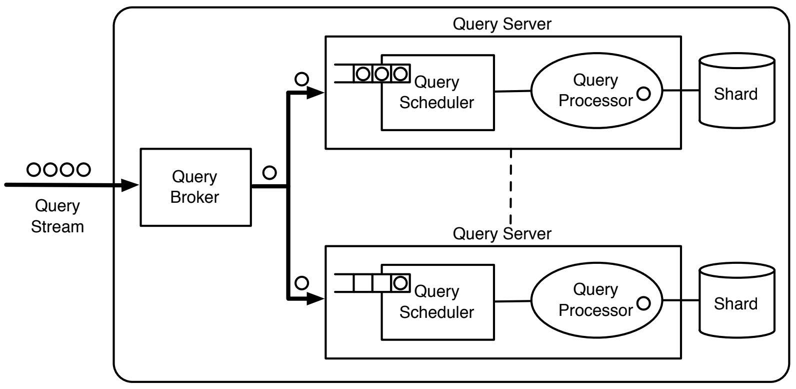

Figure 2.4: Our reference architecture of a distributed search engine node (based on [[34]])

our experimental results are independent from the number of replicas, and hence can be applied directly to each replica independently [34]. Figure 2.4 depicts our reference architecture for a single replica.

New queries arrive at a front-end machine called *query broker*, which broadcasts the query to the query servers of all shards, before collecting and merging the final results set for presentation to the user. When a query reaches a query server, it is processed immediately if the server is idle. Indeed, each query server comprises a *query processor*, which is responsible for tokenising the query and ranking the documents of its index shard according to a scoring function (in our case we use the

BM25 scoring function [88]). Strategies such as dynamic pruning [26, 77, 110] can be used to process queries in an efficient manner on each query server.

In this thesis, we consider document-sorted indices, as used by at least one major search engine [40]. Other efficient retrieval techniques such as frequency-sorted [110] or impact-sorted indices [4] are possible, which also support our objective of early termination of long running queries. However, there is no evidence of such index layouts in common use within commercial search engines [76], perhaps – as suggested by Lester *et al.* [63] – due their practical disadvantages such as difficulty of use for Boolean and phrasal queries. As such, in this work, we focus on the realistic scenario of standard document-sorted index layouts. Finally, we use disjunctive semantics for queries, as supported by Craswell *et al.* [37] who highlighted that disjunctive semantics does not produce significantly different high-precision effectiveness compared to conjunctive retrieval.

On the other hand, if the query server is already busy processing another query, each newly arrived query is placed in a *queue*, waiting to be selected by a *query scheduler* for processing. Hence, the time that a query spends with a query server, i.e. its *completion time*, can be split into two components: a *waiting time*, spent in the queue, and a *processing time*, spent being processed. While the latter depends on the particular retrieval strategy (which we call the processing strategy) and the shard's characteristics, the former depends on the specific scheduling algorithm implemented to manage the queue and on the number of queries in the queue itself.

Indeed, it has been observed that a query scheduler can make some gains in overall efficiency by re-ordering queries, thereby delaying the execution of expensive queries [69]. However, this approach only considers the cost of executing single queries, and hence cannot respond to surges in query traffic. Instead, in this work, we take a different approach, by arguing that the time available to execute a query on a query server – whilst meeting the completion time constraints – is influenced by the other queries queued on that query server. Hence in the work we propose in Chapters 5 and 6, we estimate the target completion times for a query on a server based on the prediction of queueing and completion times for the queries scheduled after the query in the queue.

The utility of query scheduling is particularly evident when queries arrive at a higher rate than the maximum sustainable peak load of the system [55]. Indeed, in our proposed framework, we set the maximum query processing time to a carefully chosen value (see Section 5.1), such that the system load is kept under control, thereby enabling an optimal management of the peak load at the cost of a slightly reduced results quality (see Section 5.4.2). Our proposed framework exploits novel machine learning models for estimating processing time under different processing strategies.

3

On Query Logs

Commercial SEs receive a huge amount of queries every day from millions of users searching for a wide variety of arguments. The behaviour of users, when they interact with the SE, is an important source of information to understand what users are interested in. All user queries and adjacent information is stored by the SEs in a log, commonly called `query log`. However, understanding the intentions of users from the collected data is not a trivial problem. Due to the importance of the information stored in the query log a big effort has been spent during time to extract knowledge from it. All these techniques are known as query log mining techniques, analysed in more detail in Section 3.1. The observation derived from the study of query log further justify our work described in Chapters 5 and 6, which exploit its characteristic for improving efficiency of SEs.

In this thesis we use the information contained in the query log to build a query recommender system in order to assist user to make better queries to satisfy his information need. An overview on query recommender system in general is presented in Section 3.2 while our query recommender application is described in Chapter 4. Furthermore, in Chapters 5 and 6 we use the information stored into the query log to build an efficiency predictor to estimate the query processing time for a set of retrieval strategies.

3.1 Query Log Mining

The general structure of a query log permits not only to extract queries submitted by users but also contains other useful information. In Table 3.1 we can see a sample of the AOL query log [81]. Besides queries, the query log contains additional information such as the user id of who submitted the query, the result he clicked, its position in the ranked list and the time of all these operations. In Table 3.1 some queries submitted from the same user, over a period of five days, are presented. It is possible to observe that the activity of the user 507 can be split in sessions where the user searched for different topics.

Extracting search sessions is an important operation performed on a query log. Search sessions are generally defined as sequences of queries, related to the same subject, submitted by the same user. They offer useful information on understanding how users refine and modify their search to find the answer they are looking for.

Understanding how to best split the query stream to infer sessions is a difficult task especially because we need to understand the topic of the queries. As a matter of fact, the same user often search multiple topics at the same time (*multitasking*), or temporarily change topic to later return to the previous one (*task switching*). As reported in [80] in the 11.4% of the cases users are pursuing multitasking sessions. This percentage raises up to 31.8% in the case of *AllTheWeb.com* users, a popular search engine at that time. In the same paper, the mean number of topic changes per session has been estimated to around 2.2 and when considering only multi-topic sessions it raises up to 3.2.

Due to these reasons, it is difficult to discover groups of coherent queries that are sent for a common purpose (*logical session*). Instead, a common technique is to extract *physical sessions*, namely groups of queries submitted by the same user in a limited period of time, where "limited" depends from the context. In the field of query log mining a physical session is around 30 minutes, but in works like [64, 66], the authors use a more complicated model over the AOL query log and

user ID	query	date	rank	url
507	kbb.com	2006-03-01 16:45:19	1	http://www.kbb.com
507	kbb.com	2006-03-01 16:55:46	1	http://www.kbb.com
507	autotrader	2006-03-02 14:48:05		
507	ebay	2006-03-05 10:50:35		
507	ebay	2006-03-05 10:50:52		
507	ebay	2006-03-05 10:51:24		
507	ebay	2006-03-05 10:52:04		
507	ebay	2006-03-05 10:52:36		
507	ebay	2006-03-05 10:58:00	69	http://antiques.ebay.com
507	ebay	2006-03-05 10:58:21		
507	ebay electronics	2006-03-05 10:59:26		
507	ebay electronics	2006-03-05 11:00:21	5	http://www.internetretailer.com
507	ebay electronics	2006-03-05 11:00:21	20	http://www.amazon.com
507	ebay electronics	2006-03-05 11:00:21	22	http://gizmodo.com
507	ebay electronics	2006-03-05 11:18:56	22	http://gizmodo.com
507	ebay electronics	2006-03-05 11:20:59		
507	ebay electronics	2006-03-05 11:21:53	66	http://portals.ebay.com
507	ebay electronics	2006-03-05 11:25:35		

Table 3.1: Some rows of the AOL query log [98]

they understand that the mean length of a session for that query log is 26 minutes.

During a search session a user often tries to refine or modify queries in order to satisfy his information need. This behaviour is studied by Lau and Horvitz in [60] where they divide the session's queries in different types:

- **new:** a query for a topic not previously searched for in the current session;
- **generalization:** a query on the same topic as the previous query, but seeking more general information;

- **specialization:** a query on the same topic as the previous query, but seeking more specific information;
- **reformulation:** a query on the same topic that can be viewed as neither a generalization nor a specialization, but a reformulation of the prior query.
- **interruption:** a query on a topic previously searched by a user but which has been interrupted by a search on another topic;
- **request for additional results:** a request for another set of results on the same query from the search service. Duplicate queries appear in the data when a person requests another set of results for the query;
- **blank queries:** log entries containing no query.

Understanding users behaviour during their sessions is important in order to discover patterns and improve the user experience increasing also the performance of the SE.

Another frequently used information is the *click-through*, namely the result selected by the user. To understand if a user has found something interesting for the query he submitted, we can exploit the links he eventually clicked within the results page. As we see in Chapter 4 and in [24], the click-through information can be used to discriminate between successful or not successful sessions, respectively those that end up with a clicked query or not.

To be able to mine useful information from the query log, a common technique is to prepare the data before using it. The reason is that the query log is often target of spam and user errors that can deteriorate the outcome of the mining process. This phase is called *data preparation* and comprises data cleaning (removing irrelevant item as, for example, the robot queries), anonymization (seeking to identify sessions and to remove sensible data to avoid privacy issues) and so on. All these data preparation techniques do not have to remove important data or change important statistic features.

A lot of effort was spent to study what the users search through SEs. One of the first important intuitions is reported in [97] published in 1999 where the authors studied 45 days of the Altervista query log. They showed that about 85% of the users visit the first page of results only. They also show that 77% of the user sessions end up just after the first query while only the 4.5% of them are longer than 3 queries. Another interesting contribution of the same paper is about the structure of the user sessions: in more than half of the cases some query terms are deleted from the query and other terms added. It is likely the user is modifying the query to restate the information need. Instead in about 12% of cases terms are either added or deleted, in this case the user is probably trying to restrict the search or make it broader. In more than 35% of cases the query change totally.

A good summary of interesting features found in query logs is reported in [61, 42, 72]. The first important thing to consider is the distribution of the query frequency. The distribution in fact follows a *power law*, meaning that the occurrences y of a query is proportional to its popularity rank x following the formula $y = Kx^{-\alpha}$. The parameters K and α are respectively a normalization constant and a parameter that shows how popularity decreases. In particular in [72, 61, 7] it is shown that the exponent is around 2.4 for different query logs: AltaVista, Yahoo! and Excite. As an example in Figure 3.1 we can see the number of accesses for the first $1,000$ most popular queries in Excite query log. The most popular query is submitted $2,219$ times, while the $1,000$th most popular query is submitted only 27 times.

Some other interesting query log statistics are presented in [81]. The *query arrival rate*, for example, is an interesting parameter useful both for developing the SE architecture and for understanding the user behaviour. In figure 3.1 we see that throughout an entire day, the query traffic is not constant. In the morning, users submit less queries than in the afternoon and, at the same time, there is correlation between topics searched by users and the time of the search (Figure 3.1).

Discovering topics is another non-trivial task in query log mining. A very first result in categorizing queries is [104]. Authors show the percentage of queries sub-

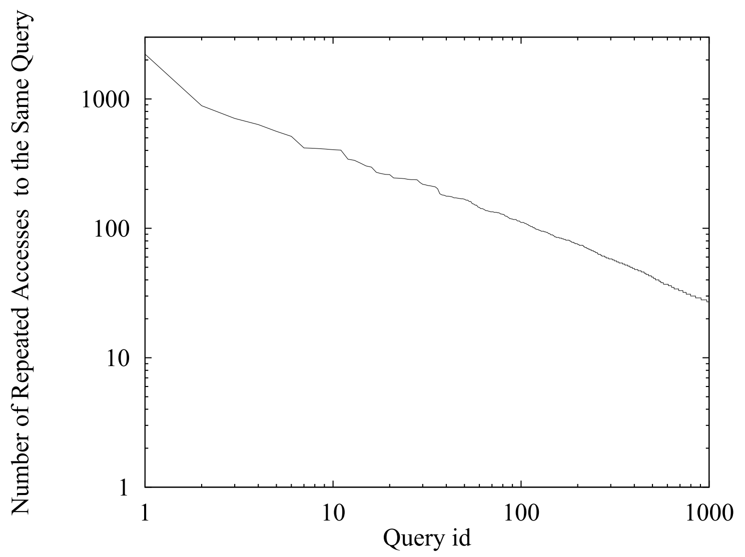

Figure 3.1: Number of accesses for the 1,000 most popular queries on Excite [72]

mitted for each topic to the Excite search engine in 1997. Recent papers showing techniques for assigning labels to each query [15, 27] adopts a set of multiple classifiers subsequently refining the classification phase.

3.2 Query Recommender System

Recommender systems are used in several domains, being specially successful in electronic commerce. They can be divided in two broad classes: those based on *content filtering*, and those on *collaborative filtering*. As the name suggests, content filtering approaches base their recommendations on the content of the items to be suggested. They face serious limitations when dealing with multimedia content and, more importantly, their suggestions are not influenced by the human-perceived *quality* of contents. On the other side, collaborative filtering solutions are based on the preferences of other users. There are two main families of collaborative filtering algorithms: memory-based and model-based. Memory-based approaches use the whole past usage data to identify similar users [93], items [92], or both [112]. Generally,

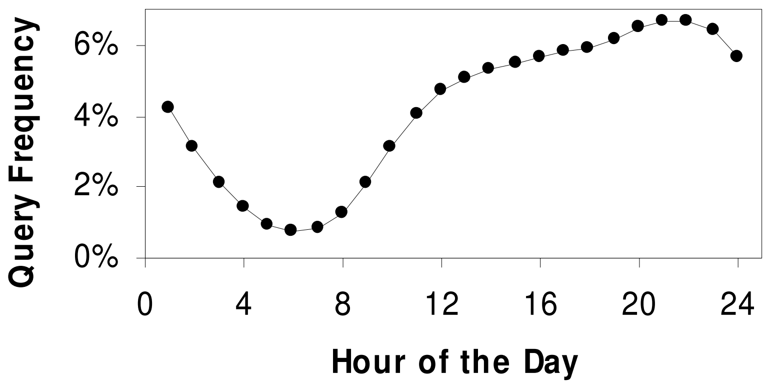

Figure 3.2: Number of accesses for the 1,000 most popular queries on Excite [81]

memory-based algorithms are quite simple and produce good recommendations, but they usually face serious scalability problems. On the other hand, model-based algorithms construct in advance a model to represent the behavior of users, allowing to predict more efficiently their preferences. However, the model building phase can be highly time-consuming, and models are generally hard to tune, sensitive to data changes, and highly dependent on the application domain. Different approaches can be adopted based on linear algebra [35, 85], clustering [111], latent class models [51], singular value decomposition [82]. An analysis of the use of collaborative filtering algorithms to the query suggestion problem can be found in [12], where the problem descending from the poor and very sparse scoring information available in query logs is highlighted.

On the other side, query suggestion techniques address specifically the problem of recommending queries to Web search engine users, and propose specific solutions and specific evaluation metrics tailored to the Web search domain. Techniques proposed during last years are very different, yet they have in common the exploitation of usage information recorded in query logs [99]. Many approaches extract the information used from the plain set of queries recorded in the log, although there are several works that take into account the chains of queries that belong to the same search session [84]. In the first category we have techniques that employ clus-

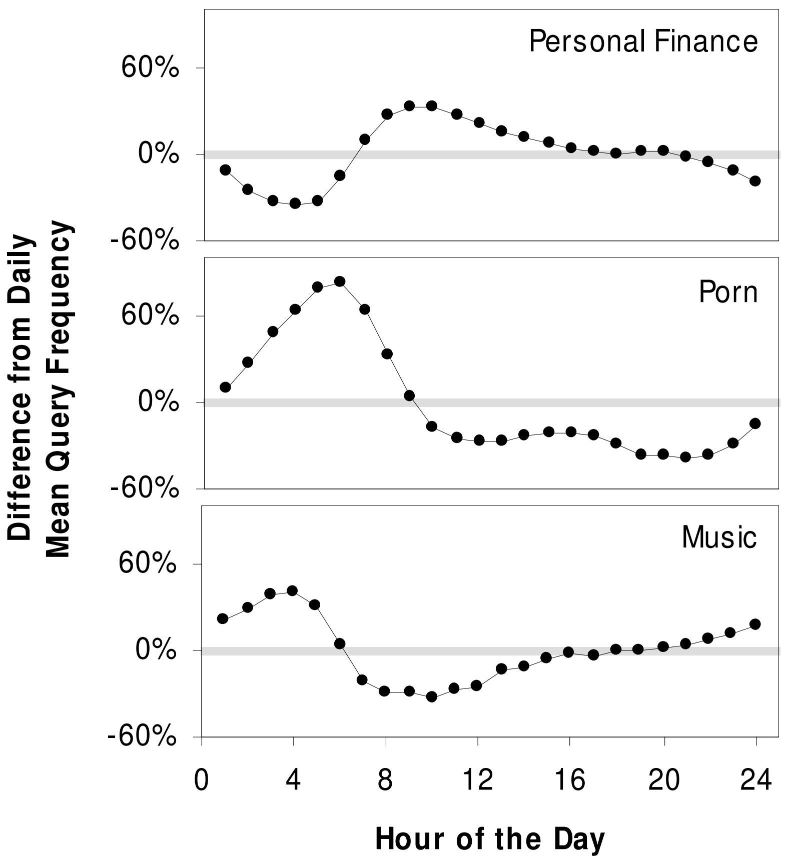

Figure 3.3: Number of accesses for the 1,000 most popular queries on Excite [81]

tering algorithms to determine groups of related queries that lead users to similar documents [115, 8, 14]. The most "representative" queries in the clusters are then returned as suggestions. Others solutions employ the reformulations of the submitted query issued by previous users [58], or propose as suggestions the frequent queries that lead in the past users to retrieve similar results [11].

[10] exploit click-through data as a way to provide recommendations. The method is based on the concept of *Cover Graph*. A Cover Graph is a bipartite graph of queries and URLs, where a query q and an URL u are connected if a user issued q and clicked on u that was an answer for the query. Suggestions for a query q are thus obtained by accessing the corresponding node in the Cover Graph and by extracting the related queries sharing more URLs. The sharing of clicked URLs

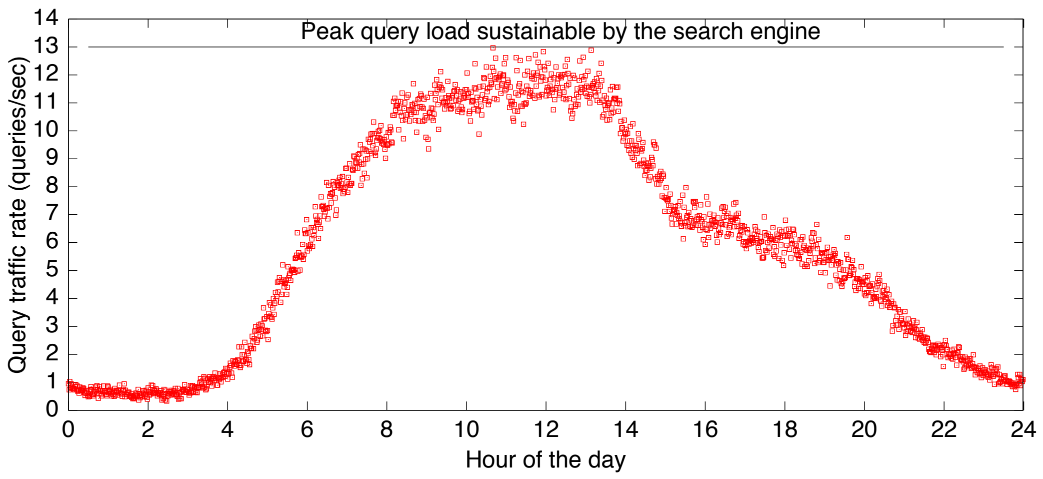

Figure 3.4: The query distribution of a 24 hours time span covering 1st May 2006 from MSN query log.

results to be very effective for devising related queries, and the Cover Graph solution has been chosen as one of the two query suggestion algorithms considered in the Chapter 4 for experimental performance comparison.

Among the proposals exploiting the chains of queries stored in query logs, [43] use an association rule mining algorithm to devise frequent query patterns. These patterns are inserted in a query relation graph which allows "concepts" (queries that are synonyms, specializations, generalizations, etc.) to be identified and suggested.

Boldi *et al.* introduce the concept of *Query Flow Graph*, an aggregated representation of the information contained in a query log [16]. A Query Flow Graph is a directed graph in which nodes are queries, and the edge connecting node q_1 to q_2 is weighted by the probability that users issue query q_2 after issuing q_1. Authors highlight the utility of the model in two concrete applications, namely, *devising logical sessions* and *generating query recommendation*. The authors refine the previous studies in [17] and [18] where a query suggestion scheme based on a random walk with restart model on the Query Flow Graph is proposed. Such suggestion algorithm is the second algorithm considered in the Chapter 4 for experimental performance comparison.

Another approach is represented by the query refinement/substitution technique discussed in [58]. The goal of query refinement is to generate a new query to replace a user's original ill-formed search query in order to enhance the relevance of retrieved results. The technique proposed includes a number of tasks such as spelling error correction, word splitting, word merging, phrase segmentation, word stemming, and acronym expansion.

The importance of rare query classification and suggestion recently attracted a lot of attention from the information retrieval community. Generating suggestions for rare queries is in fact very difficult due to the lack of information in the query logs.

The authors of [41] describe search log studies aiming at explaining behaviours associated with rare and common queries. They investigate the search behaviour following the input of rare and common queries. Results show that search engines perform less well on rare queries. The authors also study transitions between rare and common queries highlighting the difference between the frequency of queries and their related information needs.

[103] propose an optimal rare query suggestion framework by leveraging implicit feedbacks from users in the query logs. The proposed model is based on the pseudo-relevance feedback. It assumes that clicked and skipped URLs contain different level of information, and thus, they should be treated differently. Therefore, the framework optimally combines both click and skip information from users, and uses a random walk model to optimize i) the restarting rate of the random walk, and ii) the combination ratio of click and skip information. Experimental results on a log from a commercial search engine show the superiority of the proposed method over the traditional random walk models and pseudo-relevance feedback models.

Mei *et al.* propose a novel query suggestion algorithm based on ranking queries with the hitting time on a large scale bipartite graph [73]. The rationale of the method is to capture semantic consistency between the suggested queries and the original query. Empirical results on a query log from a real world search engine show that hitting time is effective to generate semantically consistent query suggestions. The authors show that the proposed method and its variations are able to boost long tail queries, and personalized query suggestion.

Broder *et al.* propose to leverage the results from search engines as an external knowledge base for building the word features for rare queries [28]. The authors train a classifier on a commercial taxonomy consisting of 6,000 nodes for categorization. Results show a significant boost in term of precision with respect to the baseline query expansion methods. Lately, Broder *et al.* propose an efficient and effective approach for matching ads against rare queries [25]. The approach builds an expanded query representation by leveraging offline processing done for related popular queries. Experimental results show that the proposed technique significantly improves the effectiveness of advertising on rare queries with only a negligible increase in computational cost.

4

SearchShortcut

In this Chapter we propose an effcient and effective solution to the problem of choosing the queries to suggest to web search engine users in order to help them in rapidly satisfying their information needs. By exploiting a weak function for assessing the similarity between the current query and the knowledge base built from historical user sessions, we re-conduct the suggestion generation phase to the processing of a full-text query over an inverted index. The resulting query recommendation technique is very effcient and scalable, and is less affected by the data-sparsity problem than most state-of-the-art proposals described in 3.2. Thus, it is particularly effective in generating suggestions for rare queries occurring in the long tail of the query popularity distribution. The quality of suggestions generated is assessed by evaluating the effectiveness in forecasting the users' behaviour recorded in historical query logs, and on the basis of the results of a reproducible user study conducted on publicly-available, human-assessed data. The experimental evaluation conducted shows that our proposal remarkably outperforms two other state-of-the-art solutions, and that it can generate useful suggestions even for rare and never seen queries.

4.1 Introduction

Giving suggestions to users of Web search engines is a common practice aimed at *driving* users toward the information bits they may need. Suggestions are normally provided as queries that are, to some extent, related to those recently submitted by the user. The generation process of such queries, basically, exploits the expertise of "skilled" users to help inexperienced ones. The knowledge mined for making this possible is contained in Web search engine logs which store all the past interactions of users with the search system. More the users that satisfied the same information need in the past, the more precise and effective the related suggestions provided by any query recommendation technique. On the other hand, to generate effective suggestions for user queries which are rare or have never been seen in the past is an open issue poorly addressed by state-of-the-art query suggestion techniques.

In a previous work, an interesting framework for the query suggestion problem is provided by the *Search Shortcut* model, and an evaluation metric for assessing the effectiveness of suggested queries by exploiting a query log is proposed [12]. Basically, the model formalizes a way of exploiting the knowledge mined from query logs to help users to *rapidly* satisfy their information need. In the same work the use of *Collaborative Filtering* (CF) algorithms is investigated. However, the work highlights some limitations in the query recommendations solutions based on collaborative filtering mainly due to the poor and very sparse scoring information available in query logs. In fact, due to the long-tail distribution of query occurrences, click information for low-frequency queries is rare and very sparse. Since implicit feedback information given by popularity and user clicks is the only source of (positive) query scoring available, most of the queries in the query log cannot be exploited to generate the recommendation model [1]. This issue affects CF-based solutions, but also many other query recommendation techniques discussed in the literature.

In this Chapter we propose an efficient and effective query recommendation algorithm that can *cover* also queries in the long tail. We adopt the Search Shortcuts model and its terminology, and re-conduct the shortcut generation phase to the processing of a full-text query over an inverted file that indexes satisfactory user sessions recorded in a query log. Unlike most state-of-the art proposals, our shortcut generation algorithm aggregates implicit feedback information present in query logs at the level of single query terms, thus alleviating the data sparseness issue. The contribution of each query terms is then combined during the suggestion generation process in order to provide recommendations also for queries that are rare or even for those that were never seen in the past. Generating suggestions for rare queries is a hot research topic [73, 103, 25], and our suggestion generation technique beyond addressing the data-sparsity problem, is both very efficient and scalable, making it suitable for a large-scale deployment in real-world search engines.

Another contribution of this Chapter consists in the methodology adopted for manually assessing the effectiveness of query suggestion techniques. The methodology exploits the query topics and the human judgements provided by the National Institute of Standards and Technology (NIST) for running the TREC Web Track's diversity task. For the purposes of the diversity task, the NIST assessors provide 50 queries, and, for each of them, they identify a representative set of subtopics, based on information extracted from the logs of a commercial search engine. We claim that *given a query topic A with all its subtopics $\{a_1, a_2, \ldots, a_n\}$, and a query suggestion technique $\mathcal{T}$, the more the queries suggested by $\mathcal{T}$ for A cover the human-assessed subtopics $\{a_1, a_2, \ldots, a_n\}$, the more $\mathcal{T}$ is effective.* To assess the effectiveness of a given query suggestion technique, we thus propose to simply ask human editors to count how many subtopics are actually covered by the suggestions generated by $\mathcal{T}$ for the TREC diversity track queries. This methodology is entirely based on a publicly-available data. It can be thus considered fair and constitute a good shared base for testing and comparing query recommendation systems. We shall define the above concept better in Section 4.3.

The experimental evaluation conducted shows that the proposed solution outperforms remarkably two state-of-the-art algorithms chosen for performance comparison purposes (presented in [10] and [16, 17]). Unlike these competitor algorithms, our solution generates in fact relevant suggestions for a vast majority of the 50 TREC queries, and the suggested queries cover a high percentage of possible subtopics. In particular, we assessed that it can generate useful suggestions even for queries that are rare or do not occur in the query log used for training the recommendation model. Moreover, the proposed algorithm outperforms the competitor solutions even on the tests measuring the effectiveness in forecasting the users' behavior recorded in historical query according to the metric used in [12].

The main original contributions of this work are thus:

1. A novel algorithm to efficiently and effectively generate query suggestions that is robust to data sparsity;
2. A novel evaluation methodology with which we can compare the effectiveness of suggestion mechanisms;
3. An extensive evaluation comparing on the same basis the proposed solution with two state-of-the-art algorithms.

The idea we present in this Chapter follows a completely new approach in relation to the algorithma t the state of the art described in Chapter 3.2. First, we infer the relevance of a query based on whether it successfully ended a search session, i.e., the *last query* of the user session allowed the user to find the information she was looking for. Recent research results have shown in fact that user behavior recorded in query log allows effective predictive models to be learned for estimating search success [49]. *Successful sessions* have also already been taken into account as a way to evaluate promotions of search results [102, 101]. Similarly, *satisfactory sessions* are considered in this Chapter as the key factor for generating useful query recommendations. All the queries in the satisfactory sessions stored in the log which terminate with the same final query are considered "related", since it is likely that these queries were issued by different users trying to satisfy a similar information need. Thus, our technique exploit a sort of collaborative clustering of queries inferred from successful user search processes, and suggest users the final queries which are the representatives of the clusters closest to the submitted query.

To deeply understand the work presented in this Chapter is suggested to read the Chapter 3, in particualar the sections about Query Log mining (Section 3.1) and Query Recommender System (Section 3.2).

The rest of the Chapter is organized as follows. The next Section briefly sketches the shortcuts model and describes the efficient algorithm designed for generating query shortcuts. The evaluation methodology based on the TREC diversification track data is discussed in Section 4.3 which also presents the encouraging results obtained by our solution in the performance comparisons tests conducted. Finally, Section 4.4 draws some conclusions.

4.2 An Efficient Algorithm for the Query Shortcuts Problem

In the following we briefly recall the basis of the *Search Shortcuts Problem* (SSP) proposed in [12], and we introduce our novel shortcuts generation algorithm.

4.2.1 The Search Shortcuts Problem

The SSP is formally defined as a problem related to the recommendation of queries in search engines and the potential reductions obtained in the users session length. This problem formulation allows a precise goal for query suggestion to be devised: *recommend queries that allowed "similar" users, i.e., users which in the past followed a similar search process, to successfully find the information they were looking for.* The problem has a nice parallel in computer systems: *prefetching.* Similarly to prefetching, search shortcuts anticipate requests to the search engine with suggestion of queries that a user would have likely issued at the end of her session.

We now introduce the notations and we recap the formal definition of the SSP.

Let $\mathcal{U}$ be the set of users of a Web search engine whose activities are recorded in a query log, and $\mathcal{Q}$ be the set of queries in that query log. We suppose the query log is preprocessed by using some session splitting method (e.g. [57], [65]) in order to extract query *sessions*, i.e., sequences of queries which are related to the same user search task. Formally, we denote by $\mathcal{S}$ the set of all sessions in a query log. Moreover, let us denote with σ_i the i-th query of σ. For a session σ of length n its **final query** is the query σ_n, i.e., the last query issued by the user in the session.

We say that a session σ is ***satisfactory*** if and only if the user has clicked on at least one link shown in the result page returned by the Web search engine for the final query σ_n, ***unsatisfactory*** otherwise. Finally, given a session σ of length n we denote $\sigma_{t|}$ the **head** of σ, i.e., the sequence of the first t, $t < n$, queries, and $\sigma_{|t}$ the **tail** of σ given by the sequence of the remaining $n - t$ queries.

Definition 1 *We define **k-way shortcut** a function h taking as argument the head of a session $\sigma_{t|}$, and returning as result a set $h\left(\sigma_{t|}\right)$ of k queries belonging to Q.*

Such definition allows a simple ex-post evaluation methodology to be introduced by means of the following similarity function:

Definition 2 *Given a satisfactory session $\sigma \in \mathcal{S}$ of length n, and a k-way shortcut function h, the similarity between $h\left(\sigma_{t|}\right)$ and a tail $\sigma_{|t}$ is defined as:*

$$s\left(h\left(\sigma_{t|}\right), \sigma_{|t}\right) = \frac{\sum\limits_{q \in h(\sigma_{t|})} \sum\limits_{m=1}^{n-t} [\![q = \left(\sigma_{|t}\right)_m]\!] f\left(m\right)}{|h(\sigma_{t|})|} \tag{4.1}$$

where $f\left(m\right)$ is a monotonic increasing function, and function $[\![q = \sigma_m]\!] = 1$ if and only if q is equal to σ_m.

For example, to evaluate the effectiveness of a given shortcut function h, the sum (or average) of the value of s computed on all satisfactory sessions in $\mathcal{S}$ can be computed.

Definition 3 *Given the set of all possible shortcut functions $\mathcal{H}$, we define **Search Shortcut Problem** (SSP) the problem of finding a function $h \in \mathcal{H}$ which maximizes the sum of the values computed by Equation (4.1) on all satisfactory sessions in $\mathcal{S}$.*

A difference between search shortcuts and query suggestion is actually represented by the function $[\![q = \left(\sigma_{|t}\right)_m]\!]$ in Equation (4.1). By relaxing the strict *equality* requirement, and by replacing it with a similarity relation – i.e., $[\![q \sim \left(\sigma_{|t}\right)_m]\!] = 1$ if and only if the similarity between q and σ_m is greater than some threshold – the problem reduces, basically, to query suggestion. By defining appropriate similarity functions, the Equation in (4.1) can be thus used to evaluate query suggestion effectiveness as well.

Finally, we should consider the influence the function $f(m)$ has in the definition of scoring functions. Actually, depending on how f is chosen, different features of a shortcut generating algorithm may be tested. For instance, by setting $f(m)$ to be the constant function $f(m) = c$, we measure simply the number of queries in common between the query shortcut set and the queries submitted by the user. A non-constant function can be used to give an higher score to queries that a user would have submitted later in the session. An exponential function $f(m) = e^m$ can be exploited instead to assign an higher score to shortcuts suggested early. Smoother f functions can be used to modulate positional effects.

4.2.2 The Search Shortcuts Generation Method

Inspired by the above SSP, we define a novel algorithm that aims to generate suggestions containing *only* those queries appearing as final in satisfactory sessions. The goal is to suggest queries having a high potentiality of being useful for people to reach their initial goal. As hinted by the problem definition, suggesting queries appearing as final in satisfactory sessions, in our view is a good strategy to accomplish this task. In order to validate this hypothesis, we analyzed the Microsoft RFP 2006 dataset, a query log from the MSN Search engine containing about 15 million queries sampled over one month of 2006.

First, we measured that the number of distinct queries that appear as final query in satisfactory sessions of the query log is relatively small if compared to the overall number of submitted queries: only about 10% of the total number of distinct queries in the query log occur in the last position of satisfactory user sessions. As expected, the distribution of the occurrences of such final queries in satisfactory user sessions is very skewed (as shown in Figure 4.1), thus confirming once more that the set of final queries *actually* used by people is limited.

Queries which are *final* in some satisfactory sessions may obviously appear also in positions different from the last in other satisfactory sessions. We verified that, when this happens, these queries appear much more frequently in positions very close to the final one. About 60% of the distinct queries appearing in the penultimate position of satisfactory sessions are also among the final queries, about 40% in positions second to the last, 20% as third to the last, and so on. We can thus argue that *final* queries are usually *close* to the achievement of the user information goal. We consider these queries as highly valued and high quality short pieces of text expressing actual user needs.

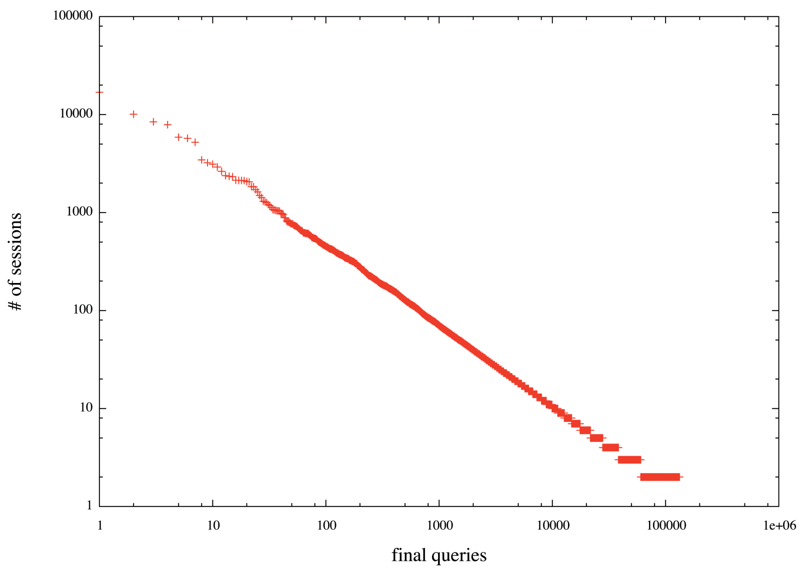

Figure 4.1: Popularity of final queries in satisfactory sessions.

The SSP algorithm proposed in this Chapter works by computing, efficiently, similarities between partial user sessions (the one currently performed) and historical satisfactory sessions recorded in a query log. Final queries of most similar satisfactory sessions are suggested to users as search shortcuts.

Let σ' be the current session performed by the user, and let us consider the sequence τ of the concatenation of all terms with possible repetitions appearing in $\sigma'_{t|}$, i.e., the head of length t of session σ'. We now compute the value of a scoring function $\delta\left(\tau, \sigma^s\right)$, which for each satisfactory session σ^s measures the similarity between its queries and the set of terms τ. Intuitively, this similarity measures how much a previously seen session overlaps with the user need expressed so far (the concatenation of terms τ serves as a bag-of-words model of user need). Sessions are ranked according to δ scores and from the subset of the top ranked sessions we suggest their final queries. It is obvious that depending on how the function δ is chosen we may have different recommendation methods. We opt for δ to be a linear combination of the BM25 metric [89] and the frequency of the final queries in the query log. More formally, given a satisfactory session σ^s of length n, its final query σ^s_n, and the sequence τ of the concatenation of all terms with possible repetitions appearing in $\sigma'_{t|}$, we define

$$\delta(\tau, \sigma^s) = \alpha \cdot BM25(\tau, \sigma^s) + \beta \cdot freq(\sigma^s_n)$$

We exploit an IR-like metric (BM25) because we want to take into much consideration words that are discriminant in the context of the session to which we are comparing. BM25, and other IR-related metrics, have been designed specifically to account for that property in the context of query/documents similarity. We borrow from BM25 the same attitude to adapt to this conditions. We also exploit the frequency of final queries in the scoring formula. By doing so we aim at promoting sessions containing final queries that are frequently used by users. The shortcuts generation problem has been, thus, reduced to the information retrieval task of finding highly similar sessions in response to a given sequence of queries. In our experiments we set $\alpha = \beta = 1/2$. Furthermore, we compute the similarity function δ only on the current query issued by the user instead of using the whole head of the session. We do this in order to be fair with respect to our competitors as they produce recommendations starting from a single query. We leave the study of the use of the whole head of the session for producing query recommendations as a future work.

The idea described above is thus translated into the following process (see Algorithm 1). For each unique *final query* contained in satisfactory sessions we define what we have called a *virtual document* identified by its *title* and its *content*. The title, i.e., the identifier of the document, is exactly the string of the final query. The content of the virtual document is instead composed of all the terms that have appeared in queries of all the satisfactory sessions ending with the final query. At the end of this procedure we have a set of virtual documents, one for each distinct final query occurring in some satisfactory sessions. Just to make things more clear, let us consider a toy example. Consider the two following satisfactory sessions: (*gambling*, *gambling places*, *las vegas*, *bellagio*), and (*las vegas*, *strip*, *las vegas hotels*, *bellagio*). We create the virtual document identified by the title **bellagio** and whose content is the text (*gambling gambling places las vegas las vegas strip las vegas hotels*). As you can see the virtual document actually contains also repetitions of the same term that are considered in the context of the BM25 metrics.

All virtual documents are indexed with the preferred Information Retrieval system, and generating shortcuts for a given user session σ' becomes simply processing the query $\sigma'_{t|}$ over the inverted file indexing such virtual documents (see Algorithm 2). We know that processing queries over inverted indexes is very fast and scalable, and these important characteristics are inherited by our query suggestion technique as well.

The other important feature of our query suggestion technique is its robustness with respect to rare and singleton queries. Singleton queries account for almost 50% of the submitted queries [99], and their presence causes the issue of the sparsity of models [1]. Since we match τ with the text obtained by concatenating all the

Algorithm 1 Offline generation of the recommendation model.

```
Require: a set $S$ of user sessions recorded in the query log.
Ensure: an inverted index $vd_{index}$ built over the virtual documents obtained from
    satisfactory sessions.
1: for all $\sigma \in S$ do
2:   if $sessionIsSatisfactory(\sigma)$ then
3:     $\tau \leftarrow getTermsFromSession(\sigma)$
4:     $vd[\sigma_n] \leftarrow \tau$ {given current satisfactory session $\sigma$, the occurrences of all the
       query-terms in $\sigma$ are added to the (initially empty) body of the virtual
       document associated to final query $\sigma_n$.}
5:   end if
6: end for
7: $vd_{index} \leftarrow buildInvertedIndex(vd)$ {we build the inverted file indexing all the
   obtained virtual documents.}
```

queries in each session, we are not bound to look for previously submitted queries as in the case of other suggestion algorithms. Therefore we can generate suggestions for queries in the long tail of the distribution whose terms have some context in the query log used to build the model.

4.3 Assessing Search Shortcuts Quality

The effectiveness of a query recommender systems can be evaluated by means of *user-studies* or through the adoption of some performance metrics. Unfortunately, both these methodologies may lack of generality and incur in the risk of being over-fitted on the system object of the evaluation. The evaluation methodology used in this Chapter tries to address pragmatically the above issues.

For what concerns the methodology based on a performance metrics, we used the one defined in Equation (4.1), and we computed the average value of similarity

Algorithm 2 Suggestion retrieval.

```
Require: the head $\sigma'_{t|}$ of length $t$ of the current user session $\sigma'$, and the recommendation model $vd_{index}$.
Ensure: the set $R$ of top-$k$ scored recommendations for the given query.
1: $\tau \leftarrow getTermsFromSession(\sigma'_{t|})$
2: $D \leftarrow getMatchingVirtualDocuments(vd_{index}, \tau)$
3: $R \leftarrow$ new $heap(k)$
4: for all $d \in D$ do
5:     $shortcut \leftarrow getTitle(d)$
6:     $R.insert(shortcut, \alpha \cdot BM25(\tau, d) + \beta \cdot freq(shortcut))$
7: end for
8: return $R$.
```

over a set of satisfactory sessions. This performance index *objectively* measures the effectiveness of a query suggestion algorithm in foreseeing the satisfactory query for the session.

In particular, we measured the values of this performance index over suggestions generated by using our *Search Shortcuts* (SS) solution and by using in exactly the same conditions two other state-of-the-art algorithms: *Cover Graph* (CG) proposed by [10], and *Query Flow Graph* (QFG), proposed by [17]. These algorithms are recent and highly reputed representatives of the best practice in the field of query recommendation. To test QFG-based query suggestion we used the original implementation kindly provided us by the authors. In the case of CG, instead, we evaluate our own implementation of the technique.

For what concerns the methodology based on user-studies, we propose an approach that measures *coverage* and the *effectiveness* of suggestions against a manually assessed and publicly available dataset.

To this purpose, we exploited the query topics and the human judgements provided by NIST for running the TREC 2009 Web Track's Diversity Task (`http:`

`//trec.nist.gov/data/web09.html`). For the purposes of the TREC diversity track, NIST provided 50 queries to a group of human assessors. Assuming each TREC query as a topic, assessors were asked to identify a representative set of subtopics covering the whole spectrum of different user needs/intentions. Subtopics are based on information extracted from the logs of a commercial search engine, and are roughly balanced in terms of popularity. Obviously the queries chosen are very different and from different categories: difficult, ambiguous, and/or faceted in order to allow the overall performance of diversification methods to be evaluated and compared. Since diversity and topic coverage are key issues also for the query recommendation task [67], we propose to use the same third-party dataset for evaluating query suggestion effectiveness as well.

Let's now introduce the definitions of *coverage*, and *effectiveness*.

Definition 4 (Coverage) *Given a query topic A with subtopics $\{a_1, a_2, \ldots, a_n\}$, and a query suggestion technique $\mathcal{T}$, we say that $\mathcal{T}$ has coverage equal to c if $n \cdot c$ subtopics match suggestions generated by $\mathcal{T}$.*

In other words, a coverage of 0.8 for the top-10 suggestions generated for a query q having 5 subtopics means that 4 subtopics of q are covered by at least one suggestion.

Definition 5 (Effectiveness) *Given a query topic A with subtopics $\{a_1, a_2, \ldots, a_n\}$, and a query suggestion technique $\mathcal{T}$ generating k suggestions, we say that $\mathcal{T}$ has effectiveness equal to e if $k \cdot e$ suggestions cover at least one subtopic.*

In other words, an effectiveness of 0.1 on the top-10 suggestions generated for a query q means that only one suggestion is relevant for one of the subtopics of q.

The methodology just described has some net advantages. It is based on a publicly-available test collection which is provided by a well reputed third-party organization. Moreover, it grants to all the researchers the possibility of measuring the performance of their solution under exactly the same conditions, with the same dataset and the same reproducible evaluation criterium. In fact, even though the matching between suggestions and topics is still human-driven the process has a very low ambiguity as we shall discuss in the next section.

4.3.1 Experimental Settings

The experiments were conducted using the Microsoft RFP 2006 query log which was preliminary preprocessed by converting all queries to lowercase, and by removing stop-words and punctation/control characters.

The queries in the log were then sorted by user and timestamp, and segmented into sessions on the basis of a splitting algorithm which simply groups in the same session all the queries issued by the same users in a time span of 30 minutes. We tested also the session splitting technique based on the Query Flow Graph proposed in [16], but for the purpose of our technique, we did not observe a significant variation in terms of quality of the generated suggestions.

Noisy sessions, likely performed by software robots, were removed. The remaining entries correspond to approximately 9M sessions. These were split into two subsets: training set with 6M sessions and a test set with the remaining 3M sessions. The training set was used to build the recommendation models needed by CG and QFG and used for performance comparison.

Instead, to implement our SS solution we extracted satisfactory sessions present in the training set and grouped them on the basis of the final query. Then, for each distinct final query its corresponding *virtual document* was built with the terms (with possible repetitions) belonging to all the queries of all the associated satisfactory sessions. Finally, by means of the Terrier search engine (`http://terrier.org/`), we indexed the resulting $1,191,143$ virtual documents. The possibility of processing queries on such index is provided to interested readers through a simple web interface available at the address `http://searchshortcuts.isti.cnr.it`. The web-based wrapper accepts user queries, interact with Terrier to get the list of final queries (id of virtual documents) provided as top-k results, and retrieves and visualizes the

Query: TREC query (n. 8): ***appraisal***

Query's subtopics:

S1: What companies can give an appraisal of my home's value?

S2: I'm looking for companies that appraise jewelry.

S3: Find examples of employee performance appraisals.

S4: I'm looking for web sites that do antique appraisals.

SS	**QFG**	**CG**
performance appraisal (**S3**)	online appraisals (**S4**)	appraisersdotcom (**S4**)
hernando county property appraiser (**S1**)		employee appraisals (**S3**)
		real estate appraisals (**S1**)
antique appraisal (**S4**)		appraisers (**S1**)
appraisers in colorado (**S1**)		employee appraisals
		forms (**S3**)
appraisals etc (**S1**)		appraisers.com (**S4**)
appraisers.com (**S4**)		*gmac*
find appraiser (**S1**)		appraisers
		beverly wv (**S1**)
wachovia bank appraisals (**S1**)		picket fence
		appraisal (**S1**)
appraisersdotcom (**S4**)		*fossillo creek san antonio*

Table 4.1: An example of the coverage evaluating process involving the TREC dataset. For the 8th TREC query ***appraisal***, one of the assessors evaluates the coverage of suggestions generated by SS, QGF, and CG. The subtopics covered by each suggestion are reported in bold between parentheses. Suggestions not covering any of the subtopics are emphasized.

associated query strings.

4.3.2 TREC queries statistics

We measured the popularity of the 50 TREC 2009 Web Track's Diversity Task queries in the training set obtained by the Microsoft RFP 2006 dataset as described in the previous section. Figure 4.2 shows the cumulative frequency distribution of the 50 TREC queries. While 8/50 queries are not present in the training set, 2/50 queries occur only one time. Furthermore, 23/50 queries have a frequency lower than 10 and 33/50 queries occur lower than 100 times. The TREC dataset thus contains a valid set of queries for evaluating the effectiveness of our method as it includes several examples of unseen and rare queries, while popular queries are represented as well. Table 4.2 shows some queries with their popularity measured in the training set.

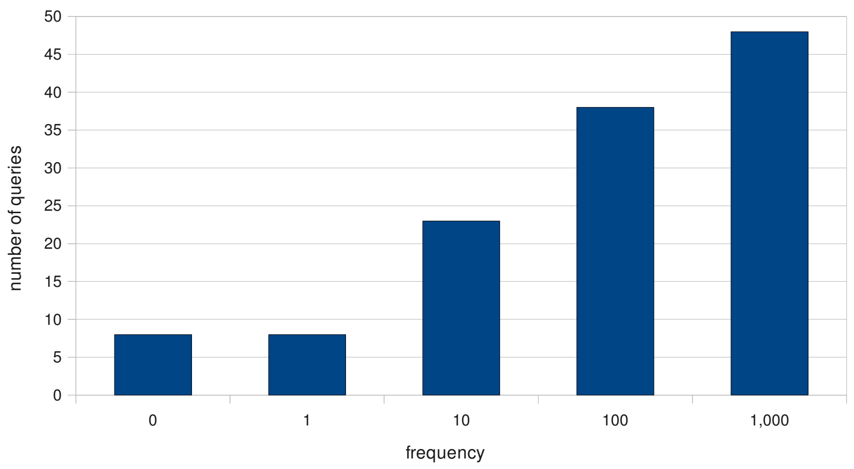

Figure 4.2: Histogram showing the total number of TREC queries (on the y axis) having at most a certain frequency (on the x axis) in the training set. For instance, the third bar shows that 23 TREC queries out of 50 occur at most ten times in the training set.

TREC Query	Frequency
wedding budget calculator	0
flame designs	1
dog heat	2
the music man	5
diversity	27
map of the united states	170
cell phones	568
starbucks	705

Table 4.2: An example of eight TREC queries with their relative frequency in the training set.

4.3.3 Search Shortcuts metric

We used Equation (4.1) to measure the similarity between the suggestions generated by SS, CG, and QFG for the first queries issued by a user during a satisfactory session belonging to the test set, and the final queries actually submitted by the same user during the same session. We conducted experiments by setting the number k of suggestions generated to 10, and, as in [12], we chose the exponential function $f(m) = e^m$ to assign an higher score to shortcuts suggested early. Moreover, the length t of the head of the session was set to $\lceil n/2 \rceil$, where n is the length of the session considered. Finally, the metric used to assess the similarity between two queries was the Jaccard index computed over the set of tri-grams of characters contained in the queries [52], while the similarity threshold used was 0.9.

Due to the long execution times required by CG, and QFG for generating suggestions, it was not possible to evaluate suggestion effectiveness by processing all the satisfactory sessions in the test set. We thus considered a sample of the test set constituted by a randomly selected group of 100 satisfactory sessions having a length strictly greater than 3. The histogram in Figure 4.3 plots the distribution of the number of sessions vs. the quality of the top-10 recommendations produced by the three algorithms. Results in the plot are grouped by quality range. As an example, the second group of bars shows the number of sessions for which the three algorithms generated suggestions having a quality (measured using Equation (4.1)) ranging from from 0.1 to 0.2. Results show that recommendations produced for the majority of sessions by QFG and CG obtains a quite low score (in the interval between 0 to 0.1), while SS produces recommendations whose quality is better distributed among all the range.

In particular, SS produces recommendations having a quality score greater than 60% for 18 sessions out of 100. Moreover, in 36 cases out of 100, SS generates useful suggestions when its competitors CG and QFG fails to produce even a single effective suggestion. On average, over the 100 sessions considered, SS obtains an average quality score equal to 0.32, while QFG and CG achieves 0.15 and 0.10, respectively.

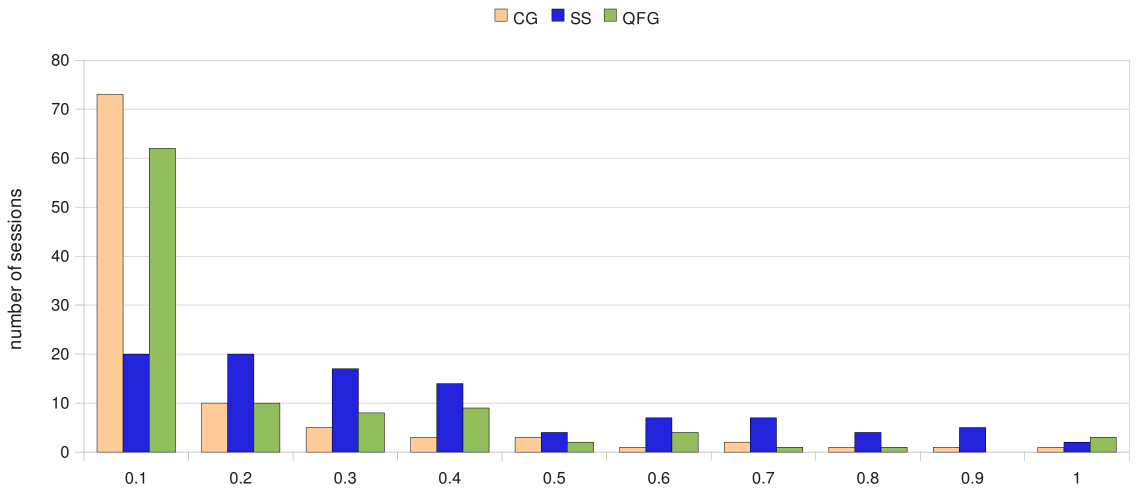

Figure 4.3: Distribution of the number of sessions vs. the quality of the top-10 recommendations produced by the three algorithms.

4.3.4 Suggestions Quality on TREC topics

The relevance of the suggestions generated by SS, CG, and QFG w.r.t. the TREC query subtopics was assessed manually[1]. Seven volunteers were chosen among CS researchers working in different research groups of our Institute. The evaluation consisted in asking assessors to assign, for any given TREC query, the top-10 suggestions returned by SS, CG, and QFG to their related subtopic. Editors were also able to explicitly highlight that no subtopic can be associated with a particular recommendation. The evaluation process was blind, in the sense that all the suggestions produced by the three methods were presented to editors in a single, lexicographically ordered sequence where the algorithm which generated any specific suggestion was hidden. Given the limited number of queries and the precise definition of subtopics provided by NIST assessors, the task was not particularly cumbersome, and the evaluations generated by the assessors largely agree. Table 4.1 shows the outcome of one of the editors for the TREC query n. 8. The note in bold after each suggestion indicates the subtopic to which the particular suggestion was assigned (e.g. *employee appraisals* in the CG column matches subtopic S3). Thus for this topic this editor gave to both SS and CG a coverage of 3/4 (3 subtopics covered out of 4), while QFG was rated 1/4. Moreover, suggestions in italic, e.g. *gmac* in the CG column, were considered by the editor not relevant for any of the subtopics. Thus, for topic *appraisals* SS and QFG score an effectiveness equal to 1 (all suggestions generated were considered relevant), whereas CG score was 4/5 (2 non relevant suggestions out of 10).

The histogram shown in Figure 4.4 plots, for each of the 50 TREC topics, the average coverage (Definition 4) of the associated subtopics measured on the basis of assessor's evaluations for the top-10 suggestions returned by SS, CG, and QFG. By

[1]All the queries suggested by the three algorithms for the 50 TREC queries are available to the interested reader along with the associated subtopic lists at the address `http://searchshortcuts.isti.cnr.it/TREC_results.html`.

looking at the Figure, we can see that SS outperforms remarkably its competitors. On 36 queries out of 50 SS was able to cover at least half of the subtopics, while CG only in two cases reached the 50% of coverage, and QFG on 8 queries out of 50. Moreover, SS covers the same number or more subtopics than its competitors in all the cases but 6. Only in 5 cases QFG outperforms SS in subtopic coverage (query topics 12, 15, 19, 25, 45), while in one case (query topic 22) CG outperforms SS. Furthermore, while SS is always able to cover one or some subtopics for all the cases, in 15 (27) cases for QFG (CG) the two methods are not able to cover any of the subtopics. The average fraction of subtopics covered by the three methods is: 0.49, 0.24, and 0.12 for SS, QFG, and CG, respectively.

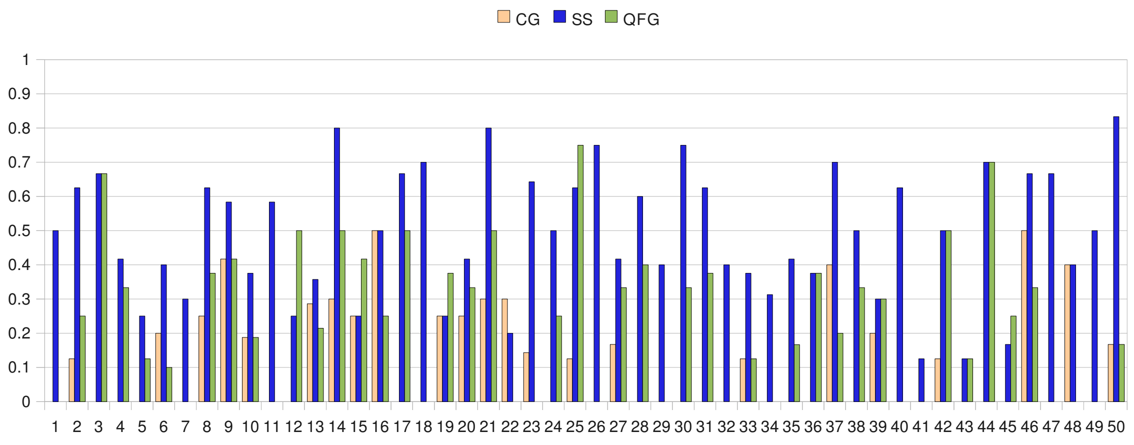

Figure 4.4: Coverage of the subtopics associated with the 50 TREC diversity-track queries measured by means of an user-study on the top-10 suggestions provided by the Cover Graph (CG), Search Shortcuts (SS), and Query Flow Graph (QFG) algorithms.

Figure 4.5 reports the effectiveness (Definition 5) of the top-10 suggestions generated by SS, QFG, and CG. Also considering this performance metric our Search Shortcuts solution results the clear winner. SS outperforms its competitors in 31 cases out of 50. The average effectiveness is 0.83, 0.43, and 0.42 for SS, QFG, and CG, respectively. The large difference measured is mainly due to the fact that both

CG and QFG are not able to generate good suggestions for queries that are not popular in the training set.

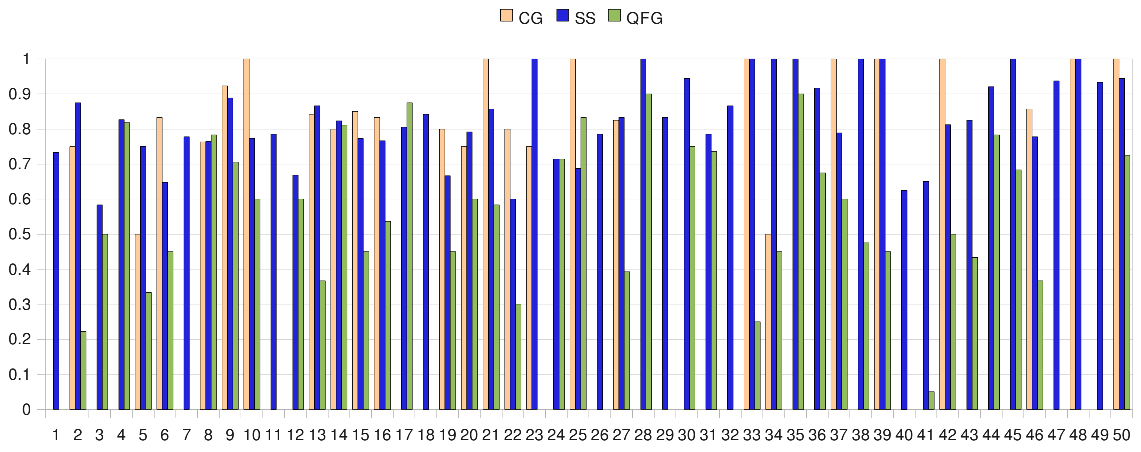

Figure 4.5: Effectiveness measured on the TREC query subtopics among the top-10 suggestions returned by the Cover Graph (CG), Search Shortcuts (SS), and Query Flow Graph (QFG) algorithms.

Regarding this aspect, the histogram in Figure 4.6 shows the average effectiveness of the top-10 suggestions returned by SS, CG and QFG measured for groups of TREC queries arranged by their frequency in the training set. SS remarkably outperforms its competitors. SS is in fact able to produce high-quality recommendations for all the categories of query analyzed, while CG and QFG can not produce recommendations for unseen queries. Furthermore, while SS produce constant quality recommendations with respect to the frequency of the TREC queries, CG and QFG show an increasing trend in the quality of recommendations as the frequency of the TREC queries increases.

For this reason, we can assert that *the SS method is very robust to data sparsity which strongly penalizes the other two algorithms, and is able to effectively produce significant suggestions also for singleton queries which were not previously submitted to the Web search engine.* We recall that singleton queries account for almost half of the whole volume of unique queries submitted to a Web search engine, and are

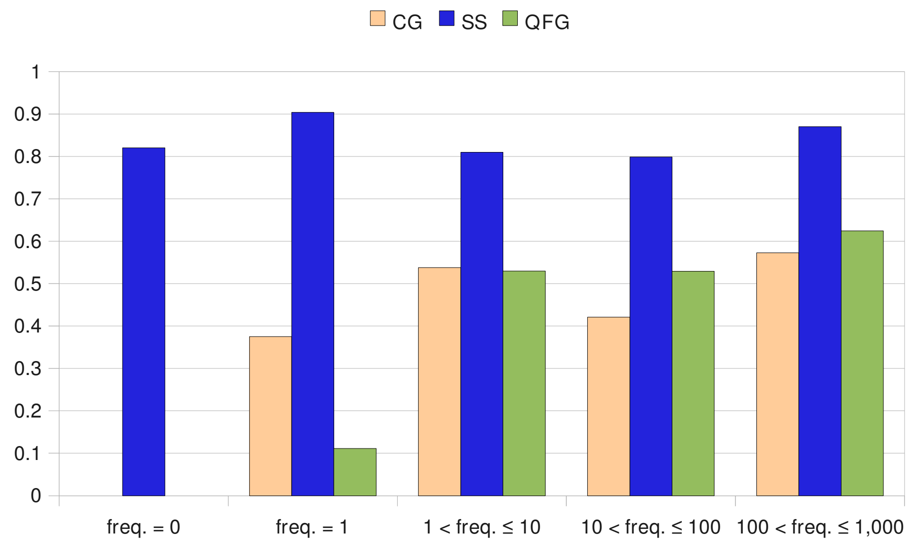

Figure 4.6: Average effectiveness of the top-10 suggestions provided by the Cover Graph (CG), Search Shortcuts (SS), and Query Flow Graph (QFG) algorithms for groups of TREC queries arranged by their frequency (freq.) in the training set.

often the hardest to answer since they ask for "rare" or badly expressed information needs. The possibility of suggesting relevant alternatives to these queries is more valuable than the one of suggesting relevant alternatives to frequent queries, which express common and often easier to satisfy needs.

Just to give an example of the results we obtained and the data on which we evaluated the quality, Tables 4.3 and 4.4 report the top-10 suggestions provided by SS, CG, and QFG for some TREC Web Track's diversity task query topics. For each query topic, the first column of the table lists the associated subtopics. These examples are representative of the figures above discussed: SS computed mostly relevant suggestions covering a significant subset of the subtopics. CG, on the other hand, performed worst and returned three suggestions only for a single query among the five reported in the table, and one single suggestion in another case. QFG returned instead 10 suggestions for three topics, and no suggestions in two cases.

Queries and their subtopics	SS	CG	QFG
TREC query (n. 18): ***wedding budget calculator*** (faceted); Not present in the training set;			
1. I want to find online guides, tips, and checklists for planning a wedding. 2. I am looking for spreadsheets or templates to help me tabulate a budget for a wedding. 3. I want to find some example wedding budgets. 4. I'm looking for information on planning a wedding shower, like theme ideas and budget guidelines. 5. How can I plan an inexpensive wedding?	budget wedding budget sheet sample wedding budgets budget calculator budget outside wedding wedding planning budget wedding bouquets how to have a celebrity wedding on a budget planning a wedding on a budget wedding costs	*no suggestion provided*	*no suggestion provided*
TREC query (n. 49): ***flame designs*** (ambiguous); Frequency in the training set: 1;			
1. Find free flame design clipart I can use on a web-site. 2. How do I make realistic flame images using Photo-shop? 3. I'm looking for good flame tattoo designs. 4. Find pictures of flames and fire. 5. I want to find flame design decals I can put on my car or motorcycle. 6. I'm looking for flame design stencils.	flames flame fonts penny flame flame illustration flaming text flaming fonts tattoo flame designs flame pattern comforters in flames band flame art	*no suggestion provided*	*no suggestion provided*
TREC query (n. 50): ***dog heat*** (ambiguous); Frequency in the training set: 2;			
1. What is the effect of excessive heat on dogs? 2. What are symptoms of heat stroke and other heat-related illnesses in dogs? 3. Find information on dogs' reproductive cycle. What does it mean when a dog is in heat?	heat heat cycle of a dog can a dog be spayed in heat dogs heat dog in heat symptoms do female dogs howl in heat cycle dog heat exhaustion signs of a female dog in heat heat cycle of dogs when do dauschound go into heat	female dogs in heat how long how often do dogs come in heat female dogs in heat	dogs in heat do female dogs howl in heat cycle how to tell if your dog is pregnant can a dog be spayed in heat memphis spaying pet vacs pet vax myspace yspace myspac

Table 4.3: Query suggestions provided by Search Shortcuts, Cover Graph, and Query Flow Graph for some rare TREC diversity-track query topics.

Queries and their subtopics	**SS**	**CG**	**QFG**
TREC query (n. 42): ***the music man*** (faceted); Frequency in the training set: 5;			
1. Find lyrics for songs from The Music Man.	till there was you	the music man movie	the music man on broadway
2. Find current performances of The Music Man.	musical music man lyrics		the music man summary
3. Find recordings of songs from The Music Man.	the music man soundtrack		state fair musical
4. I'm looking for the script for The Music Man.	the music man summary		till there was you
	elephant man music		dizzy gilelespee
	70s music rubberband man		oysters rockefeller recipe
	encino man songs		archnid
	music man lyrics		female whale
	free music on msn		brewski
	music man trouble in river city		fats dominos first name
TREC query (n. 24): ***diversity*** (faceted); Frequency in the training set: 27;			
1. How is workplace diversity achieved and managed?	diversity in education	*no suggestion provided*	accepting diversity
2. Find free activities and materials for running a diversity training program in my office.	diversity inclusion		dispariging remarks
3. What is cultural diversity? What is prejudice?	cultural diversity		diverse world
4. Find quotes, poems, and/or artwork illustrating and promoting diversity.	diversity test		diversity director
	accepting diversity		diversity poem
	diversity poem		diversity test
	diversity skills		minority & women
	diverse learners presentation		civil liberties
	picture of diverse childern		inclusion
	advantages of diversity		gender and racial bias

Table 4.4: Query suggestions provided by Search Shortcuts, Cover Graph, and Query Flow Graph for some frequent TREC diversity-track query topics.

4.4 Conclusions

We proposed a very efficient solution for generating effective suggestions to Web search engine users based on the model of *Search Shortcut*. Our original formulation of the problem allows the query suggestion generation phase to be re-conducted to the processing of a full-text query over an inverted index. The current query issued by the user is matched over the inverted index, and final queries of the most similar satisfactory sessions are efficiently selected to be proposed to the user as query shortcuts. The way a satisfactory session is represented as a virtual document, and the IR-based technique exploited, allow our technique to generate in many cases effective suggestions even to rare or not previously seen queries. The presence of *at least* one query term in at least a satisfactory session used to build our model, permits in fact SS to be able to generate *at least* a suggestion.

By using the automatic evaluation approach based on the metric defined in Equation (4.1), SS outperformed QFG in quality of a 0.17, while the improvement over CG was even greater (0.22). In 36 evaluated sessions out of 100, SS generated useful suggestions when its competitors CG and QFG failed to produce even a single useful recommendation.

An additional contribution of this work regards the evaluation methodology used, based on a publicly-available test collection provided by a highly reputed organization such as the NIST. The proposed methodology is objective and very general, and, if accepted in the query recommendation scientific community, it would grant researchers the possibility of measuring the performance of their solution under exactly the same conditions, with the same dataset and the same evaluation criterium.

On the basis of the evaluation conducted by means of the user-study, SS remarkably outperformed both QFG and CG in almost all the tests conducted. In particular, suggestions generated by SS covered the same number or more TREC subtopics than its two counterparts in 44 cases out of 50. In 36 cases the number of subtopics covered by SS suggestions was strictly greater. Only in 5 cases QFG outperformed SS, while this never happens with CG. Also when considering effectiveness, i.e. the number of relevant suggestions among the top-10 returned, SS resulted the clear winner with an average number of relevant suggestions equal to 8.3, versus 4.3, and 4.2 for QFG, and CG, respectively. Moreover, differently from competitors SS resulted to be very robust w.r.t. data sparsity, and can produce relevant suggestions also to queries which are rare or not present in the query log used for training.

All the queries suggested by the three algorithms for the 50 TREC queries given as input to assessors are available along with the associated subtopic lists at `http://searchshortcuts.isti.cnr.it/TREC_results.html`. Moreover, a simple web-based wrapper that accepts user queries and computes the associated top-20 SS recommendations is available at `http://searchshortcuts.isti.cnr.it`.

As future works we intend to evaluate the use the whole head of the user session for producing query recommendations. Furthermore, we want to study if the sharing of the same final queries induces a sort of "clustering" of the queries composing the satisfactory user sessions. By studying such relation which is at the basis of our query shortcut implementation, we could probably find ways to improve our methodology. Finally, it would be interesting to investigate how IR-like diversification algorithms (e.g., [2]) could be integrated in our query suggestion technique in order to obtain diversified query suggestions [67], [20].

5 Load-Sensitive Selective Pruning

A search engine infrastructure must be able to provide the same quality of service to all queries received during a day. During normal operating conditions, the demand for resources is considerably lower than under peak conditions (Figure 3.4), yet an oversized infrastructure would result in an unnecessary waste of computing power. A possible solution adopted in this situation might consist of defining a maximum threshold processing time for each query, and dropping queries for which this threshold elapses (Chapter 6), leading to disappointed users. In this Chapter, we propose and evaluate a different approach, where, given a set of different query processing strategies with differing efficiency, each query is considered by a framework that sets a maximum query processing time and selects which processing strategy is the best for that query, such that the processing time for all queries is kept below the threshold. The processing time estimates, used by the scheduler, are learned from past queries.

In this Chapter we assume a distributed search engine, like the one described in Chapter 2, where data is distributed according to a *document partitioning* strategy [9]. In this work, we assume a multi-node search engine without replicas, because our experimental results are independent from the number of replicas, and hence can be applied directly to each replica independently [33].

We experimentally validate our approach on $10,000$ queries from a standard TREC dataset with over 50 million documents, and we compare it with several baselines. These experiments encompass testing the system under different query loads and different maximum tolerated query response times. Our results show that, at the cost of a marginal loss in terms of response quality, the search system is able to answer 90% of queries within half a second during times of high query load.

5.0.1 Query Efficiency Prediction

The query scheduler component used in this thesis, must select the next query to be processed from the queue of waiting queries. To achieve this, it is fundamental to know in advance an estimation of the processing time for the query to be scheduled. Efficiency predictions can be used to estimate the response time of a search engine for a query [69].

Moffat et al. [76] stated that the response time of a query is related to the posting list lengths of its constituent query terms. However, in dynamic pruning strategies (e.g. WAND [26]), the response time of a query is more variable, as not every posting is scored, and many postings can be skipped, resulting in reduced retrieval time. As a result, for WAND, the length of the posting lists is insufficient to accurately predict the response time of a query [69]. In fact, the response time of WAND depends also on the number of postings that are actually scored, as well as the *pruning difficulty* of the query, i.e. the number of postings that overlap for the constituent query terms, and the extent to which high-scoring documents occur towards the start of the posting lists. *Query efficiency predictors* [69] have been proposed to address the problem of predicting the response time of WAND for an unseen query. In particular, various term-level statistics are computed for each term offline. When a new query arrives, the term-level features are aggregated into query-level statistics, which are used as input to a learned regression model.

In this thesis, arising from our focus on the `TAAT-CS` pruning strategy, we propose query efficiency predictions for `TAAT-CS`, by describing a set of features that can be easily used to estimate the efficiency of a query through a learned approach. These predictions represent our estimates for the query processing time, which we exploit to determine a maximum amount of processing time to allocate for each query.

5.1 Load-Driven Selective Pruning

One of the problems that must be addressed to build a large-scale Web search engine is how to provide the service when the received query volume is excessively high. In particular, when the entire system is overloaded, the response time of the queries increases, making it necessary to answer queries more rapidly. A common strategy is to drop queries that have been waiting or executing for a long time, returning empty results list; alternatively, it is possible to set a time threshold and interrupt the retrieval whenever a query is going to take too much time. Both strategies are suboptimal and have the huge drawback of disappointing the users who submitted those queries that have been dropped. Typically, in search systems critical situations arise when bursts of queries are submitted (almost) at the same time. See, for instance, the peak load around 12 PM in the query workload plotted in Figure 3.4.

In this section, we discuss a novel load-sensitive framework, based on query efficiency predictors and taking into account other features like the length of the list of queries waiting to be processed and the duration each query has been queued for. We aim to dynamically adapt the retrieval strategy, by reducing the processing time of queries when the system is heavily loaded. Indeed, during high query load, we propose to adopt aggressive pruning strategies, thus speeding up query processing, while possibly impacting negatively on the effectiveness of the returned results.

Let us consider the search engine state depicted in Figure 5.1, which shows the system at time t. It has n queries $q_1, \ldots, q_n$ waiting to be processed in the scheduler's queue. Let t_i be the arrival time of query q_i, where $t_i \leq t_j$ whenever $i < j$, i.e., $t_1 \leq \ldots, \leq t_n \leq t$. Query q_1 is the head of the queue, as it has been queued for the longest time.

Until time t, the query processor was busy by processing the previous queries

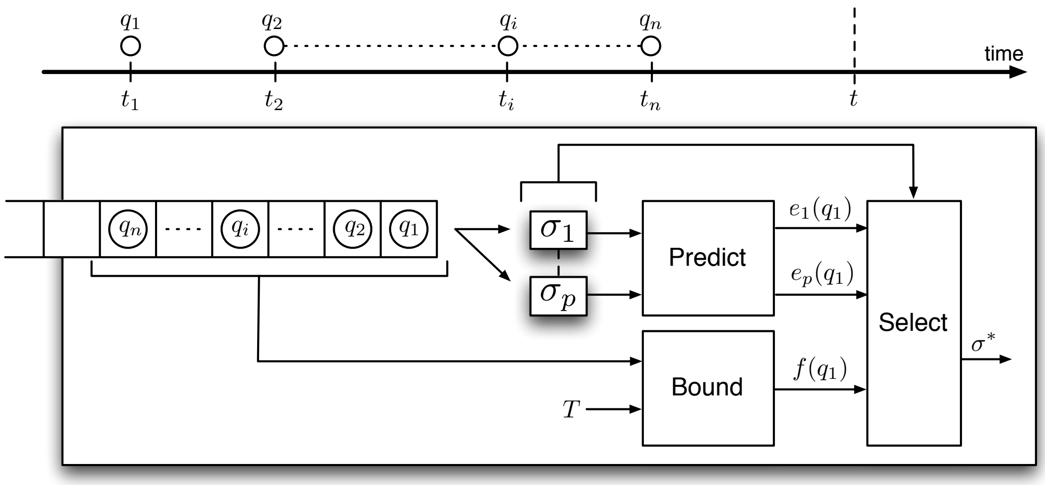

Figure 5.1: The components of the proposed load-sensitive selective pruning framework (bottom), along with a representation of the variables depicting the queries currently queued (top).

(not shown in the figure), and at time t it becomes idle. Then, the query scheduler must select the next query to be processed. We assume that scheduling follows a first-in first-out discipline, that is, query q_1 – which has been queued for the longest time – is selected for processing next. Furthermore, each query can be processed by several processing strategies $\sigma_1, \ldots, \sigma_p$, such as `TAAT` or `DAAT` with different levels of dynamic pruning aggressiveness. We assume that strategy σ_1 is the search engine's full processing strategy, such as `TAAT` or `DAAT`, while subsequent strategies are increasingly more efficient, such that σ_p is the most efficient processing strategy. Moreover, we assume that, while σ_{i+1} is more efficient than σ_i, the effectiveness of σ_i is, in general, better than the effectiveness of σ_{i+1}. This assumption is well-founded, because efficient processing strategies typically have a negative impact on the corresponding retrieval effectiveness [68, 113].

For query q_1, we associate with each strategy σ_k the processing time $e_k(q_1)$, which the strategy is predicted to take to process query q_1. This means that, for example, $e_1(q_1)$ represents the processing time of query q_1 when the less efficient (but most effective) processing strategy is adopted, while $e_p(q_1)$ represents the most efficient yet less effective predicted processing strategy.

A constant time threshold T represents the maximum time budget for the processing of any query: the completion time of any query must be not greater than T, such that its results can be presented to the user in a timely manner. This means that the time elapsed between the arrival of any query and its processing finish time must not exceed T. Note that, since the query already spent some time in the queue, its available processing time, i.e., the maximum time it is allowed to spend in processing, is not, in general, equal to T, but it is decreased by the time it has spent in the queue. Moreover, if there are other subsequent queries queued, then it can be considered unfair for the query to take all time available, while other queries are starved. Hence, we argue that the available processing time for each query is bounded by some time budget depending on various factors such as the time the query has spent in the queue, and the number of queued queries.

The definition of a suitable time budget is central to this work. Let $f(q_i)$ be this time budget for query q_i, which has to ensure "*fairness*" in query processing: whenever the query workload is close to the maximum allowed, enqueued queries should be assigned reduced time budgets for their processing. Once $f(q_i)$ has been computed, we have to select the processing strategies able to process the query within the time budget, i.e. any strategy $\sigma_k(q_i)$ such that $e_k(q_i) \leq f(q_i)$. Finally, among all these strategies, we select the best strategy in term of effectiveness, i.e., according to our assumptions, the strategy that takes the largest processing time among all admissible strategies. The definition of a suitable time budget function $f(q_i)$ depends on various features: the position of the query in the queue, its arrival time, the current time, and the status of the queue.

The outline of the proposed selective pruning framework is shown in Algorithm 3. For a queue of queries awaiting processing, $q_1, \ldots, q_n$, their expected processing times for all possible processing strategies are estimated. This allows the time budget to be calculated $f(q_1)$ for the next query to be processed. Thereafter, we choose an appropriate query processing strategy, which aims to ensure that the query meets its completion time threshold T, while providing results that are as effective as possible.

Algorithm 3 Load-Sensitive Selective Pruning Framework

```
Input:   The queries $q_1, \ldots, q_n$
         The completion time threshold $T$
Output:  The selected processing strategy $\sigma^*$ for query $q_1$
1: for all processing strategies $\sigma_k$, $k = 1, \ldots, p$
2:     for all enqueued queues $q_i$, $i = 1, \ldots, n$
3:         expected processing time $e_k(q_i) \leftarrow \mathsf{Predict}(\sigma_k, q_i)$
4: Time budget $f(q_1) \leftarrow \mathsf{Bound}(T, \sigma_1(q_1), \ldots, \sigma_p(q_n))$
5: Processing strategy $\sigma^* \leftarrow \mathsf{Select}(f(q_1), e_1(q_1), \ldots, e_p(q_1))$
```

In order to select the processing strategy σ^*, we must implement the following functions within our framework:

- Predict(): Defines a mechanism allowing to predict the processing time for each query in the queue when the processing strategy can be selected among the different dynamic pruning processing strategies. This mechanism is used to estimate the processing times $e_k(q_i)$ of the available processing strategies, and the pruning strategy that will most likely to process the query within the desired time threshold T.

- Bound(): Defines a method to compute the time budget $f(q_1)$ for query q_1, depending on the global time threshold T and on the queries waiting to be processed. The time budget defines a bound on the processing time that query q_1 will be permitted.

- Select(): Defines a mechanism to select the "best" processing strategy that is able to process query q_1 according to the maximum processing time, $f(q_1)$, that q_1 is allowed to take and that maximises the resulting query effectiveness.

Similar to previous work on selective pruning [106], it follows that the processing times of a query can be estimated through the use of query efficiency prediction, i.e. Predict(). However, as no such predictors have previously been defined for TAAT strategies such as TAAT-CS, in Section 5.2 we address query efficiency prediction for TAAT. In the remainder of this section, we propose mechanisms for Bound() (Section 5.1.1) and Select() (Section 5.1.2).

5.1.1 Bound()

We assume a list of queries $q_1, q_2, \ldots, q_n$ that are currently (at time t) in the queue of the system. Each query is associated with its arrival time t_i. Roughly speaking, the query processing time bound $f(q_1)$ has the following goals:

1. **Efficiency**: q_1 (the least recently queued query) will have a completion time not greater than T, the global time threshold.
2. **Effectiveness**: The time available to process q_1 will be as large as possible, such that the most effective processing strategy can be deployed.
3. **Fairness**: Queries $q_2, \ldots, q_n$ received after q_1 are not starved of processing time, and hence are each able to meet T.

Clearly, these three goals can be at odds with each other. In the following, we describe three methods of defining $f(q_1)$ that address some or all of the goals to varying extents:

Perfectionist. Query q_1 is processed as effectively as possible, i.e. using the most inefficient processing strategy:

$$f(q_1) = \underset{k}{\operatorname{argmax}}\{e_k(q_1)\} = e_1(q_1).$$

This method ignores the waiting time spent in the queue, and makes no attempt to prune aggressively queries such that the threshold T can be met, by this query or other queries in the queue. In other words, it is a method that is neither fair nor efficient. For this reason we use it as a baseline to upper bound the effectiveness of the method.

Manic. Query q_1 is processed as fast as possible, by using the most efficient, aggressive pruning strategy for all queries:

$$f(q_1) = \operatorname*{argmin}_k \{e_k(q_1)\} = e_p(q_1).$$

In this method, we ignore the waiting time that the query q_1 has spent in the queue. Similarly to the Perfectionist method, Manic serves as a baseline method that does not consider the fairness or effectiveness goals. However, in contrast to Perfectionist, Manic consumes the least computing resources, and hence is the fairest method, even if the other queries does not exploit the unused resources.

Selfish. The query q_1, enqueued at time t_1, should be processed within $t_1 + T$ seconds. Then, at time t, we have to decrease the total time T allowed for query completion by t_1 seconds. Formally, the remaining time Δ_1 to process query q_1 such that threshold T is met is:

$$\Delta_1 = (t_1 + T) - t$$

If $\Delta_1 > 0$, the processing time bound is $f(q_1) = \Delta_1$, and depends only on the time q_1 has spent in the queue, without consideration for the processing time needed for other queued queries. In particular, if the time threshold T for this query has elapsed ($\Delta_1 \leq 0$), then the query is processed as fast as possible, as in the Manic case:

$$f(q_1) = \begin{cases} \Delta_1 & \text{if } \Delta_1 > 0 \\ e_p(q_1) & \text{otherwise} \end{cases}$$

Altruistic. The previous method has the disadvantage that q_1 processing is bound with the maximum amount of time available (given the time spent in the queue), disregarding the queries that are still in the queue. This can penalise queued queries $q_2, \ldots, q_n$ that have not yet been processed. In contrast, Altruistic enforces "fairness", by firstly computing how much time is left to empty the current queue.

This time is simply the time at which the lastly queued query q_n should be completed $(t_n + T)$ minus the current time. Formally, Δ_n, the remaining time to finish processing upto query n, is:

$$\Delta_n = (t_n + T) - t$$

Then, to compute the maximum time available for q_1 we have to subtract the minimum time necessary to process all the queued queries. This time is simply given by the sum of the estimations $e_p(q_i)$ of the processing time needed by the fastest processing strategy p. Hence, we define the available *slack time*, $\widetilde{\Delta}_n$, as:

$$\widetilde{\Delta}_n = \Delta_n - \sum_{i=1}^{n} e_p(q_i).$$

If $\widetilde{\Delta}_n > 0$, we evenly distribute this extra slack time to the queued queries. In doing so, if some time is left to process all enqueued queries faster than the minimum possible, each one might receive a fair amount of extra processing time[1]. Hence the processing bound for query q_1 becomes $e_p(q_1) + \widetilde{\Delta}_n/n$. However, this quantity can exceed Δ_1, and will result in too much extra budget assigned to query q_1, beyond the time threshold T. In this case, the processing bound for the query q_1 is simply Δ_1. Finally, if $\widetilde{\Delta}_n \leq 0$, we process the query as fast as possible, as in the Manic case, i.e.,

$$f(q_1) = \begin{cases} \min\left\{\Delta_1, e_p(q_1) + \widetilde{\Delta}_n/n\right\} & \text{if } \widetilde{\Delta}_n > 0 \\ e_p(q_1) & \text{otherwise} \end{cases}$$

The Altruistic method to compute Bound() is a central contribution of this Chapter. Once the time budget $f(q_1)$ has been computed, it is used by the query processor to "*select*" the most suitable processing strategy among those available to process the query. In the following, we describe Select(), which is the function used to take these decisions.

[1]This is true as far as no additional queries are received.

5.1.2 Select()

Given the time budget $f(q_1)$ granted by Bound(), the role of the Select() function is to choose the most effective strategy $\sigma^* = \sigma_k \in \{\sigma_1, \ldots \sigma_p\}$ to resolve query q_1 within the assigned budget $f(q_1)$. Primarily, the selection of an appropriate processing strategy is based on the estimated query processing times $e_1(q_1), \ldots, e_p(q_1)$. Assuming the estimates are sorted in descending order of expected processing times, i.e., $e_1(q_1) \geq \cdots \geq e_p(q_1)$, we can identify the strategy σ_k such that $\forall i = 1, \ldots, k-1$, $e_i(q_1) > f(q_1)$ and $\forall i = k, \ldots, p$, $e_i(q_1) \leq f(q_1)$. This means that all strategies $\sigma_k, \sigma_{k+1}, \ldots \sigma_p$ have an estimated processing time for query q_1 that respects the time budget $f(q_1)$. In other words, we select the smallest value of k in $1 \ldots p$ such that $e_k(q_1) \leq f(q_1)$, that is the strategy whose expected completion time is not greater than the budget the query has been granted by Bound(). Hence, among all these strategies, we select the best strategy in terms of effectiveness, i.e., according to our assumptions, the strategy that takes the largest processing time among all admissible strategies. Note that, in the case that no strategy is able to process query q_1 within the computed time budget, we always select the most aggressive processing strategy, i.e., σ_p. As a remark, when Manic and Perfectionist methods are used Select() will resort to always pick CS$-$1000 (i.e. σ_p) and `DAAT` (i.e. σ_1), respectively.

Both Bound() and Select() descriptions have been given using the informal, and implicit, concept of an *efficiency predictor*. In the next section, we detail in a more precise way how – inspired by the work in [106] – we predicting the efficiency of a `TAAT-CS` strategy before processing commences.

5.2 Processing Strategies & Predictors

The framework we described in the previous section relies on the concept of query efficiency predictors. In our definition, given a query and a set of query processing strategies, efficiency predictors return the estimated query processing time for each one of the strategies considered.

The load-sensitive selective pruning framework proposed in Section 5.1 is general with respect to the deployed retrieval strategy. However, in this work we focus on two particular strategies, namely `DAAT` and `TAAT-CS`. In particular, We adopt `DAAT` for full processing. Full-processing is chosen when, in normal load conditions, processing time is not constrained. On the other hand, when the system is experiencing workload peaks, we resort to use faster and less precise processing strategies, specifically, based on the term-at-a-time `Continue-Strategy` (`TAAT-CS`) [77]. In the remainder of this section, we define the details of `TAAT-CS` (Section 5.2.1), before explaining how the processing time of both `DAAT` and `TAAT-CS` can be accurately measured (Section 5.2.2).

5.2.1 TAAT-CS Dynamic Pruning

As defined in [77], `TAAT-CS` works as follows. Given a set of terms to process, sorted in decreasing order of posting list length, an OR phase processes the posting lists one by one until we have K accumulators. From this point, no new accumulators are created, and an AND phases processes the remaining posting lists by intersecting them with the existing accumulators. The efficiency of the AND phase can benefit from skip pointers within the posting lists, such that the postings of documents that are not in the top-K accumulators are not decompressed, leading to IO benefits. Therefore, smaller values of K correspond to more aggressive pruning, as the AND phase is started earlier, and more skipping can occur during this phase. However, smaller K values are likely to lead to result lists with degraded effectiveness.

Our implementation of the `TAAT-CS` dynamic pruning strategy adopts a further heuristic, to optimise the initial phase in which new accumulators are created. Given that `DAAT` processing is faster than `TAAT` processing [45], we alter the accumulator creation phase as follows. We select the smallest l lists such that the sum of their lengths is greater than or equal to the number of accumulators K. These posting lists for this initial set of terms are processed using a `DAAT` strategy, instead of `TAAT`. In doing so, the resulting number of accumulators will never be greater than the number of accumulators we will get after processing the first list with a classic `TAAT-CS` strategy. After this modified OR phase, the processing strategy proceeds with the AND phase as in `TAAT-CS`. Using our refined strategy, we may end up with less accumulators than using the traditional `TAAT-CS`. However, in our initial experiments, we found that this happens only for 0.01% of the 10,000 queries used in this experiments. Yet, on average, the response time of our `DAAT`/`TAAT-CS` strategies exhibit a 2x improvement over the classical `TAAT-CS` strategy.

The adoption of the `DAAT`/`TAAT-CS` strategy motivates also the comparison of our selective pruning strategies with `DAAT`, instead than `TAAT`. Beating `DAAT` as a baseline is, in general, tougher than beating `TAAT` in terms of efficiency [45]. In the following, we refer to our `DAAT`/`TAAT-CS` with K accumulator as CS-K (e.g. CS-1000 uses $K = 1000$ accumulators), without stressing anymore the use of `DAAT` for the initial phase. As a side note, we are not aware of any previous work studying this small variation on `TAAT-CS`. Therefore, to the best of our knowledge, this is another new contribution presented by this work.

5.2.2 Query Efficiency Prediction

In the preceding, we defined the processing strategies used within this Chapter. In this section, we describe how we obtain query efficiency predictions for the processing strategies. In particular, we are inspired by the query efficiency predictors for `DAAT` previously defined by Macdonald *et al.* [69]. However, in this work we also use `TAAT-CS` for aggressive pruning. Hence, in the following we devise a method for predicting the processing time of CS-K, before retrieval commences, using a *Linear Regression*-based technique.

First of all, we have define a set of features to represent each query. In the case of `DAAT`, Macdonald *et al.* [69] show that there is a strong correlation between the

Query Efficiency Prediction Features
total number of postings in the query's term lists
number of terms in the query
variance of the length of the posting lists
mean of the length of the posting lists
length of the shortest posting list
length of the longest posting list
number of terms processed in the first phase of CS
length of the posting lists processed in the first phase of CS
number of terms processed in the second phase of CS
length of the posting lists processed in the second phase of CS

Table 5.1: Features used for prediction processing time: the top features are method independent, the bottom features are method dependent, for CS.

distribution of postings in the query terms and the response time of the query itself. Therefore, to predict the response time of `DAAT` we use the features listed in the top part of Table 5.1.

On the other hand, as discussed above, `TAAT-CS` strategies do not score all postings in the posting lists of the query terms. Hence, we do not expect that relying only on posting features can lead to good predictions. Instead, given the characteristics of our `TAAT-CS` strategies (a first phase where we fully evaluate a subset of terms, a second phase where we use the remaining terms to update the accumulators found in the first phase) we build a regression model using the features listed in the bottom part of Table 5.1, in addition to the method independent featured listed in the top part. It is of note that all of these query efficiency prediction features can be calculated using commonly available statistics, particularly the length of the query term's corresponding posting lists, before retrieval commences, and hence query efficiency predictions can be made with very low overheads, as soon as a query arrives at a query server.

In total, our prediction method models the problem using a feature space made up of 10 distinct features. As our reference architecture is a distributed one, each query server might have different response times for the same query. For this reason, we need to build different models for each server.

We adopt a linear regression model to estimate the running time $e_j(q_i)$ of query q_i when scored using method j. In other words, we model $e_j(q_i)$ as a linear combination of the features $\mathsf{f}i$ weighted by a real value λ_f. Features and weights are different for each scoring method thus we indicate $\mathsf{f}_j i$ and λ_{jf} to refer to values for scoring method j. Formally,

$$e_j(q_i) = \lambda_{j0}\mathsf{f}_j 0 + \ldots + \lambda_{j9}\mathsf{f}_j 9.$$

Linear regression is then used to find the values for various λ_{jf} with the goal of minimising the least square error on a training set of queries [69].

In the next section, we define the experimental setup for our experiments. In particular, we show the accuracy of the proposed efficiency predictors for `TAAT-CS`, before showing how the proposed selective scheduling framework proposed in Section 5.1 can increase the ability of a search engine to effectively and efficiently handle different traffic query loads.

5.3 Experimental Setup

In the following experiments, we deploy a widely used document collection created as part of TREC, namely the ClueWeb09 (cat. B) collection, which comprises around 50 million English Web documents, and is designed to represent the first-tier index of a commercial Web search engine. We index the document collection using the Terrier search engine [79], removing standard stopwords and applying Porter's English stemmer. The resulting index is document partitioned into ten separate index shards, while maintaining the original ordering of the collection. Each inverted index shard also has skipping information embedded, to permit skipping [77] during the CONTINUE phase of `TAAT-CS`.

For the retrieval experiments, we use a distributed C++ search system engine, accessing the index produced by Terrier. Our experiments are conducted on a cluster of twelve quad-core machines, where each machine has one Intel Xeon 2.40GHz X32230 CPU and 8GB of RAM, connected using Gigabit Ethernet. Our distributed architecture is organised with a single index shard on each of the ten query servers. Two additional nodes are used as follows: one as the query broker, and one as the client application that sends the queries to the system. Finally, each query server has a queue used to keep queries coming from the broker, while the Query Processor on each query server processes queries one at a time. As query processing strategies, we use `DAAT`, as well as `TAAT-CS` with different accumulators, i.e. CS-1000, CS-2000, CS-5000 and CS-10000. Documents are scored using BM25, with parameters at the default settings [88].

We use queries from the TREC Million Query Track 2009 [36], which contains $40,000$ queries, some of which have relevance assessments. In our experiments, $30,000$ of these queries are used as the training set for learning λ values in our regression models, while the other $10,000$ are used for testing the accuracy of the predictors, and retrieval experiments. Indeed, for measuring the accuracy of our query efficiency predictors, we use root mean square error (RMSE), while for retrieval effectiveness, we compute NDCG@1000 using the 687 queries out of the $10,000$ that have relevance assessments from TREC 2009. Efficiency is measured using mean response time.

5.4 Experiments

In the following, we address these research questions:

RQ1. What is the accuracy of the linear regression-based approach for query efficiency prediction for `TAAT-CS`? (Section 5.4.1)

RQ2. Do the proposed methods achieve effective and efficient retrieval under different query loads? (Section 5.4.2)

RQ3. To what extent can efficient query per second servicing be attained for different time thresholds? (Section 5.4.3)

5.4.1 Predictors Error Evaluation

Efficiency predictors, which aim to predict the processing time of a query before retrieval commences, are an important component of our work. In this first research question, we aim to ensure that our estimations, particularly for TAAT CS pruning strategies, are accurate. We compare the accuracy of the features listed in Table 5.1 when combined using linear regression. In particular, we compare the set that only includes the six "method independent" features, with the set that includes, in addition to the previous six, the four "method dependent" features proposed for TAAT CS. Table 5.2 reports the accuracy of the linear regression models combining the six and ten features, as well as a baseline predictor that uses only the total number of postings for the query terms as a feature. In the table, we report the mean, over the ten query servers, of the query processing time (QPT) for each strategy, as well as the Root Mean Square Error (RMSE), and the percentage of queries for which the prediction error is less than 10 milliseconds. The best value in each row for each measure is highlighted.

On analysing Table 5.2, we note that for DAAT, using the six features improves over the baseline single feature predictor by 42% (from RMSE $8.78 \cdot 10^{-3}$ to $4.98 \cdot 10^{-3}$), with 95% of the queries having a prediction error of less than 10 ms. On the other hand, using only the six features is insufficient for accurate processing time prediction for the CS-K strategies – for instance, for CS-10000, only 65% of queries are accurately predicted within 10 ms. However, for the linear regression models that uses the additional 4 method dependent features (10 features in all)[2], the error is one order of magnitude lower, and for the vast majority of queries (95-99%) our linear model is able to guess the correct response time up to a 10ms error.

Therefore, in answering research question **RQ1**, we find that the proposed linear regression model is accurate, with an error smaller than 10 ms in more than 95% of the cases. In particular, the best performing models for predicting CS-K strategies

[2]The 4 method dependent features do not apply to DAAT.

		1 Feature: sum of postings		6 Features: method independent		10 Features: incl. method depend.	
Strategy	QPT	RMSE	err $\leq$ 10 ms	RMSE	err $\leq$ 10 ms	RMSE	err $\leq$ 10 ms
DAAT	0.110 s.	$8.78 \cdot 10^{-3}$	87.83 %	$\mathbf{4.98 \cdot 10^{-3}}$	**95.53 %**	-	-
CS-1000	0.025 s.	$1.96 \cdot 10^{-2}$	65.85 %	$1.07 \cdot 10^{-2}$	86.08 %	$\mathbf{2.88 \cdot 10^{-3}}$	**99.44 %**
CS-2000	0.030 s.	$2.50 \cdot 10^{-2}$	63.52 %	$1.96 \cdot 10^{-2}$	80.38 %	$\mathbf{3.55 \cdot 10^{-3}}$	**98.63 %**
CS-5000	0.037 s.	$2.64 \cdot 10^{-2}$	56.74 %	$2.16 \cdot 10^{-2}$	72.30 %	$\mathbf{4.29 \cdot 10^{-3}}$	**97.11 %**
CS-10000	0.044 s.	$2.78 \cdot 10^{-2}$	51.31 %	$2.32 \cdot 10^{-2}$	65.10 %	$\mathbf{4.64 \cdot 10^{-3}}$	**96.55 %**

Table 5.2: Mean query processing time (QPT, in seconds), as well as prediction accuracy using various feature sets, for each processing strategy.

are those obtained by the full set of ten features described in Section 5.2, while in the case of DAAT, the six features describing the lengths of the lists associated with query terms perform very well. Therefore, in the following experiments, we use six features for the prediction of DAAT processing times and the full set of ten features for the prediction of TAAT CS-K scoring strategies.

5.4.2 Efficiency and Effectiveness Analysis

In this section, we experiment to address **RQ2**, in comparing the efficiency and effectiveness of our proposed load-sensitive selective pruning framework. In particular, we compare our methods, Selfish and Altruistic, with three different baselines: Perfectionist and Manic, as well as applying CS$-$10000 for all queries. We remark that, by their respective definitions, Perfectionist corresponds to a pure `DAAT` full processing strategy and Manic corresponds to using CS-1000. Within this section, we use a maximum threshold time of $T = 0.5$ seconds, which mandates that the results for each query must be returned, including both queueing and processing, before this time elapses. Later, in Section 5.4.3, we analyse how T affects the performances of our methods.

We analyse our methods in terms of query response time and effectiveness, stressing our search system with different rates of queries, measured in queries per second (q/s). The query response time corresponds to measuring how much time the query spends within the queues and being processed – in other words the time a user waits for the results to be returned. We evaluate effectiveness using NDCG@1000, exploiting the 687 queries that have relevance assessments and we use *RBO* metric [114] to examine the results degradation between full processing and the proposed strategies for all of the $10,000$ queries tested.

Firstly, we experiment to determine the average response time of the various methods by varying the number of queries per second submitted to the search system. We remark that queries are submitted uniformly, in other words a submission

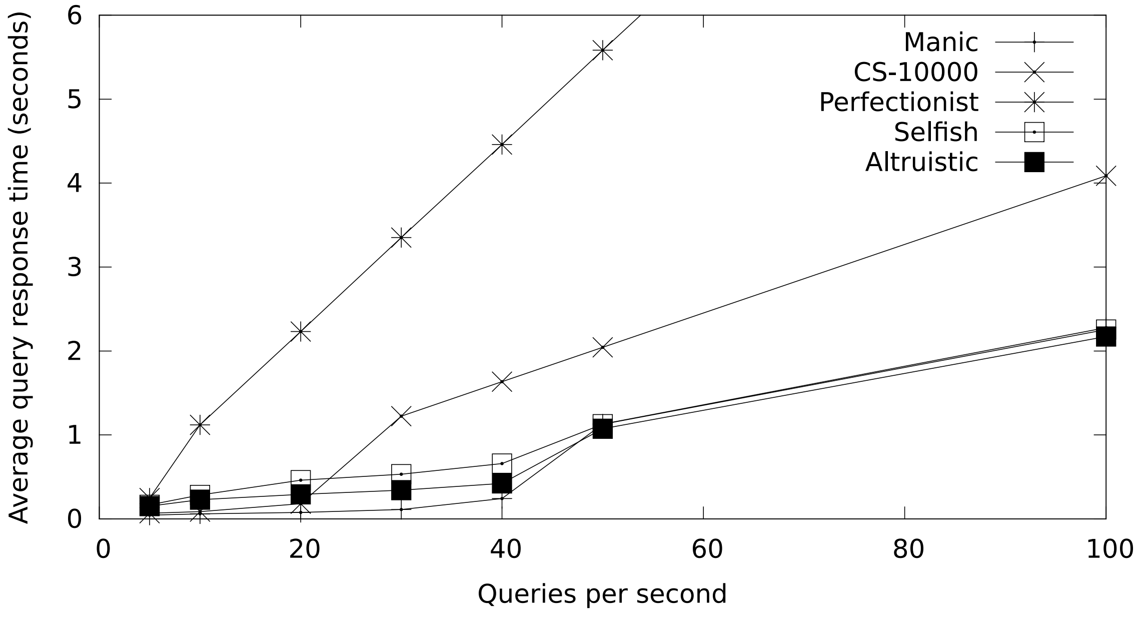

Figure 5.2: Average query response time in seconds for different methods, $T = 0.5$.

rate of N q/s corresponds to submitting a query every $1/N$ seconds. Figure 5.2 shows the behaviour of the various techniques we tested under various load conditions. As expected, when using Perfectionist method, the mean response time exceeds the threshold ($T = 0.5$) for all except very low workloads. CS$-$10000 can sustain slightly higher loads than Perfectionist, however for loads greater than 20 q/s the response times are well above the threshold.

Figure 5.3 enlarges the curves of Figure 5.2 for query response times up to the threshold $T = 0.5$. This allows us to better analyse the behaviour of the various methods for a workload of 40 q/s or less. Clearly, Manic attains the smallest response times, as it aggressively prunes all queries. However, both Selfish and Altruistic methods are less efficient than Manic, but still achieve the threshold up to 40 q/s.

To show how the various methods cope with queries of varying efficiency, Figure 5.5 plots the actual query response times for a subset (one hundred) of all the test queries, for a query workload of 40 q/s. In particular, the response times for Manic, CS-10000, Selfish, and Altruistic are shown. Spikes in the lines correspond

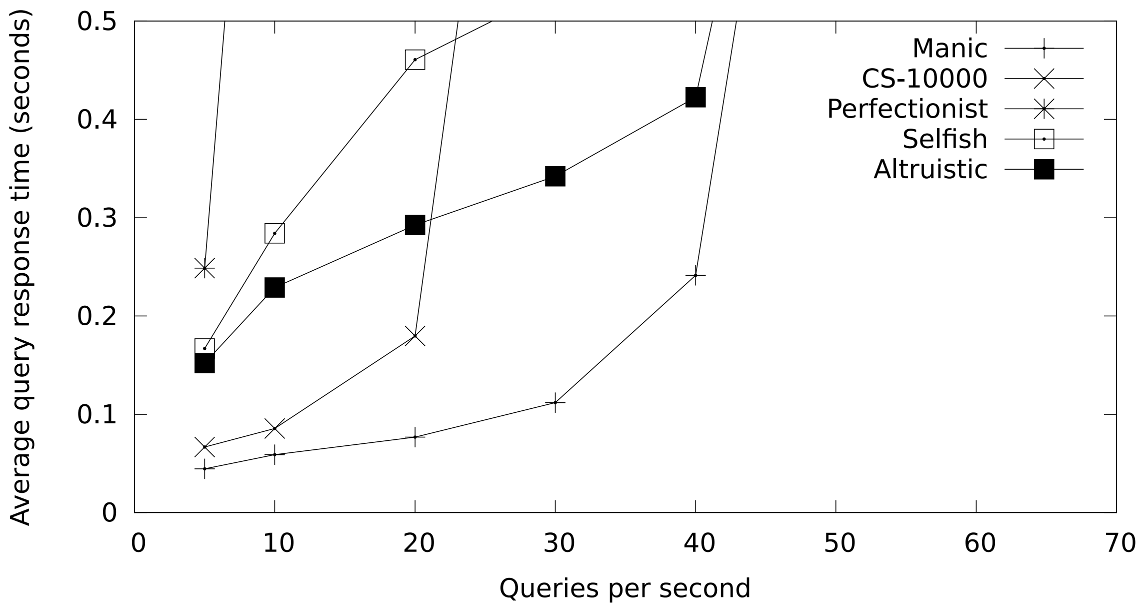

Figure 5.3: Average query response time in seconds for different methods (enlargement of Figure 5.2).

to the effect of expensive queries on other later queries. Indeed, expensive queries delay the queries submitted later, as expected though Selfish and Altruistic are more uniform than the others. In particular, in the case of Altruistic, the line is also close to the time threshold, indicating a better utilisation of the resources.

Finally comparing Figures 5.4 and 5.5 we can observe that Altruistic obtains approximately the same response time in the two settings, while the other lines are always more skewed.

To determine how the threshold is adhered to for different methods and workloads, Figure 5.6 shows the percentage of queries whose response time are within the threshold $T = 0.5$. From the figure, we observe that for Selfish, the percentage of queries meeting the 0.5 seconds deadline falls for workloads greater than 20 q/s, whereas Altruistic and Manic are able to keep this percentage above 90% for workloads up to 40 q/s.

We now analyse the effectiveness of the proposed methods. In Figure 5.7, we plot the NDCG@1000 values achieved for the different methods and workloads. These

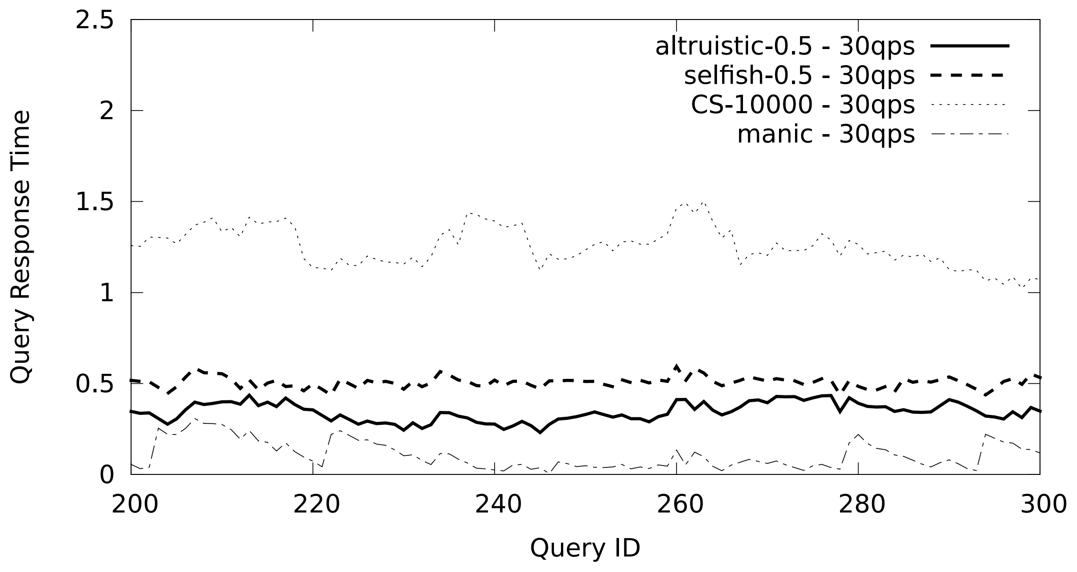

Figure 5.4: Query response time for 100 queries, arrival rate 30 q/s. $T = 0.5$

results are mirrored in Table 6.1, where we also show the significance of our results (paired t-test) compared to the Manic method (i.e. CS-1000). From the curves in Figure 5.7, we observe that the three baseline methods (Manic, CS-10000, and Perfectionist) have a constant effectiveness under any load conditions, as they do not apply any form of adaptation to load changes. On the other hand, Selfish and Altruistic adapt the processing strategy according to the load level, such that while effectiveness degrades further as load increases, effectiveness is still significantly better than applying Manic, for all of the tested workloads. This considerations can be also applied to Figure 5.8 that reports the NDCG@20.

Finally, to complete our answering of research question **RQ2**, effectiveness and efficiency results must be compared, by examining Figures 5.2 and 5.3 (query processing time) and Table 6.1 (NDCG@1000). For relatively low workloads, i.e. 5-20 q/s, the effectiveness of the Altruistic method is better than other approaches and is able to meet the time constraint of processing queries in less than $T = 0.5$ seconds.

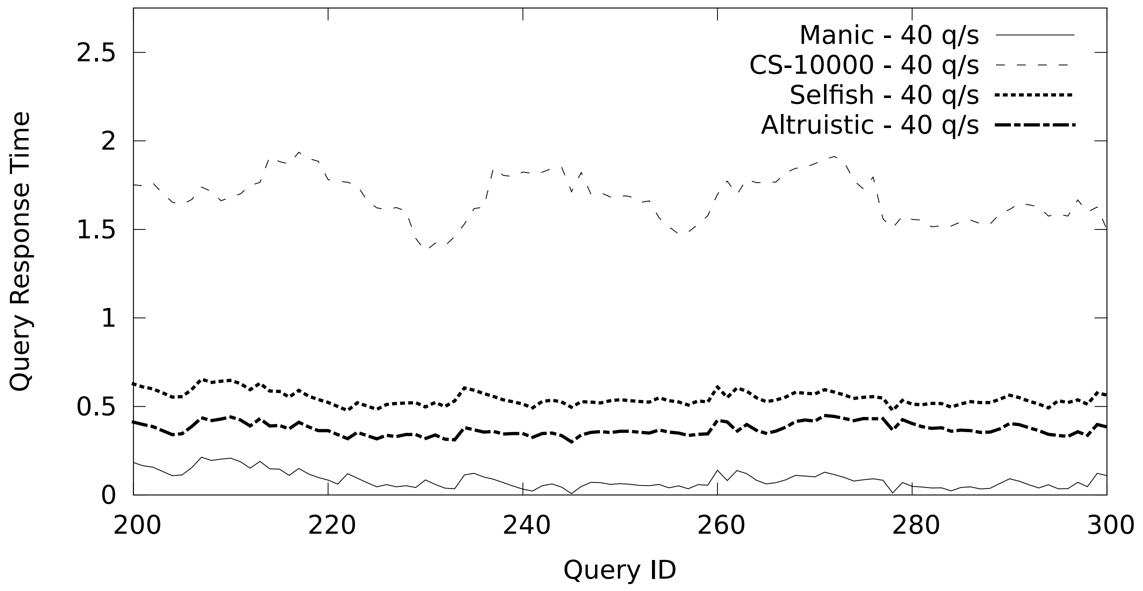

Figure 5.5: Query response time for 100 queries, arrival rate 40 q/s, $T = 0.5$.

On the other hand, for higher query workloads of 30 and 40 q/s, Altruistic is clearly the most effective method able to keep response times below T. This is explained in that Altruistic is able to fairly distribute query processing resources, enabling later queries in the queue to meet the deadline whilst still maintaining a significantly high effectiveness compared to the Manic method. For workloads greater than 40 q/s, none of our proposed methods are able to respect the time constraint. Nevertheless, it is worth remarking that Altruistic still attains higher effectiveness than Manic, which in turn has the same average response time.

To better validate our approaches we use also the *RBO* metric to measure how pruning affects the ranked list. Using this metric we are also able to test all of the 10,000 queries because we do not need the relevance judgments. *RBO* metric was firstly introduced in [114] to compare different ranked lists. We compare all the method proposed with a full evaluation strategy (`DAAT`) computed over our SE. We set the parameter $p = 0.9$ that means that the first 10 ranks have 86% of the weight of the evaluation[114]. Using the ranked list returned by the full processing strategy as our best possible results, the `DAAT` strategy achieve always a *RBO* of 1.

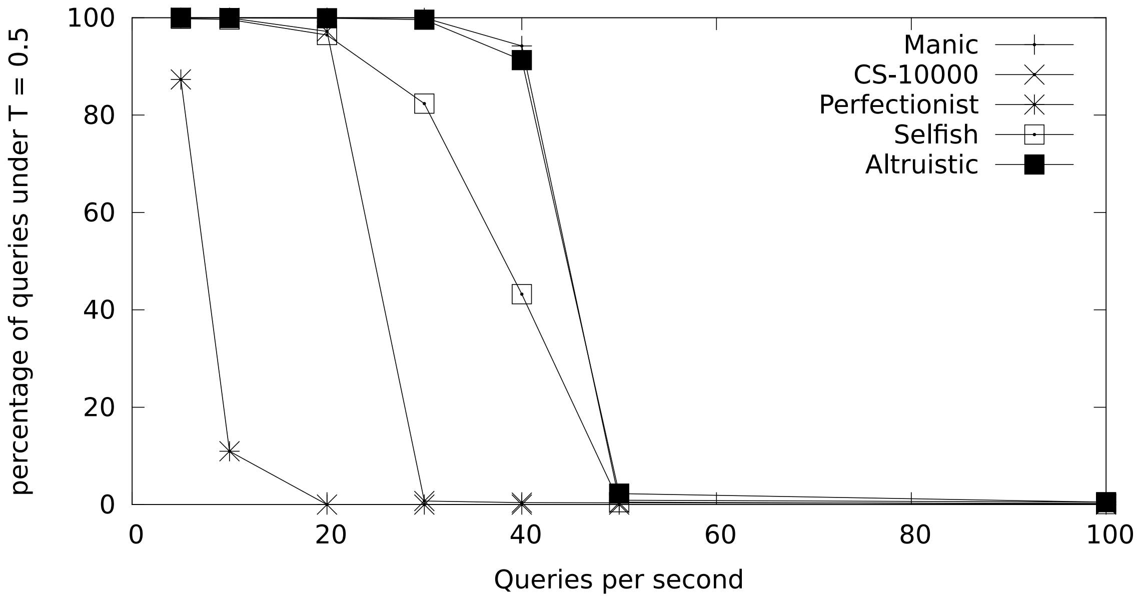

Figure 5.6: Percentage of queries achieving $T = 0.5$ for different methods and query workloads.

Manic method lost about 10% of RBO in any of the load settings. Also in this case, our methods obtain remarkable results. In particular Altruistic got an RBO 20% better in high workload in relation to Manic and achieve an RBO $=\sim 1$ for unloaded settings.

5.4.3 Effect of Parameter T on Response Times

Another advantage of our proposed framework is that the parameter T can be adjusted to tune the response time of the queries and adapt the retrieval strategy to our needs. In Figure 5.10 & 5.11, we show how the response times and NDCG@1000 with respect to three different thresholds, $T = \{0.1, 0.25, 0.5\}$. In particular, in addition to Manic, we show the results for Altruistic, which was the best performing in the previous section, with a suffix denoting the T value (e.g. Altruistic-0.25 for $T = 0.25$).

As expected, on examination of Figure 5.10, we find that by lowering the time threshold we also reduce the maximum sustainable load. In general, Altruistic at-

Method	5 q/s	10 q/s	20 q/s	30 q/s	40 q/s	50 q/s	100 q/s
Manic	0.316	0.316	0.316	0.316	0.316	0.316	0.316
CS-10000	0.348▲	0.348▲	0.348▲	0.348▲	0.348▲	0.348▲	0.348▲
Perfectionist	0.352▲	0.352▲	0.352▲	0.352▲	0.352▲	0.352▲	0.352▲
Selfish	0.352▲	0.352▲	0.347▲	0.339▲	0.329▲	0.318△	0.318△
Altruistic	0.352▲	0.352▲	0.350▲	0.345▲	0.335▲	0.324▲	0.324▲

Table 5.3: Effectiveness (NDCG@1000) for the different methods for $T = 0.5$. Statistically significant improvements vs. Manic, as measured by the paired t-test, are denoted by △ (p $<$ 0.05) and ▲ (p $<$ 0.01).

tains the highest effectiveness whilst being able to answer within the threshold to a relatively high query load. Indeed, Altruistic can achieve $T = 0.5$ with query loads upto 40 q/s, while $T = 0.25$ is achieved upto 30 q/s, and upto 10 q/s for $T = 0.1$ seconds. On the other hand, on examination of Figure 5.11, we observe that the effectiveness of Altruistic increases for larger T, particularly for low workloads. Indeed, for the challenging $T = 0.1$ threshold, even if at 5 q/s not all queries attain maximal effectiveness, we note that only a 5% difference[3] with the NDCG@1000 of Perfectionist (0.347 vs. 0.352).

Overall, from these results, we find that for very high query arrival rates and challenging time thresholds, it is impossible to attain T (for instance, $T = 0.1$ seconds at rates above 10 q/s). If the system must ensure that T is met, then the only alternative is to interrupt or drop late queries during processing. In the Chapter 6 we conduct experiments on the effectiveness of both dropping and interrupting query processing.

To summarise, for research question **RQ3** we find that in response to more challenging time thresholds T, the Altruistic method is able to adjust efficiency to facilitate servicing higher query loads within T than Perfectionist, whilst improving

[3]Due to the large number of queries, all differences are statistically significant for $p < 0.05$.

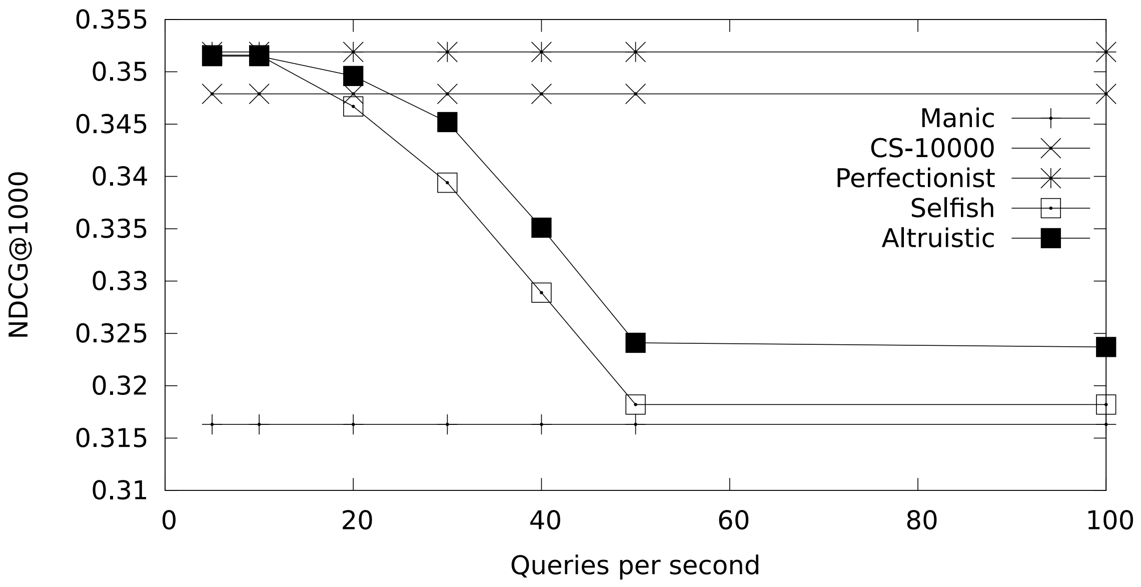

Figure 5.7: NDCG@1000 computed over the 687 queries in the test set, $T = 0.5$.

over the effectiveness of the uniform Manic method.

5.5 Conclusions

In this Chapter, we presented an innovative solution to the important problem of processing queries during times of high system load. In particular, we design a query processing framework relying on the two novel functions, namely Predict(), and Bound(). These use the predicted processing time for a query to calculate a processing time budget for that query, depending on both a global response time threshold, and the other queries waiting to be processed, while taking into account goals such as efficiency, effectiveness and fairness. This allows an appropriate dynamic pruning strategy to be selected for each query. For Predict(), we presented a regression-based model that can correctly predict the processing times of both DAAT and TAAT strategies with less than 10ms error in more than 90% of cases.

On the other hand, for Bound(), we proposed an Altruistic among other methods, which is able to fairly allocate processing resources across all queries currently

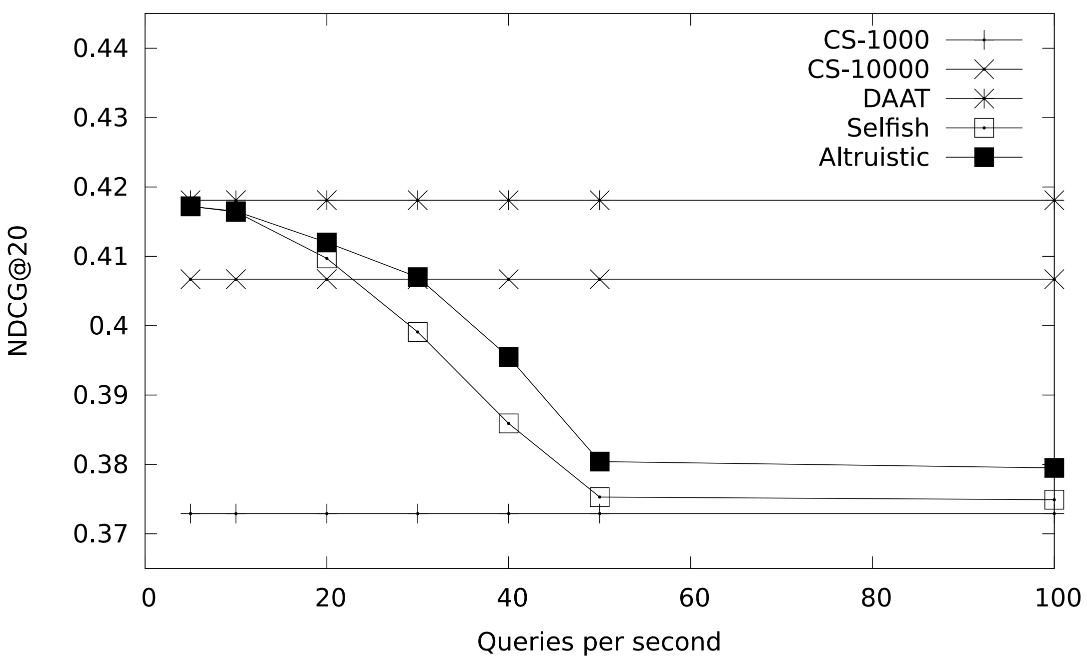

Figure 5.8: NDCG@20 computed over 687 queries in the test set, $T = 0.5$.

queued. Through extensive experiments on a standard test collection, Altruistic is shown capable of processing queries within various time thresholds T and with the smallest loss in terms of NDCG@1000. Finally, we show that Altruistic not only on average is able to stay within the time threshold T, but, under high load, is also the method that has the smallest percentage of queries for which the processing time exceeds T. Indeed, our results show that at a workload of 40 queries per second, Altruistic is able to meet a deadline of 0.5 seconds for 90% of queries (see Figure 5.6) while still attaining significantly high effectiveness (Table 6.1). In contrast, the next most effective Selfish method can only meet the deadline for 40% of queries.

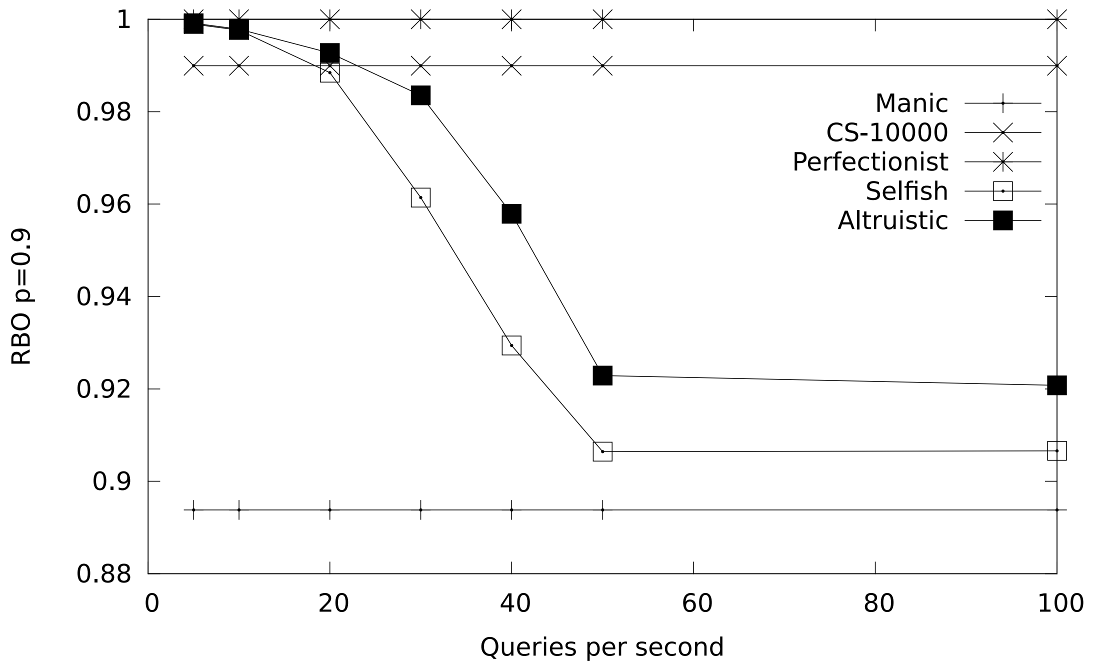

Figure 5.9: RBO p=0.9 computed over the 10,000 queries in the test set, $T = 0.5$.

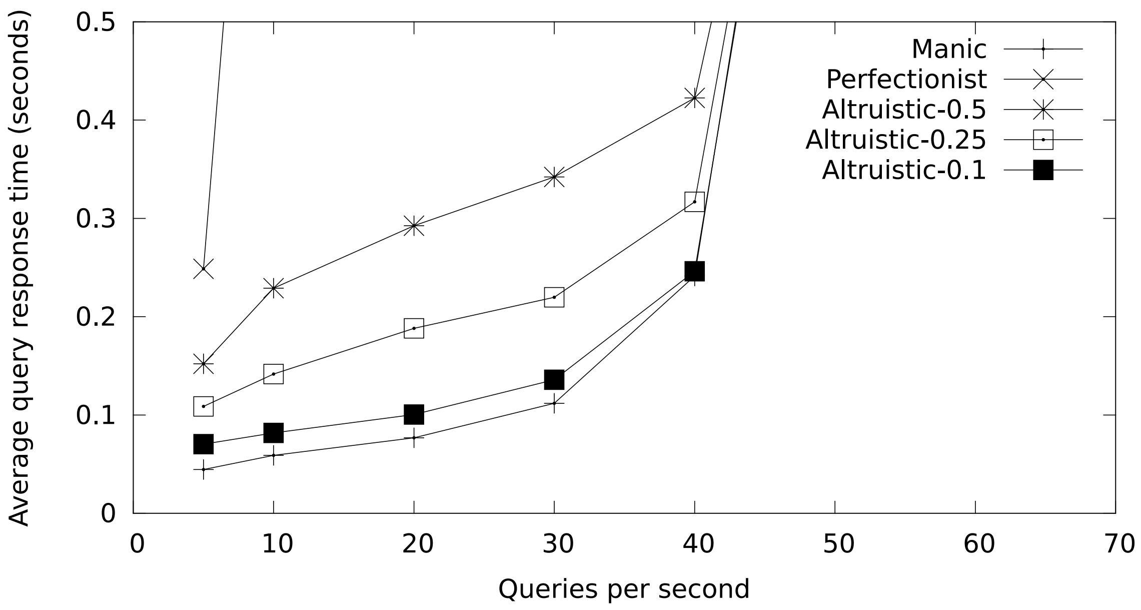

Figure 5.10: Average query response time in seconds for different T.

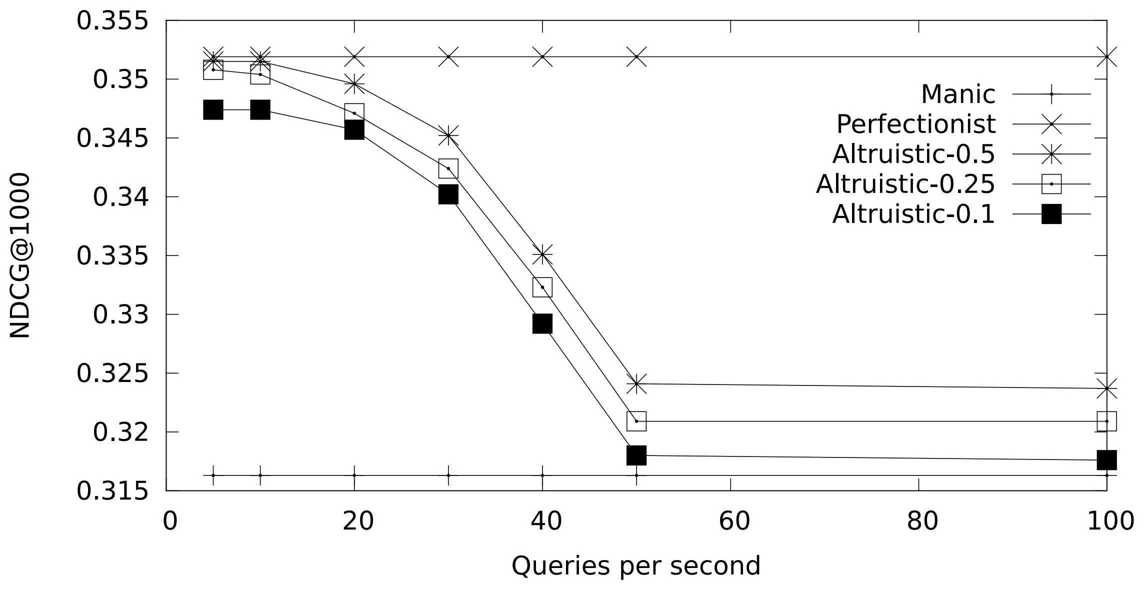

Figure 5.11: NDCG@1000 for different T.

6

Highly Loaded Search Engine

In this Chapter, we investigate the performances of different dropping solutions with the goal of maintaining the query response time under a specified time threshold. As explained in Chapter 5 it is possible to build efficiency predictors for `TAAT-CS` and `DAAT` to meet a time threshold when the web search engine is higly loaded. In Chapter 5 we also see that there is a breaking point where, using the fastest pruning strategy, the method proposed is not able to answer queries within the fixed time threshold.

In fact, in Figure 5.2, we see that Manic, the method that selects always the fastest strategy, for more than 40 queries per second, is not able to answer within the time threshold.

In this Chapter we present a method for always meeting the time threshold also when the strategy proposed in the previous Chapter fails. We compare naïve solutions with a novel method based on efficiency prediction that leverages a machine learning technique similar to the one in Chapter 5. We consider full `DAAT` as the baseline strategy. The methods presented work over the same distributed environment described previously and use the predictors to understand when a query must be dropped to avoid partial or incomplete results. We test our solution while varying the query arrival rate, from 5 to 100 queries per second (q/s), and measuring the query response time and the effectiveness in terms of NDCG@20 for all the methods proposed. We find that our strategy is able to sustain a workload of up to 100 queries per second with a relative degradation of only 15% with regard to the baseline methods, by reducing the number of queries dropped. Furthermore, while the baseline methods are not able to sustain moderately low workloads within the specified threshold, we are able to process up to 20 query per second with no statistically significant degradation in effectiveness.

6.1 Baselines

The aim of this Chapter is to study dropping techniques for keeping the response time of a highly loaded SE below a given time threshold. Our reference architecture is a distributed SE composed of a query broker that forwards requests to a pool of query servers. As depicted in Figure 6.1, we consider that each query server processes one query at a time. If a query server is processing a query, and other queries arrive, they are locally enqueued until they can be processed. Hence, the query response time for a query q is the sum of the time spent in the queue $wt(q)$ and processing time $pt(q)$.

The length of the queue at each query server depends on the query arrival rate and the processing time of the previous queries. In general, for higher query arrival rates, the query response time are increased, due to the longer waiting times. To ensure low query response times in a high load environment, we set a maximum processing threshold T that queries must be answered within.

We adopt two baseline strategies that define how a query server responds to a query for which T has elapsed during processing. The first strategy (hereinafter, `Drop`) interrupts the processing of a query and returns an empty list of results whenever the elapsed waiting and processing time exceeds T (i.e. when $wt(q)+pt(q) \geq T$). Similar to the `Drop` strategy, the second baseline (hereinafter, `Partial-Drop`) monitors the current query processing time and if it exceeds the deadline, it returns the partial results list that has been computed thus far (instead of dropping all results that have already been computed). Finally, we note that each query server acts independently from the other servers, in an autonomous fashion: each queue

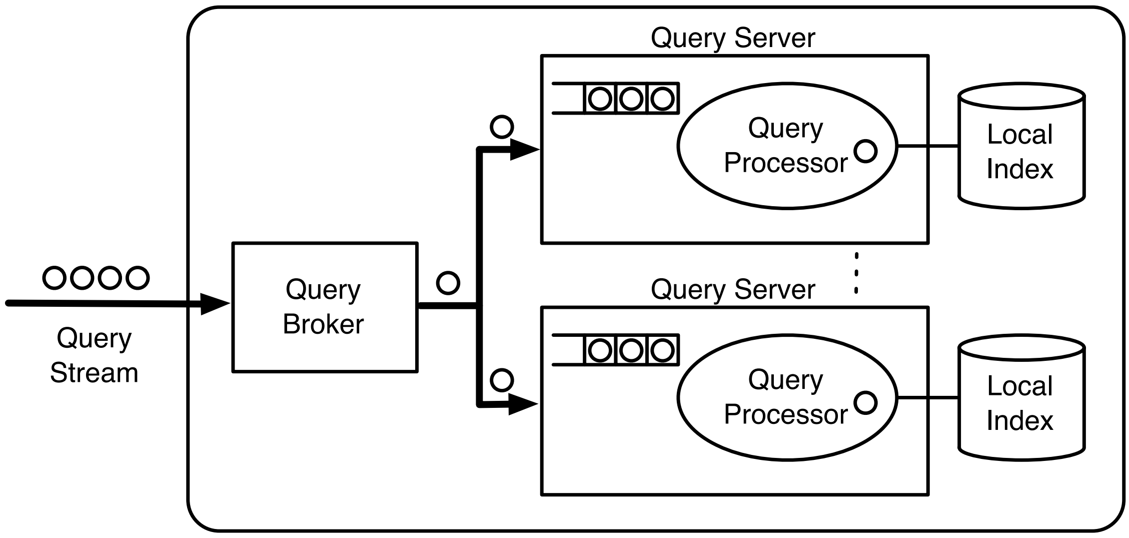

Figure 6.1: Architecture of our distributed Web SE.

is managed locally, and any dropping strategy is enforced locally. Hence, even if a query is fully processed on one query server, it can be (partially-)dropped on another server, causing the final results returned to the user to be partial in nature.

6.2 Prediction-based dropping

Unlike the previously described baselines, we aim to predict the response time of a query, to decide whether the query should be dropped or not before starting the actual processing phase. In particular, it has been shown that the response time of a query can be accurately predicted based on the total number of postings to be scored when using a full `DAAT`retrieval strategy [69].

In particular, we use the predicted response times at each query server to understand if the query can be processed within the remaining time on that server before T has elapsed. Given the predicted response time $\widehat{pt}(q)$ of query q, if the inequality $\widehat{pt}(q) \leq T - wt(q)$ does not hold, then the query is dropped before processing starts and the next query is processed from the queue. In this way, the query server does not consume processing resources for queries that cannot be fully (and effectively) completed within the remaining time until the threshold T has elapsed.

Query efficiency prediction for `DAAT` can be achieved using a machine learned algorithm using the total number of postings to be scored as feature [69]. To further improve the response time estimations, we use five additional features, which are listed in Table 5.1. In this case we use always `DAAT` and we do not need the four method dependent features. All features can be easily computed during the processing time without affecting the query response time.

The predicted response times are computed using a machine learned model implemented by a *linear regression* of the features. We learn a prediction model for each query server using the statistics of query terms on the local index. The coefficients of the regression model are computed by minimising the mean squared error on a set of training queries. In the following, we refer to our prediction-based dropping strategy as `ML-Drop`, and experiment to ascertain its properties in terms of efficiency and effectiveness.

6.3 Experimental Setup

The reseach questions we want to answer in this Chpter are:

1. What is the accuracy of our response time predictors? (Section 5.4.1)
2. What are the benefits of our `ML-Drop` strategy with respect to the two baseline strategies, `Drop` and `Partial-Drop`? (Section 6.4)

Since in this work we use `DAAT` as the retrival strategy for all of our methods, the answer to the first research question is already answered in the previous Chapter in the Section 5.4.1. The only difference is that in this work we do not need the part about `TAAT-CS` because we always use `DAAT` as the retrieval strategy.

The setting we use in this experiment is the same used in Section 5.3 except for the time threshold, we set here $T = 0.5s$ for all of the experiments. We choose this value because it is a reasonable time from the user perspective and at the same time using this time threshold, in our architecture, 98% of queries can be answered using the full `DAAT` strategy when the system is not heavy loaded.

Anyway we showed in the Section 5.4.3, that the approach we use works also when we vary the time threshold.

In the following experiments we compare our dropping strategy with two baseline strategies.

6.4 Dropping Strategies

We analyse the performance of the different query dropping strategies, namely `Drop`, `Partial-Drop` and `ML-Drop`. We compute the average query response time of the dropping strategies and we compare them to the full `DAAT` processing strategy without any dropping.

Figure 6.2 shows the average query response time vs. the number of queries per second (denoted q/s) received by the system. From the figure, we observe that using the full `DAAT` processing for all the queries implies an increasing query response time that is caused by congestion at the queues. However, all the other strategies (`Drop`, `Partial-Drop` and `ML-Drop`) manage to answer, on average, within the time constraint, as the superimposed curves show. As expected, the `Drop` and `Partial-Drop` strategies achieve this threshold, as they are both defined such that processing terminates at time T. In the case of our approach (`ML-Drop`), instead, respecting the time constraint means that our prediction models are able to identify queries to drop that cannot be processed within the time threshold.

Next, we examine the impact on effectiveness of the different processing methods. Figures 6.3 and 6.4 present effectiveness in terms of NDCG@20 and NDCG@1000, while Table 6.1 reports the same NDCG@20 values, in conjunction with statistical significance tests using the paired t-test.

As expected, full processing (`DAAT`) always obtains the best effectiveness, at the price of a higher query response time. The other strategies obtain an effectiveness dependent on the system load, since the number of dropped queries is impacted

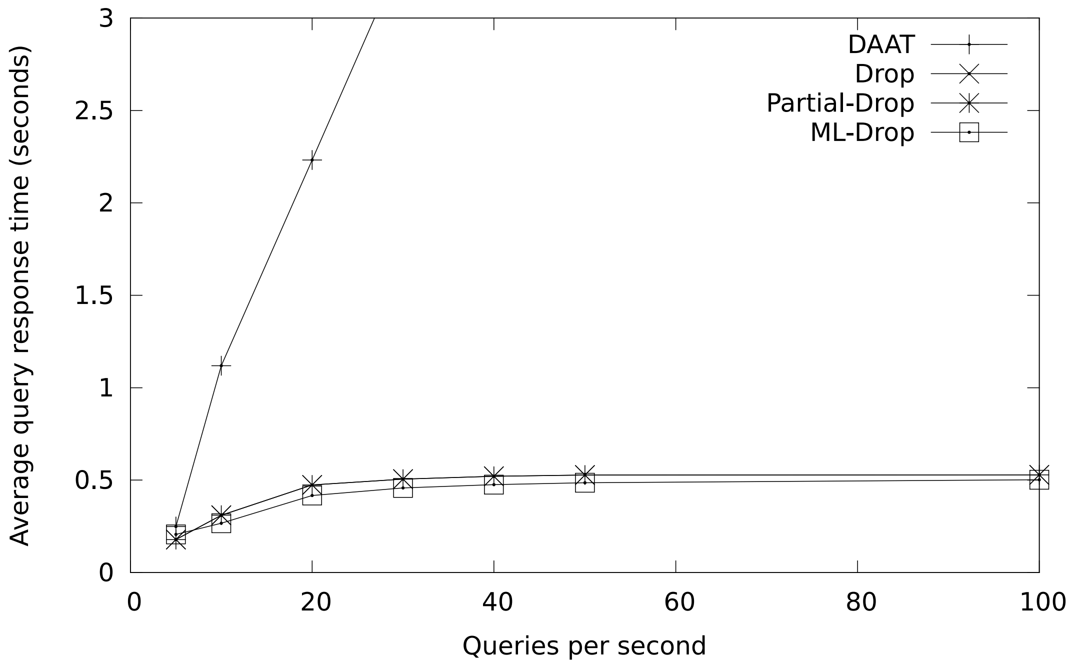

Figure 6.2: Average query response time (in seconds) for different dropping strategies.

by the remaining time for processing queries. This time is inversely proportional to the waiting time of the query itself. The least effective method is `Drop`: even though it achieves high effectiveness when the system is unloaded, NDCG@20 and NDCG@1000 decrease quickly as the load increases, because the processing of many queries cannot be finished within the permitted time. Consequently, these queries are dropped by the query server and the time spent is wasted, as no results are returned to the broker. The other baseline, `Partial-Drop`, obtains a better effectiveness in comparison to `Drop`. This is expected, because by returning partial results that have been computed within the limited processing time, some relevant results for some queries can be retrieved on average.

On the other hand, the effectiveness of `ML-Drop` is always higher than the two baselines. For instance, when queries arrive at a rate of 100 q/s, `ML-Drop` results in an effectiveness drop of 15% NDCG@20 (0.228 to 0.195, significant for $p < 0.05$),

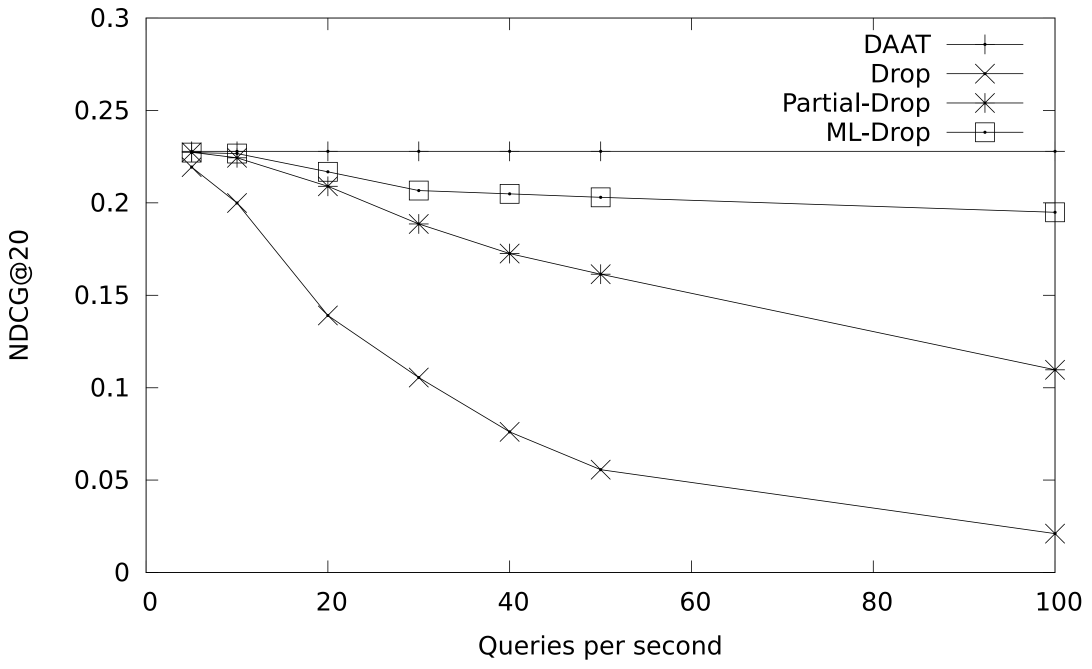

Figure 6.3: Effectiveness (NDCG@20).

compared to `Partial-Drop` which would result in a 52% drop in effectiveness, significant for $p < 0.01$. Similarly, for query arrival rates up to 20 q/s, `ML-Drop` exhibits no significant degradation in effectiveness, which is in contrast with both `Drop` and `Partial-Drop`. The gaps are more evident in the case of NDCG@1000 where the effectiveness of `Partial-Drop` is 50% worse than `ML-Drop` for query arrival rates about 100 q/s.

The relatively high effectiveness of `ML-Drop` compared to `Partial-Drop` can be explained as follows. In `Partial-Drop`, queries can partially be processed, expending valuable processing time while they would not likely not return any relevant documents. Instead, in `ML-Drop`, as queries which cannot be processed before T elapses are immediately discarded, leaving the potential for more queries to be fully processed.

To illustrate this, we analyse the number of queries *globally* dropped for the different methods. A query is globally dropped when it is dropped by all query

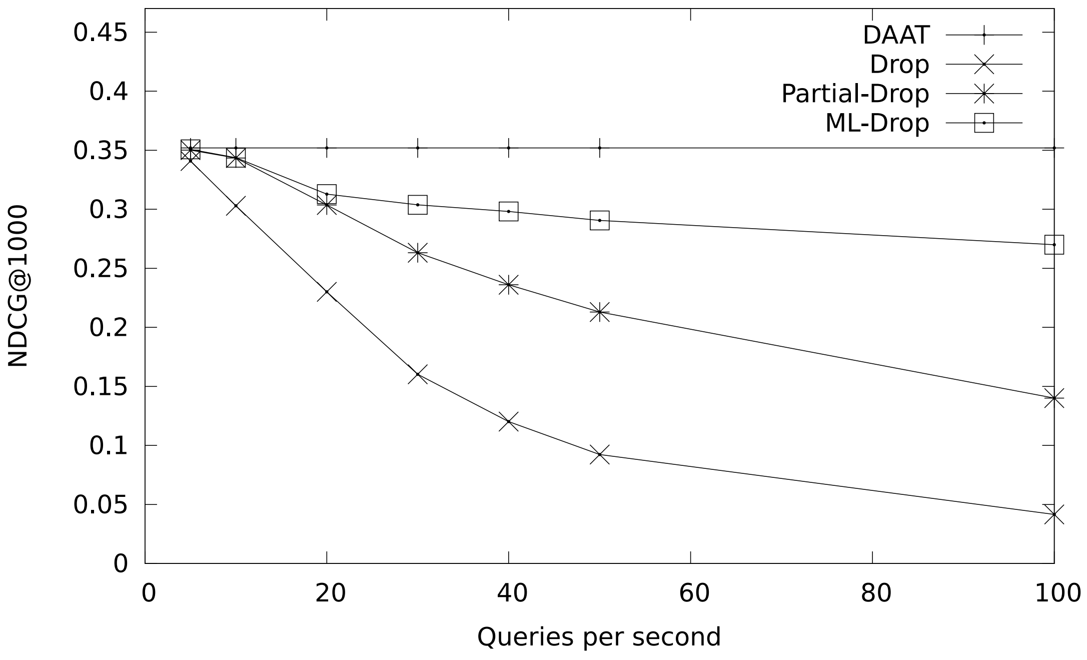

Figure 6.4: Effectiveness in terms of NDCG@1000, $T = 0.5s$.

servers processing it. Indeed, as query servers are independent, a query can be dropped only in a subset of the query servers. It is therefore possible that some queries have partial results even when the `Drop` strategy is used. In Figure 6.5, we show both the number of queries globally dropped and the number of queries that have partial results. In the case of `Partial-Drop`, the expiry of the time threshold can cause local partial results to be sent back to the broker. For high query loads, i.e., 100 q/s, this impacts about 90% of processed queries. For the same high arrival rate, the `Drop` strategy globally drops around 60% of queries while returning partial results for around 30% of queries. However, in the case of the `ML-Drop` strategy, the number of queries globally dropped or with partial results markedly decreases in relation to the other strategies.

Hence, in addressing our second research question, we find that the proposed `ML-Drop` strategy reduces the number of queries dropped under high load, resulting in improved effectiveness. Indeed, when 100 queries per second arrive, `ML-Drop` is

Method	5 q/s	10 q/s	20 q/s	30 q/s	40 q/s	50 q/s	100 q/s
`DAAT`	0.228	0.228	0.228	0.228	0.228	0.228	0.228
`Drop`	0.219 ▼	0.200 ▼	0.140 ▼	0.105 ▼	0.076 ▼	0.056 ▼	0.021 ▼
`Partial-Drop`	0.227	0.224 ▽	0.210 ▼	0.189 ▼	0.173 ▼	0.161 ▼	0.110 ▼
`ML-Drop`	0.227	0.227	0.217	0.207 ▽	0.205 ▽	0.203 ▽	0.195 ▽

Table 6.1: Effectiveness (NDCG@20) for the different methods. Statistically significant degradations vs. `DAAT`, as measured by the paired t-test, are denoted by ▽ ($p < 0.05$) and ▼ ($p < 0.01$).

able to answer up to 40% of the queries without effectiveness degradations, while for `Drop` and `Partial-Drop` strategies this happens for only 10% of queries.

6.5 Conclusion

In this Chapter, we analysed dropping and stopping methods for query processing in presence of an unsustainable workload. Our aim was to answer queries within a fixed time threshold, whilst maintaining overall effectiveness of the results. To address this goal, we used two baselines reportedly deployed by Web `SEs`, where a timer causes the query processing to be interrupted or the query to be dropped entirely when the time threshold expires. In addition, we propose a novel dropping method based on the predicted efficiency of queries. We test the proposed method on a distributed SE using 10,000 queries and a collection of 50 million documents, varying the number of queries per second. Our efficiency predictor models learned using linear-regression over six features are able to predict the query response time for `DAAT` with an error less than 10 ms in more than 93% of the cases. Using these predictors to select the queries to drop, the fixed time constraint can be achieved. We also compared the effectiveness of the results using relevance assessments from the TREC Million Query track. Our results showed that our method obtain up to 80% improvement in comparison to the most effective of the used baselines. Finally,

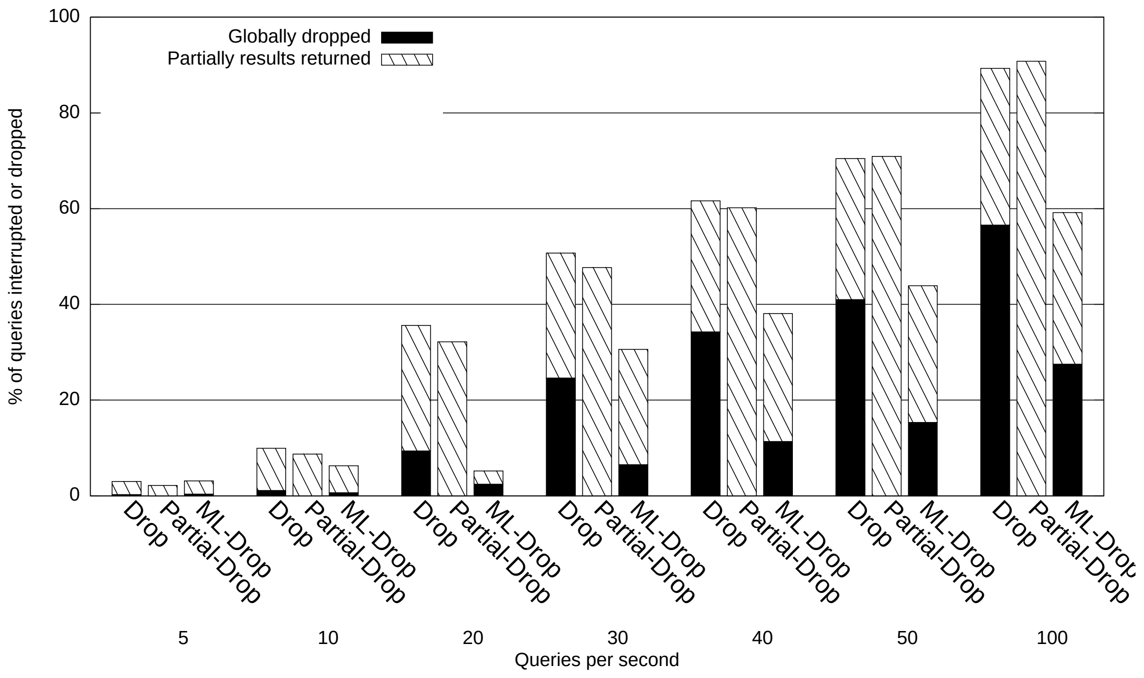

Figure 6.5: Percentage of globally dropped or partially evaluated queries.

we showed that our method decreases the number of dropped queries when the system is overloaded.

7 Conclusions

In this thesis we proposed different solutions to improve the efficiency and effectiveness of web search engines by exploiting the data stored into query logs. The aim of the thesis is to offer solutions for efficiently and effectively managing high query load conditions in a web search engine. The problem was tackled from various perspectives. From the first point of view, we proposed an effective query recommendation algorithm for the Query Shortcuts Problem (Chapter 4). Our query recommender aims not only at enhancing the SE effectiveness along with the SE user experience, but also the overall efficiency of the SE by reducing the average length of user sessions. The quality of query suggestions generated was assessed by evaluating the effectiveness in forecasting the users behaviour recorded in historical query logs, and on the basis of the results of a reproducible user study conducted on publicly-available, human-assessed data. The experimental evaluation conducted showed that our proposal remarkably outperforms two other state-of-the-art solutions, and that it could generate useful suggestions even for rare and never seen queries.

As future work we intend to evaluate the use of the whole head of the user session for producing query recommendations. Furthermore, we want to study if the sharing of the same final queries induces a sort of "clustering" of the queries composing the satisfactory user sessions. By studying this relationship, which lays at the basis of our query shortcut implementation, we could probably find newer and better ways to improve our methodology. Finally, it would be interesting to investigate how IR-like diversification algorithms (e.g., [2]) could be integrated in our query suggestion technique in order to obtain diversified query suggestions [67], [20].

From the second point, of view we presented two kind of strategies to avoid high query response time when the web search engine is highly loaded, one which dynamically selects adequate pruning strategies, presented in Chapter 5, and the other one which dynamically selects which queries to drop in order to maximize the overall effectiveness, presented in Chapter 6).

In particular, in Chapter 5, we designed a query processing framework relying on the two novel functions, namely Predict(), and Bound(). These use the predicted processing time for a query to calculate a processing time budget for that query, depending on both a global response time threshold, and the other queries waiting to be processed, while taking into account goals such as efficiency, effectiveness and fairness. This allows to select an appropriate dynamic pruning strategy to be selected for each query. For Predict(), we presented a regression-based model that can correctly predict the processing times of both `DAAT` and `TAAT` strategies with less than 10ms error in more than 90% of cases. On the other hand, for Bound(), we proposed the Altruistic method, which is able to allocate, in a fair manner, processing resources across all queries currently queued. After extensive experiments on a standard test collection, Altruistic is shown to be the most efficient, capable of processing queries within various time thresholds T and with the smallest loss in terms of NDCG@1000. Finally, we showed that not only Altruistic is, on average, able to stay within the time threshold T under high load, but it also has the smallest percentage of queries for which the processing time exceeds T. Indeed, our results show that at a workload of 40 queries per second, Altruistic is able to meet a deadline of 0.5 seconds for 90% of queries (see Figure 5.6) while still attaining significantly high effectiveness (Table 6.1). In contrast, the next most effective Selfish method can only meet the deadline for 40% of queries.

An interesting future work to be explored is the definition of a computational optimisation problem addressing Bound(), which could be adapted to a continuous stream of incoming queries. On the other hand, we believe that improved definitions for Predict() and Bound() could take into consideration the entire distributed search architecture, rather than each query server independently.

Moreover, in Chapter 6 we analysed techniques for dropping or interrupting queries under unsustainable workload. We used two baselines reportedly deployed by web SEs, where a timer causes the query processing to be interrupted or the query to be dropped entirely when the time threshold expires. In addition, we propose a novel dropping method based on the predicted efficiency of queries. We test the proposed method on a distributed SE using $10,000$ queries and a collection of 50 million documents, varying the number of queries per second. We also compared the effectiveness of the results using relevance assessments from the TREC Million Query track. Our results showed that our method obtained up to 80% improvement in comparison to the most effective of the baselines. Finally, we showed that our method decreases the number of dropped queries when the system is overloaded.

Future extensions of this work could encompass a dropping algorithm that changes the order of the queries queued on each query server. Alternatively, it is possible to give a different priority to different query servers, thereby combining the solution proposed by this work with collection selection methods.

The work on the efficiency of SEs, discussed in Chapter 5 and 6, opens several directions for future work. First of all it is possible to use both of the methods presented in Chapters 5 and 6 together and implement a heuristic method to obtain an improved quality of service. It is also possible to improve the methodology proposed exploiting a query scheduling algorithm to reorder queued queries, by first processing the ones which can be resolved faster and then the remaining ones.

Bibliography

[1] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE TKDE*, 17(6):734–749, 2005.

[2] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proc. WSDM'09*. ACM, 2009.

[3] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999.

[4] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42. ACM, 2001.

[5] Vo Ngoc Anh and Alistair Moffat. Impact transformation: effective and efficient web retrieval. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 3–10. ACM, 2002.

[6] Ricardo Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. Challenges on distributed web retrieval. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 6–20. IEEE, 2007.

[7] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search

engines. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 183–190. ACM, 2007.

[8] Ricardo Baeza-Yates, Carlos Hurtado, and Marcelo Mendoza. *Query Recommendation Using Query Logs in Search Engines*, volume 3268/2004 of *LNCS*. Springer Berlin / Heidelberg, November 2004.

[9] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Publishing Company, USA, 2nd edition, 2008.

[10] Ricardo Baeza-Yates and Alessandro Tiberi. Extracting semantic relations from query logs. In *Proc. KDD'07*. ACM, 2007.

[11] Evelyn Balfe and Barry Smyth. Improving web search through collaborative query recommendation. In *Proc. ECAI'04*. IOS Press, 2004.

[12] Ranieri Baraglia, Fidel Cacheda, Victor Carneiro, Diego Fernandez, Vreixo Formoso, Raffaele Perego, and Fabrizio Silvestri. Search shortcuts: a new approach to the recommendation of queries. In *Proc. RecSys'09*. ACM, 2009.

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

[14] Doug Beeferman and Adam Berger. Agglomerative clustering of a search engine query log. In *Proc. KDD'00*. ACM, 2000.

[15] Steven M Beitzel, Eric C Jensen, David D Lewis, Abdur Chowdhury, and Ophir Frieder. Automatic classification of web queries using very large unlabeled query logs. *ACM Transactions on Information Systems (TOIS)*, 25(2):9, 2007.

[16] Paolo Boldi, Francesco Bonchi, Carlos Castillo, Debora Donato, Aristides Gionis, and Sebastiano Vigna. The query-flow graph: model and applications. In *Proc. CIKM'08*. ACM, 2008.

[17] Paolo Boldi, Francesco Bonchi, Carlos Castillo, Debora Donato, and Sebastiano Vigna. Query suggestions using query-flow graphs. In *Proc. WSCD'09*. ACM, 2009.

[18] Paolo Boldi, Francesco Bonchi, Carlos Castillo, and Sebastiano Vigna. From 'dango' to 'japanese cakes': Query reformulation models and patterns. In *Proc. WI'09*. IEEE, September 2009.

[19] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: a scalable fully distributed web crawler. *Software: Practice and Experience*, 34:2004, 2003.

[20] Ilaria Bordino, Carlos Castillo, Debora Donato, and Aristides Gionis. Query similarity by projecting the query-flow graph. In *Proc. SIGIR'10*. ACM, 2010.

[21] Daniele Broccolo, Craig Macdonald, Salvatore Orlando, Iadh Ounis, Raffaele Perego, Fabrizio Silvestri, and Nicola Tonellotto. Load-sensitive selective pruning for distributed search. In *CIKM 2013*, 2013.

[22] Daniele Broccolo, Craig Macdonald, Salvatore Orlando, Iadh Ounis, Raffaele Perego, Fabrizio Silvestri, and Nicola Tonellotto. Query processing in highly-loaded search engines. In *String Processing and Information Retrieval*, pages 49–55. Springer International Publishing, 2013.

[23] Daniele Broccolo, Lorenzo Marcon, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Generating suggestions for queries in the long tail with an inverted index. *Information Processing & Management*, 2011.

[24] Daniele Broccolo, Lorenzo Marcon, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Generating suggestions for queries in the long tail with

an inverted index. *Information Processing & Management*, 48(2):326–339, 2012.

[25] Andrei Broder, Peter Ciccolo, Evgeniy Gabrilovich, Vanja Josifovski, Donald Metzler, Lance Riedel, and Jeffrey Yuan. Online expansion of rare queries for sponsored search. In *Proc. WWW'09*. ACM, 2009.

[26] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In Henrique Paques, Ling Liu, and David Grossman, editors, *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 426–434, New York, NY, USA, 2003. ACM.

[27] Andrei Z Broder, Marcus Fontoura, Evgeniy Gabrilovich, Amruta Joshi, Vanja Josifovski, and Tong Zhang. Robust classification of rare queries using web knowledge. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 231–238. ACM, 2007.

[28] Andrei Z. Broder, Marcus Fontoura, Evgeniy Gabrilovich, Amruta Joshi, Vanja Josifovski, and Tong Zhang. Robust classification of rare queries using web knowledge. In *Proc. SIGIR'07*. ACM, 2007.

[29] Brendon Cahoon, Kathryn S McKinley, and Zhihong Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems (TOIS)*, 18(1):1–43, 2000.

[30] James P Callan, Zhihong Lu, and W Bruce Croft. Searching distributed collections with inference networks. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 21–28. ACM, 1995.

[31] B Barla Cambazoglu, Vassilis Plachouras, and Ricardo Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 411–418. ACM, 2009.

[32] B Barla Cambazoglu, Emre Varol, Enver Kayaaslan, Cevdet Aykanat, and Ricardo Baeza-Yates. Query forwarding in geographically distributed search engines. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 90–97. ACM, 2010.

[33] Berkant Barla Cambazoglu and Ricardo Baeza-Yates. Scalability challenges in web search engines. In Massimo Melucci and Ricardo Baeza-Yates, editors, *Advanced Topics in Information Retrieval*, volume 33 of *The Information Retrieval Series*, pages 27–50. Springer Berlin Heidelberg, 2011.

[34] Berkant Barla Cambazoglu and Ricardo Baeza-Yates. Scalability challenges in web search engines. In Massimo Melucci and Ricardo Baeza-Yates, editors, *Advanced Topics in Information Retrieval*, volume 33 of *The Information Retrieval Series*, pages 27–50. Springer Berlin Heidelberg, 2011.

[35] John Canny. Collaborative filtering with privacy via factor analysis. In *Proceedings of the 25th annual international ACM SIGIR 2002 Conference*, pages 238–245, New York, NY, USA, 2002. ACM.

[36] Ben Carterette, Virgiliu Pavlu, Hui Fang, and Evangelos Kanoulas. Million query track 2009 overview. In Ellen M. Voorhees and Lori P. Buckland, editors, *TREC*, volume Special Publication 500-278. National Institute of Standards and Technology (NIST), 2009.

[37] Nick Craswell, Dennis Fetterly, and Marc Najork. The power of peers. In Paul Clough, Colum Foley, Cathal Gurrin, Gareth J.F. Jones, Wessel Kraaij, Hyowon Lee, and Vanessa Murdoch, editors, *ECIR '11: Advances in Information*

Retrieval, 33rd European Conference on IR Research, Heidelberg, Germany, 2011. Springer.

[38] J Shane Culpepper, Matthias Petri, and Falk Scholer. Efficient in-memory top-k document retrieval. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 225–234. ACM, 2012.

[39] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM.

[40] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In Ricardo A. Baeza-Yates, Paolo Boldi, Berthier A. Ribeiro-Neto, and Berkant Barla Cambazoglu, editors, *Proceedings of the Second International Conference on Web Search and Web Data Mining, WSDM 2009*, New York, NY, USA, 2009. ACM.

[41] Doug Downey, Susan Dumais, and Eric Horvitz. Heads and tails: studies of web search with common and rare queries. In *Proc. SIGIR'07*. ACM, 2007.

[42] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems (TOIS)*, 24(1):51–78, 2006.

[43] Bruno M. Fonseca, Paulo Golgher, Bruno Pôssas, Berthier Ribeiro-Neto, and Nivio Ziviani. Concept-based interactive query expansion. In *Proc. CIKM'05*. ACM, 2005.

[44] Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Zien. Evaluation strategies for top-k queries over memory-

resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

[45] Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Y. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *PVLDB*, 4(12):1213–1224, 2011.

[46] E Fox, D Harman, R Baeza-Yates, and W Lee. *Inverted files.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[47] Ana Freire, Craig Macdonald, Nicola Tonellotto, Iadh Ounis, and Fidel Cacheda. Scheduling queries across replicas. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 1139–1140. ACM, 2012.

[48] Ana Freire, Craig Macdonald, Nicola Tonellotto, Iadh Ounis, and Fidel Cacheda. Hybrid query scheduling for a replicated search engine. In *Advances in Information Retrieval*, pages 435–446. Springer, 2013.

[49] Ahmed Hassan, Rosie Jones, and Kristina Lisa Klinkner. Beyond dcg: user behavior as a predictor of a successful search. In *Proc. WSDM'10*, pages 221–230, New York, NY, USA, 2010. ACM.

[50] David Hawking. Efficiency/effectiveness trade-offs in query processing (from theory into practice workshop, 1998 sigir conf.). In *ACM SIGIR Forum*, volume 32, pages 16–22. ACM, 1998.

[51] Thomas Hofmann. Latent semantic models for collaborative filtering. *ACM Trans. Inf. Syst.*, 22:89–115, January 2004.

[52] Anni Järvelin, Antti Järvelin, and Kalervo Järvelin. s-grams: Defining generalized n-grams for information retrieval. *IPM*, 43(4):1005 – 1019, 2007.

[53] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *Parallel and Distributed Systems, IEEE Transactions on*, 6(2):142–153, 1995.

[54] Simon Jonassen and Svein Erik Bratsberg. A combined semi-pipelined query processing architecture for distributed full-text retrieval. In *Web Information Systems Engineering–WISE 2010*, pages 587–601. Springer, 2010.

[55] Simon Jonassen, B. Barla Cambazoglu, and Fabrizio Silvestri. Prefetching query results and its impact on search engines. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '12, pages 631–640, New York, NY, USA, 2012. ACM.

[56] K. Sparck Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. In *Information Processing and Management*, pages 779–840, 2000.

[57] Rosie Jones and Kristina Lisa Klinkner. Beyond the session timeout: automatic hierarchical segmentation of search topics in query logs. In *Proc. CIKM'08*. ACM, 2008.

[58] Rosie Jones, Benjamin Rey, Omid Madani, and Wiley Greiner. Generating query substitutions. In *Proc. WWW'06*. ACM, 2006.

[59] Marcin Kaszkiel and Justin Zobel. Term-ordered query evaluation versus document-ordered query evaluation for large document databases. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 343–344. ACM, 1998.

[60] Tessa Lau and Eric Horvitz. Patterns of search: analyzing and modeling web query refinement. *COURSES AND LECTURES-INTERNATIONAL CENTRE FOR MECHANICAL SCIENCES*, pages 119–128, 1999.

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

[62] Nicholas Lester, Alistair Moffat, William Webber, and Justin Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Web Information Systems Engineering–WISE 2005*, pages 470–477. Springer, 2005.

[63] Nicholas Lester, Alistair Moffat, William Webber, and Justin Zobel. Space-limited ranked query evaluation using adaptive pruning. In Anne H. H. Ngu, Masaru Kitsuregawa, Erich J. Neuhold, Jen-Yao Chung, and Quan Z. Sheng, editors, *Proceedings of the 6th International Conference on Web Information Systems Engineering (WISE 05)*, volume 3806 of *Lecture Notes in Computer Science*, pages 470–477, New York, NY, USA, November 2005. Springer.

[64] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri, and Gabriele Tolomei. Identifying task-based sessions in search engine query logs. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 277–286. ACM, 2011.

[65] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri, and Gabriele Tolomei. Identifying task-based sessions in search engine query logs. In *Proc. WSDM'11*, pages 277–286, New York, NY, USA, 2011. ACM.

[66] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri, and Gabriele Tolomei. Discovering tasks from search engine query logs. *ACM Trans. Inf. Syst.*, 31(3):14, 2013.

[67] Hao Ma, Michael R. Lyu, and Irwin King. Diversifying query suggestion results. In *Proc. AAAI'10*. AAAI, 2010.

[68] Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. Effect of dynamic pruning safety on learning to rank effectiveness. In *Proceedings of the 35th inter-*

national ACM SIGIR conference on Research and development in information retrieval, SIGIR '12, pages 1051–1052, New York, NY, USA, 2012. ACM.

[69] Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. Learning to predict response times for online query scheduling. In *Proceedings of SIGIR 2012*, pages 621–630, 2012.

[70] Mauricio Marin and Carlos Gomez. Load balancing distributed inverted files. In *Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 57–64. ACM, 2007.

[71] Mauricio Marin, Carlos Gomez-Pantoja, Senen Gonzalez, and Veronica Gil-Costa. Scheduling intersection queries in term partitioned inverted files. In *Euro-Par 2008–Parallel Processing*, pages 434–443. Springer, 2008.

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

[73] Qiaozhu Mei, Dengyong Zhou, and Kenneth Church. Query suggestion using hitting time. In *Proc. CIKM'08*. ACM, 2008.

[74] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355. ACM, 2006.

[75] Alistair Moffat, William Webber, Justin Zobel, and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231, 2007.

[76] Alistair Moffat, William Webber, Justin Zobel, and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10:205–231, June 2007.

[77] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *Transactions on Information Systems*, 14(4):349–379, 1996.

[78] ALISTAIR AUTOR MOFFAT, Timothy C Bell, et al. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[79] Iadh Ounis, Gianni Amati, Vassilis Plachouras, Ben He, Craig Macdonald, and Christina Lioma. Terrier: A high performance and scalable information retrieval platform. In *Proceedings of the OSIR Workshop 2006*, pages 18–25, France, August 2006. École Nationale Supérieure des mines de Saint-Etienne.

[80] Seda Ozmutlu, H Cenk Ozmutlu, and Amanda Spink. Multitasking web searching and implications for design. *Proceedings of the American Society for Information Science and Technology*, 40(1):416–421, 2003.

[81] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale*, volume 6, pages 1–7. Citeseer, 2006.

[82] A. Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD Cup and Workshop*, volume 2007. Citeseer, 2007.

[83] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, 47(10):749–764, 1996.

[84] Filip Radlinski and Thorsten Joachims. Query chains: learning to rank from implicit feedback. In *Proc. KDD'05*. ACM Press, 2005.

[85] Jasson D. M. Rennie and Nathan Srebro. Fast maximum margin matrix factorization for collaborative prediction. In *Proceedings of the 22nd ICML 2005 Conference*, pages 713–719, New York, NY, USA, 2005. ACM.

[86] Berthier A Ribeiro-Neto and Ramurti A Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM conference on Digital libraries*, pages 182–190. ACM, 1998.

[87] Knut Magne Risvik, Trishul Chilimbi, Henry Tan, Karthik Kalyanaraman, and Chris Anderson. Maguro, a system for indexing and searching over very large text collections. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 727–736. ACM, 2013.

[88] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '94, pages 232–241, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[89] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, 2009.

[90] G. Salton and M.J. McGill. *Introduction to modern information retrieval.* McGraw-Hill computer science series. McGraw-Hill, 1983.

[91] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[92] Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *Proc. WWW'01.* ACM, 2001.

[93] Upendra Shardanand and Pattie Maes. Social information filtering: algorithms for automating "word of mouth". In *Proc. SIGCHI'95.* ACM, 1995.

[94] Milad Shokouhi. Central-rank-based collection selection in uncooperative distributed information retrieval. In *Advances in Information Retrieval*, pages 160–172. Springer, 2007.

[95] Luo Si and Jamie Callan. Relevant document distribution estimation method for resource selection. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 298–305. ACM, 2003.

[96] Luo Si and Jamie Callan. Unified utility maximization framework for resource selection. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 32–41. ACM, 2004.

[97] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a very large web search engine query log. In *ACm SIGIR Forum*, volume 33, pages 6–12. ACM, 1999.

[98] Fabrizio Silvestri. Mining query logs: Turning search usage data into knowledge. *Foundations and Trends in Information Retrieval*, 4(1âĂŤ2):1–174, 2010.

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

[100] Fabrizio Silvestri and Rossano Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1219–1228. ACM, 2010.

[101] Barry Smyth. A community-based approach to personalizing web search. *Computer*, 40(8):42–50, 2007.

[102] Barry Smyth, Evelyn Balfe, Oisin Boydell, Keith Bradley, Peter Briggs, Maurice Coyle, and Jill Freyne. A live-user evaluation of collaborative web search. In *IJCAI*, 2005.

[103] Yang Song and Li-wei He. Optimal rare query suggestion with implicit user feedback. In *Proc. WWW'10*. ACM, 2010.

[104] Amanda Spink, Bernard J. Jansen, Dietmar Wolfram, and Tefko Saracevic. From E-Sex to E-Commerce: Web Search Changes. *Computer*, 35(3):107–109, March 2002.

[105] Paul Thomas and Milad Shokouhi. Sushi: scoring scaled samples for server selection. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 419–426. ACM, 2009.

[106] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. Efficient and effective retrieval using selective pruning. In *Proceedings of WSDM 2013*, pages xxx–yyy, 2013.

[107] Frederik Transier and Peter Sanders. Compressed inverted indexes for in-memory search engines. In *ALENEX*, pages 3–12, 2008.

[108] Andrew Trotman. Learning to rank. *Information Retrieval*, 8(3):359–381, 2005.

[109] Howard Turtle and James Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, 31(6):831–850, 1995.

[110] Howard Turtle and James Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[111] L. Ungar and D. Foster. Clustering methods for collaborative filtering. In *Proceedings of the Workshop on Recommendation Systems*. AAAI Press, Menlo Park California, 1998.

[112] Jun Wang, Arjen P. de Vries, and Marcel J. T. Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proc. SIGIR'06*. ACM, 2006.

[113] Lidan Wang, Jimmy Lin, and Donald Metzler. Learning to efficiently rank. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 138–145, New York, NY, USA, 2010. ACM.

[114] William Webber, Alistair Moffat, and Justin Zobel. A similarity measure for indefinite rankings. 28(4):20:1–20:??, November 2010.

[115] Ji-Rong Wen, Jian-Yun Nie, and Hong-Jiang Zhang. Clustering user queries of a search engine. In *Proc. WWW'01*. ACM, 2001.

[116] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[117] Jiangong Zhang and Torsten Suel. Optimized inverted list assignment in distributed search engine architectures. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[118] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)*, 23(4):453–490, 1998.