Load-Sensitive Selective Pruning for Distributed Search

Daniele Broccolo[1,3], Craig Macdonald[2], Salvatore Orlando[1,3],
Iadh Ounis[2], Raffaele Perego[1], Fabrizio Silvestri[1,4], Nicola Tonellotto[1]
[1] National Research Council of Italy [2] University of Glasgow
[3] Ca' Foscari University of Venice [4] Yahoo! Research, Barcelona, Spain
{firstname.lastname}@isti.cnr.it[1], {craig.macdonald, iadh.ounis}@glasgow.ac.uk[2]
silvestr@yahoo-inc.com[4]

ABSTRACT

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, 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, leading to disappointed users. In this paper, 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. 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, our search system is able to answer 90% of queries within half a second during times of high query volume.

Categories and Subject Descriptors: H.3.3 [Information Storage & Retrieval]: Information Search & Retrieval

Keywords: Query Efficiency Prediction, Scheduling

1. INTRODUCTION

Commercial Web search engines are expected to process user queries under tight response time constraints while being able to operate under heavy query traffic loads. Queries that cannot be processed within their time constraint experience degraded result quality [5]. Operating under these conditions requires building a very large infrastructure involving thousands of computers and making continuous investments to maintain this infrastructure [3]. Hence, optimising the efficiency of Web search engines is important to reduce their infrastructure costs.

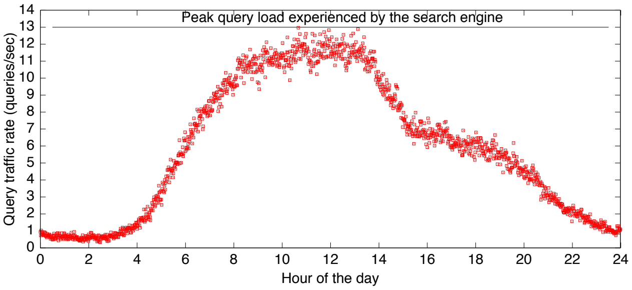

Figure 1: The query distribution of a 24 hours time span covering 1st May 2006 from MSN query log.

The user query volume typically received by a Web search engine is illustrated in Figure 1, showing how the rate of queries received can vary through the course of a day. In order to guarantee that each query is processed with subsecond response times, the computing/communication infrastructure has to support worst-case query volume, which reaches its maximum during the day time (about 13 queries per second from 10:00 to 14:00 in the workload depicted in Figure 1), typically around midday. Hence, the typical approach taken by Web search engines is to deploy a distributed search architecture [8]. According to this architecture, the servicing of a search query uses many query servers, each in charge of a partition of the global index. When a query reaches one of these servers, it is *processed* immediately if the server is idle, otherwise it is placed in a *queue* waiting for processing. Hence, the *completion time* of a query at each server includes both a *waiting time* and a *processing time*, in turn relying on the processing strategy.

We argue that in order to realise a Web search engine that can answer a query within a given deadline, there are three options for how the system can respond in presence of high query volume. Firstly, if the system is not able to reduce the processing time of each query and the scheduling of the queries cannot be modified, queries with long processing times might simply be dropped – resulting in error pages being returned to users – or interrupted – where a partial result list is presented. While this technique does guarantee high quality results for a (possibly small) subset of queries, it is unfortunate that this is the only possible choice when the query volume cannot be sustained by a given search engine infrastructure. The second option, discussed in recent works [10, 14], is based on *query efficiency predictions*, along with suitable scheduling algorithms, to re-order the

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.
Copyright 2013 ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505699 .

query queue. The aim of this technique is to reduce the overall queueing time of queries, thus increasing the query throughput. A third, alternative option, proposed in this paper, is to dynamically select a suitable query processing (retrieval) strategy to process the queries to satisfy the per-query deadlines, thus reducing the query processing time. In particular, when deciding between processing strategies, our proposed approach considers the time necessary to satisfy the per-query deadlines for all queries queued after the current query. In this way it can allocate available resources *fairly* across the waiting queries.

On the other hand, a strategy that reduces the processing time of a query has obvious drawbacks: we need to exploit approximate processing strategies, such as *dynamic pruning* [4, 16, 20], that can reduce the quality of the query results. However, pruning can be applied *selectively*, on a per-query basis [19], depending on the expected processing time of the query and the status of the search engine. In this work, we go further, as our novel scheduling methodology can selectively adopt dynamic pruning processing strategies only when the system is experiencing high workloads, thereby trading off some effectiveness to ensure efficiency.

In summary, this paper argues that the effectiveness of search results can be maintained whilst meeting completion time constraints by choosing an appropriate pruning strategy to use for each query to be processed by a given server. In particular, our method can examine not just the current query, but also the other queries queued for processing. Hence, the contributions of this paper are two-fold: We propose a load-sensitive selective pruning framework for bounding the permitted processing time of a query, which consider goals such as meeting a time threshold, effectiveness and fairness to other queries waiting to be processed; Moreover, to support our proposed framework, we propose an accurate approach for query efficiency prediction of term-at-a-time dynamic pruning strategies. Our experiments show that the proposed framework is able to produce results of quality comparable to that of a search system that does not bound query processing time, while at high query workloads the system can still respond to queries in less than a predetermined time threshold. For instance, when 40 queries per second are arriving at the search engine, our framework is able to answer 90% of queries within 0.5 seconds with a 5% drop in results quality compared to an effective processing strategy for which 0% of queries meet the time threshold.

The remainder of this paper is structured as follows: In Section 2, we introduce the necessary preliminaries by discussing the context of our work; Section 3 discusses related work in efficient retrieval; In Section 4, we propose our framework for load-sensitive selective pruning; Section 5 discusses the processing strategies we deploy, and how their response times can be accurately predicted; Section 6 defines the experimental setup for the evaluation that follows in Section 7; We provide concluding remarks in Section 8.

2. PRELIMINARIES

Web search engines have to manage huge quantities of documents while achieving the goal of effectively answering users' queries, and doing so efficiently – i.e., within a fraction of a second. To achieve this multi-objective goal despite the large size of the Web, the corpus of documents the search engine must manage are partitioned into sub-collections that are each manageable by a single machine. This results in several *query servers* engaged in answering a user's query, each of them storing the *index shard* [3] for a subset of the index built on the corpus.

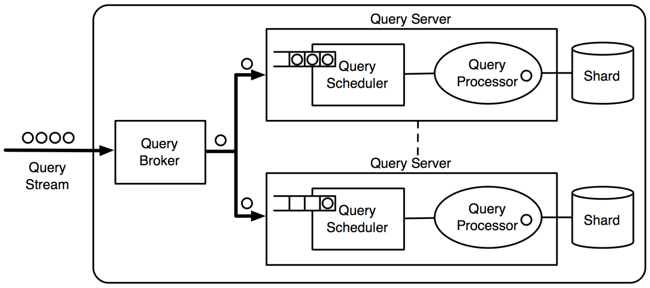

Figure 2: Our reference architecture of a distributed search engine node (based on [5]).

Without loss of generality, in this work we assume a distributed search engine where data are distributed according to a *document partitioning* strategy [2]. 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. 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 [5].

Figure 2 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 [18]). Strategies such as dynamic pruning [4, 16, 20] can be used to process queries in an efficient manner on each query server. In this work, we consider document-sorted indices, as used by at least one major search engine [8]. Other efficient retrieval techniques such as frequency-sorted [20] or impact-sorted indices [1] 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 [15], perhaps – as suggested by Lester *et al.* [12] – 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.* [7] who highlighted that disjunctive semantics does not produce significantly different high-precision effectiveness compared to conjunctive retrieval.

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 [14]. 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 this paper, 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 [11]. Indeed, in our proposed framework, we set the maximum query processing time to a carefully chosen value (see Section 4), 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 7.2). Our proposed framework exploits novel machine learning models for estimating processing time under different processing strategies.

3. RELATED WORK

Having defined the architecture context of our work, in this section we discuss some related work on which various components of our architecture rely, namely dynamic pruning (Section 3.1), query efficiency prediction (Section 3.2) and selective pruning (Section 3.3).

3.1 Dynamic Pruning

The strategies to match documents to a query fall in two categories [16]: in a *term-at-a-time (TAAT)* strategy, the posting lists of query terms are processed and scored in sequence, while, in a *document-at-a-time (DAAT)* strategy, the query term postings lists are processed in parallel. To attain the typical sub-second response times of Web search engines, various techniques to enhance retrieval efficiency have been proposed (e.g. [4, 16, 20]). In particular, *dynamic pruning* aims to eliminate the scoring of documents that will not be present in the final list of top results. Most DAAT dynamic pruning strategies [4, 20] exhibit efficiency improvements without negatively impacting effectiveness, but some TAAT dynamic pruning techniques [12, 16], while they enhance efficiency, negatively impact retrieval effectiveness because some relevant document can be pruned.

In this work, we consider the Continue TAAT dynamic pruning strategy [16], which we denote TAAT-CS. Our choice of the TAAT-CS strategy is motivated by the fact that its overall efficiency is directly proportional to the number of accumulator to create in the first phase [16]. Indeed, the fine tuning of the number of accumulators gives us the flexibility to directly control the efficiency of the pruning strategy.

3.2 Query Efficiency Prediction

The query scheduler component 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 estimate of the processing time for the query to be scheduled. Indeed, efficiency predictions estimate the response time of a search engine for a query [14].

Moffat et al. [15] 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 [4]), the response time of a query is more variable, as not every posting is scored, and many postings can be skipped [16], 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 [14]. *Query efficiency predictors* [14] 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 work, 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.

3.3 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 [4]. 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 [16].

Typically, the aggressiveness is selected a priori to any retrieval, independent of the query to be processed and its characteristics. However, in [19], Tonellotto *et al.* showed how the Wand pruning strategy could 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 work makes an important improvement to selective pruning compared to [19], by 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.

4. 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.

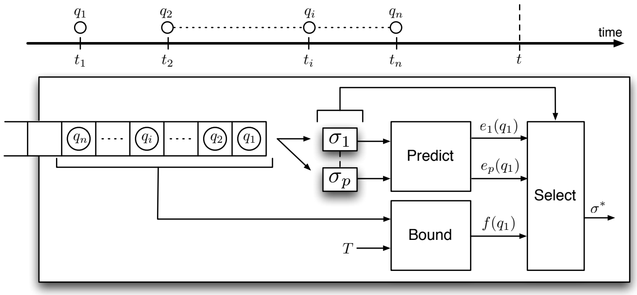

Figure 3: The components of the proposed load-sensitive selective pruning framework (bottom), along with a representation of the variables depicting the queries currently queued (top).

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 1.

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 3, which shows the system at time t. There are 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 (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_{k+1} is more efficient than σ_k, the effectiveness of σ_k is, in general, better than the effectiveness of σ_{k+1}. This assumption is well-founded, because efficient processing strategies typically have a negative impact on the corresponding retrieval effectiveness [13, 19, 21].

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 has 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 available time, 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 paper. 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 aspects: 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 1. 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 1 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 $q_1$
for all processing strategies $\sigma_k$, $k = 1, \ldots, p$
    for all enqueued queues $q_i$, $i = 1, \ldots, n$
        expected processing time $e_k(q_i) \leftarrow \mathsf{Predict}(\sigma_k, q_i)$
Time budget $f(q_1) \leftarrow \mathsf{Bound}(T, \sigma_1(q_1), \ldots, \sigma_p(q_n))$
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 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 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 [19], it follows that the processing times of a query can be estimated through the use of query efficiency prediction [14], i.e. Predict(). However, as no such predictors have previously been defined for TAAT strategies such as TAAT-CS, in Section 5 we address query efficiency prediction for TAAT. In the remainder of this section, we propose mechanisms for Bound() (Section 4.1) and Select() (Section 4.2).

4.1 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 four 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:

$$f(q_1) = \operatorname*{argmax}_k \{e_k(q_1)\} = e_1(q_1).$$

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 with maximal effectiveness.

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 explicitly 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 do not exploit the unused resources.

Selfish. The query q_1, enqueued at time t_1, should be processed by time $t_1 + T$. Hence, at time t, the amount of remaining time Δ_1 to process the query such that threshold T is met has decreased by $t - t_1$ seconds, i.e.:

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

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. Then, if the time threshold T for this query has elapsed ($\Delta_1 \leq 0$), 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 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 up to 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 our paper. 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.

4.2 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 where $1 \leq k \leq p$ is the smallest such that $e_k(q_1) \leq f(q_1)$. In order words, we select σ_k as the best strategy in terms of effectiveness, whose expected completion time is not greater than the budget the query has been granted by Bound().

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

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 the 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 [19] – we predicting the efficiency of a TAAT-CS strategy before processing commences.

5. PRUNING 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 4 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 document-at-a-time (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 a high workload, we resort to use faster and less precise processing strategies, specifically, based on the term-at-a-time-continue strategy (TAAT-CS) [16]. In the remainder of this section, we define the details of TAAT-CS (Section 5.1), before explaining how the processing time of both DAAT and TAAT-CS can be accurately measured (Section 5.2).

5.1 TAAT-CS Dynamic Pruning

As defined in [16], 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 [16] 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 [9], we alter the accumulator creation phase as follows. We select the shortest l posting 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 paper. Yet, on average, the response time of our DAAT/TAAT-CS strategies exhibit a 2x improvement over the classical TAAT-CS strategy.

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

The adoption of the DAAT/TAAT-CS strategy motivates also the comparison of our selective pruning strategies with DAAT, instead than TAAT. Indeed, in terms of efficiency, out-performing DAAT as a baseline is, in general, more difficult than for TAAT [9]. In the following, we refer to our DAAT/ TAAT-CS with K accumulators as CS-K (e.g. CS-1000 uses $K = 1000$ accumulators), without further mention of 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 Query Efficiency Prediction

In the preceding section, we defined the processing strategies used within this paper. 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.* [14]. 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 define a set of features to represent each query. In the case of DAAT, Macdonald *et al.* [14] show that there is a strong correlation between the 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 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 using DAAT, and 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 1, in addition to the method-independent features 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,

$$e_j(q_i) = \lambda_{j0}\mathsf{f}_j0 + \ldots + \lambda_{j9}\mathsf{f}_j9.$$

Linear regression is then used to find the values for various λ_{jf} with the goal of minimising the least square error of processing time on a training set of queries [14].

In the next section, we define the experimental setup for our experiments. In particular, our experiments demonstrate the accuracy of the proposed efficiency predictors for TAAT-CS, before showing how the proposed selective scheduling framework proposed in Section 4 can increase the ability of a search engine to effectively and efficiently handle different traffic query loads.

6. 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 [17], 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, which is stored on disk, also has skipping information embedded, to permit skipping [16] 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. Only a single core on each query server is used to serve queries.[2] 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 [18].

We use queries from the TREC Million Query Track 2009 [6], 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 computed over 5 runs for each test. In our experiments, we do not use query caching, in order to better analyse the impact of our models on the processing performance. Moreover, adding a cache in front of our architecture would only reduce the query arrival rate, but not the efficiency and effectiveness of our method.

7. 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 7.1)
RQ2. Do the proposed methods achieve effective and efficient retrieval under different query loads? (Section 7.2)
RQ3. To what extent can efficient query per second servicing be attained for different time thresholds? (Section 7.3)

7.1 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 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 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 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)[3], the error is one order of magnitude lower, and for the vast majority of queries (95-99%) our linear model is able to predict 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.

[2] While increasing the number of cores on each query server obviously increases throughput, we prefer to use a single-threaded environment to reduce any resource contention that may reduce the reliability of experimental results.

[3] 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 ≤ 10 ms	RMSE	err ≤ 10 ms	RMSE	err ≤ 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 2: Mean query processing time (QPT, in seconds), as well as prediction accuracy using various feature sets, for each processing strategy.

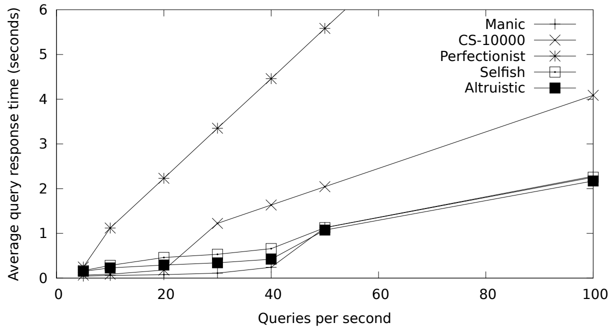

Figure 4: Average query response time in seconds for different methods, $T = 0.5$.

In particular, the best performing models for predicting CS-K strategies are those obtained by the full set of ten features described in Section 5, 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 the DAAT processing times and the full set of ten features for the prediction of TAAT CS-K processing times.

7.2 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 7.3, 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.

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. As the Million query track query set does not have query arrival times, queries are submitted at uniform query rate – in other words a submission rate of N q/s corresponds to submitting a query every $1/N$ seconds. This allows us to measure the behaviour of the various techniques under various load conditions, as shown in Figure 4. 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.

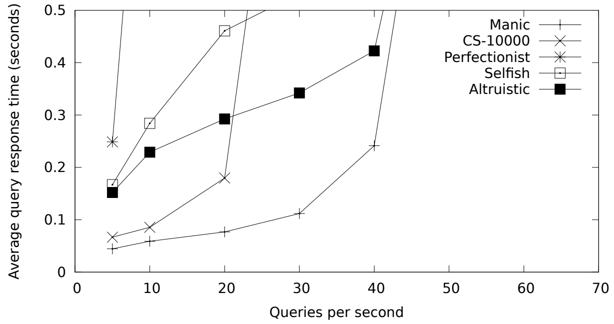

Figure 5: Average query response time in seconds for different methods (enlargement of Figure 4).

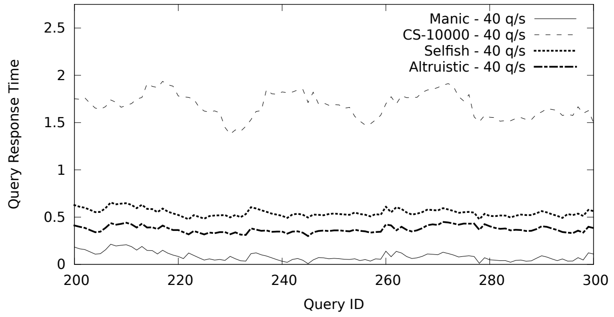

Figure 6: Query response time for 100 queries, arrival rate 40 q/s, $T = 0.5$.

Figure 5 enlarges the curves of Figure 4 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 6 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 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.

To determine how the threshold is adhered to for different methods and workloads, Figure 7 shows the percentage of queries whose response time are within the threshold

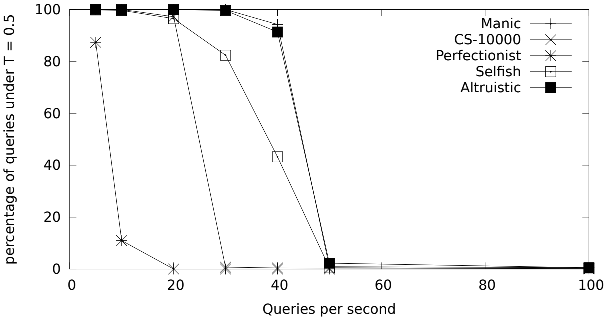

Figure 7: Percentage of queries achieving $T = 0.5$ for different methods and query workloads.

$T = 0.5$. 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 8, we plot the NDCG@1000 values achieved for the different methods and workloads. These results are mirrored in Table 3, 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 8, 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.

Finally, to complete our answering of research question **RQ2**, effectiveness and efficiency results must be compared, by examining Figures 4 and 5 (query processing time) and Table 3 (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. 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.

7.3 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 the requirements of the search engine. In Figure 9 & 10, 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 $T = 0.25$).

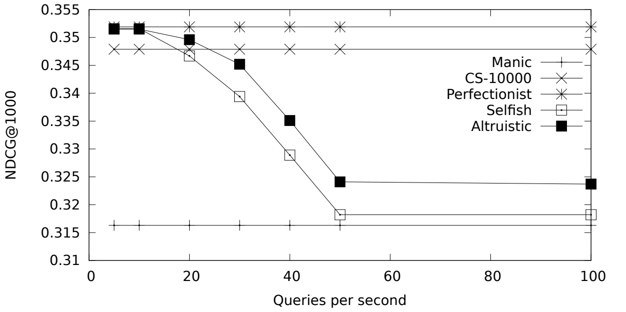

Figure 8: NDCG@1000 computed over the 687 queries in the test set, $T = 0.5$.

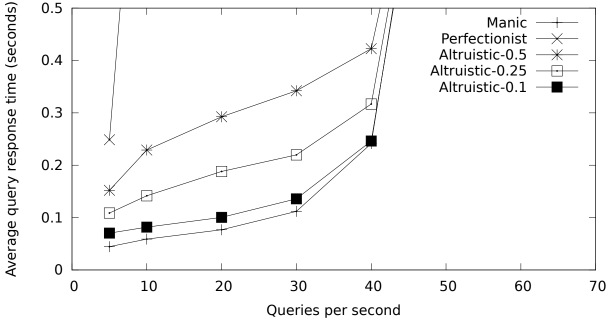

Figure 9: Average query response time in seconds for different T.

As expected, on examination of Figure 9, we find that by lowering the time threshold we also reduce the maximum sustainable load. In general, Altruistic attains 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 up to 40 q/s, while $T = 0.25$ is achieved up to 30 q/s, and up to 10 q/s for $T = 0.1$ seconds. On the other hand, on examination of Figure 10, 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 this is only a 5% difference[4] 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.[5]

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 over the effectiveness of the uniform Manic method.

8. CONCLUSIONS

In this paper, we presented an innovative solution to the important problem of processing queries during times of high

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

[5] While we conducted experiments on the effectiveness of both dropping and interrupting query processing, we do not fully report these results due to lack of space. In brief, as expected, the drop strategy for Perfectionist results in a dramatic decrease in NDCG@1000, down to 0.097 for 40 q/s. A slightly better result is obtained when the queries that exceed the deadline are interrupted, and partial results returned. In this case, NDCG@1000 was 0.23, which is still markedly lower than Manic for the same setting.

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 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 ▲ ($p < 0.01$).

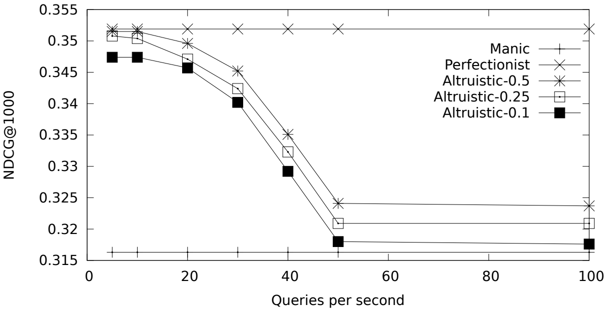

Figure 10: NDCG@1000 for different T.

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 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 7) while still attaining significantly high effectiveness (Table 3). In contrast, the next most effective Selfish method can only meet the deadline for 40% of queries.

This paper opens several directions for future work. For instance, one direction is the definition of a computational optimisation problem to address Bound() that can adapt to the continuous stream of queries arriving. On the other hand, we believe that improved definitions for Predict() and Bound() can consider the entire distributed search architecture, rather than each query server in independence.

Acknowledgements.

This work was partially supported by the EU projects InGeoCLOUDS (no. 297300), MIDAS (no. 318786), E-CLOUD (no. 325091), the Italian PRIN 2011 project "Algoritmica delle Reti Sociali Tecno-Mediate" (2013-2014) and the Regional (Tuscany) project SECURE! (FESR PorCreo 2007-2011).

9. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proceedings of SIGIR 2001.*

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 2nd ed., 2008.

[3] L. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *Micro*, 23(2), 2003.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of CIKM 2003.*

[5] B. B. Cambazoglu and R. Baeza-Yates. Scalability challenges in web search engines. In *Advanced Topics in Information Retrieval*, volume 33 of *The Information Retrieval Series.* Springer, 2011.

[6] B. Carterette, V. Pavlu, H. Fang, and E. Kanoulas. Million Query Track 2009 Overview. In *Proceedings of TREC 2009.*

[7] N. Craswell, D. Fetterly, and M. Najork. The power of peers. In *Proceedings of ECIR 2011.*

[8] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proc. of WSDM 2009.*

[9] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *PVLDB*, 4(12), 2011.

[10] A. Freire, C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda. Scheduling queries across replicas. In *Proceedings of SIGIR 2012.*

[11] S. Jonassen, B. B. Cambazoglu, and F. Silvestri. Prefetching query results and its impact on search engines. In *Proceedings of SIGIR 2012.*

[12] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Proceedings of WISE 2005.*

[13] C. Macdonald, N. Tonellotto, and I. Ounis. Effect of dynamic pruning safety on learning to rank effectiveness. In *Proceedings of SIGIR 2012.*

[14] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *Proceedings of SIGIR 2012.*

[15] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(5), 2007.

[16] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM TOIS*, 14(4), 1996.

[17] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A high performance and scalable information retrieval platform. In *Proceedings of the OSIR Workshop at SIGIR 2006.*

[18] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of SIGIR 1994.*

[19] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient and effective retrieval using selective pruning. In *Proceedings of WSDM 2013.*

[20] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *IPM*, 31(6), 1995.

[21] L. Wang, J. Lin, and D. Metzler. Learning to efficiently rank. In *Proceeding of SIGIR 2010.*