# Predictive Parallelization: Taming Tail Latencies in Web Search

Myeongjae Jeon[1], Saehoon Kim[2], Seung-won Hwang[2], Yuxiong He[3], Sameh Elnikety[3], Alan L. Cox[1], Scott Rixner[1]

[1]Rice University [2]POSTECH [3]Microsoft Research

## ABSTRACT

Web search engines are optimized to reduce the high-percentile response time to consistently provide fast responses to almost all user queries. This is a challenging task because the query workload exhibits large variability, consisting of many short-running queries and a few long-running queries that significantly impact the high-percentile response time. With modern multicore servers, parallelizing the processing of an individual query is a promising solution to reduce query execution time, but it gives limited benefits compared to sequential execution since most queries see little or no speedup when parallelized. The root of this problem is that short-running queries, which dominate the workload, do not benefit from parallelization. They incur a large parallelization overhead, taking scarce resources from long-running queries. On the other hand, parallelization substantially reduces the execution time of long-running queries with low overhead and high parallelization efficiency. Motivated by these observations, we propose a predictive parallelization framework with two parts: (1) predicting long-running queries, and (2) selectively parallelizing them.

For the first part, prediction should be accurate and efficient. For accuracy, we study a comprehensive feature set covering both term features (reflecting dynamic pruning efficiency) and query features (reflecting query complexity). For efficiency, to keep overhead low, we avoid expensive features that have excessive requirements such as large memory footprints. For the second part, we use the predicted query execution time to parallelize long-running queries and process short-running queries sequentially. We implement and evaluate the predictive parallelization framework in Microsoft Bing search. Our measurements show that under moderate to heavy load, the predictive strategy reduces the 99th-percentile response time by 50% (from 200 ms to 100 ms) compared with prior approaches that parallelize all queries.

## Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; D.4.1 [**Operating Systems**]: Process Management—*Threads*


*SIGIR'14,* July 6–11, 2014, Gold Coast, Queensland, Australia.

http://dx.doi.org/10.1145/2600428.2609572 .

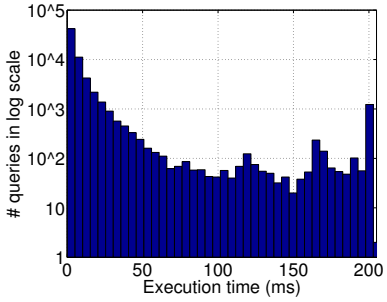



**Figure 1: Histogram of the sequential query execution time of 70K queries at Bing search. The x-axis is in 5 ms bins, and the y-axis is in log scale. Measurement setups are provided in Section 5.**

## Keywords

Parallelism; Query execution time prediction; Tail latency

## 1. INTRODUCTION

Achieving consistently low response times is a primary design objective for web search engines. Long query response times directly degrade user satisfaction and reduce revenues [22]. To provide timely responses to user queries, a web search engine operates under a service level agreement (SLA), *e.g.*, 100 ms for 99th-percentile response time. In particular, search engines are optimized to reduce the high-percentile response time, which is more important than the mean response time [9].

Reducing the high-percentile response time (also called the *tail latency*) is challenging because the search engine workload exhibits high variability as shown in Figure 1. Most queries are short-running, with more than 85% taking less than 15 ms. However, few queries are very long-running taking up to 200 ms. In particular, the average of the execution time is 13.47 ms while 99th-percentile execution time is 200 ms, which is 15 times the average. The gap between the median and the 99th-percentile is even larger at 56 times. Therefore, to reduce high-percentile response time, it is important to speed up the long-running queries.

Parallelizing the processing of each query is a promising solution to reduce query execution time [24, 14]. This is motivated by current hardware trends. A modern server has several cores, and with parallelization, multiple threads execute a query concurrently using the available cores to reduce execution time. When servers are lightly loaded and there are sufficient number of available cores, parallelizing the execution of all queries reduces their execution time, thereby reducing the response time.

However, parallelizing all queries is ineffective under moderate and high load because it comes with an overhead that varies among queries. Our measurements show that long-running queries achieve better speedup with lower overhead and higher parallelization efficiency. In contrast, parallelizing short-running queries is ineffective, giving no performance benefit while consuming additional resources. As the load increases, spare processor resources become limited and we can no longer afford to parallelize all queries. We thus propose a method to selectively parallelize long-running queries for reducing the high-percentile response time, and to execute short-running queries sequentially avoiding their parallelization overhead.

In this paper, we argue that (1) accurate and efficient prediction of query execution times is essential for (2) implementing an effective selective parallelization scheme.

For *prediction*, we first identify the requirements of the predictor in terms of accuracy and efficiency in order to support selective parallelization to reduce tail latency. We find that the state-of-the-art predictor for query response time [16] is not accurate enough and uses expensive features that consume large memory space. We improve the predictor from three aspects. (1) While the prior work focuses mainly on term features, we exploit a comprehensive list of term and query features to achieve higher accuracy. (2) Our predictor incorporates query rewriting, which is common in modern search engine, to improve accuracy further. (3) To reduce the prediction cost, we introduce a memory-efficient feature set, which, compared with using all features, achieves comparable prediction accuracy while saving more than 90% of the memory space needed for caching the features. These techniques improve precision (from 0.62 to 0.83) and recall (from 0.49 to 0.78) of prediction compared with the prior work, while reducing prediction overhead in terms of both memory and CPU usage.

For *parallelization*, we use the predicted query execution time to selectively parallelize the long-running queries and to execute the short-running queries sequentially. Our implementation results on production servers of Bing search show that under moderate to heavy load, predictive parallelization reduces the 99th-percentile response time by 50% (from 200 ms to 100 ms) compared with prior approaches that parallelize all queries. It also significantly outperforms a recent adaptive parallelism policy [14].

The contribution of the paper is the design and evaluation of the predictive parallelism framework:

- Improved prediction: We systematically explore the design space of prediction features and learning algorithms to construct a predictor that incorporates query rewriting and uses a memory-efficient feature set that achieves high accuracy. Our proposed framework is the first satisfying dual requirements of accuracy and efficiency to enable predictive parallelization. The predictor is trained and evaluated using real user query logs and production web index, reflecting many important aspects that have been neglected in the previous studies.

- Selective parallelization: We use the predicted query execution time to parallelize only long-running queries. We evaluate this predictive parallelization framework in a production environment of a major commercial search engine. The results show that the proposed solution significantly reduces the 99th-percentile latency compared with sequential execution and two state-of-the-art parallelization policies. It is currently deployed and used to serve millions of user queries daily.

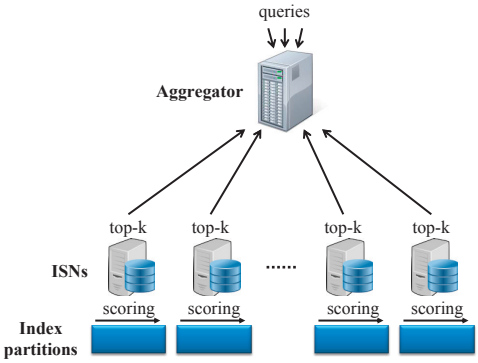



**Figure 2: Index serving system architecture.**

The remainder of this paper is organized as follows. Section 2 provides the background on the search engine environment. Section 3 discusses query parallelization. Section 4 develops the prediction framework. We evaluate the proposed predictive parallelization framework experimentally in Section 5. We contrast our contributions to related work in Section 6 and draw conclusions in Section 7.

## 2. SEARCH ENGINE ENVIRONMENT

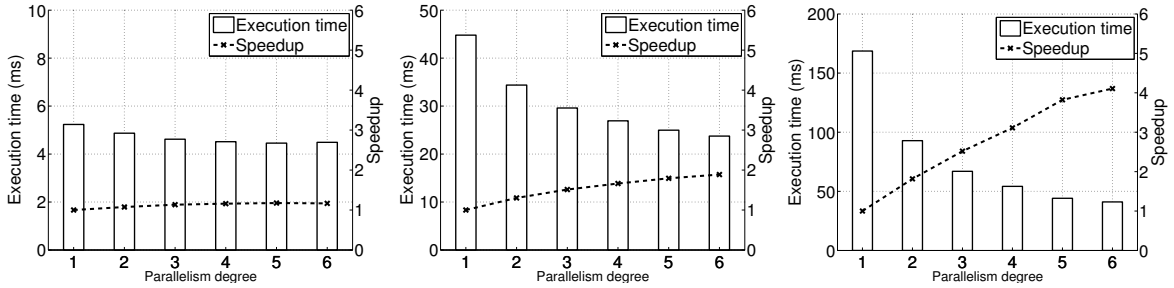
### 2.1 System Architecture

Figure 2 illustrates a common architecture of an index serving system consisting of index serving nodes (ISNs) and a single or multiple levels of aggregators (also known as brokers) [8, 14]. An entire web index, containing information on billions of Web documents, is document-sharded [3] and distributed among a large number of ISNs. When a user sends a query and its results are not cached, the aggregator propagates the query to all ISNs hosting the web index to process and answer it. Each ISN searches its fragment of the web index to retrieve the top-$k$ (currently $k = 4$) matching Web documents to the aggregator.[1] The aggregator receives the results from the ISNs, and merges them to compute the response for the user query. ISNs are the workhorse of the index serving system. They constitute over 90% of the total hardware resources. Moreover, they are on the critical path for query processing and account for more than two-thirds of the total query processing time. We use Microsoft Bing, a commercial search engine, to evaluate our techniques.

### 2.2 ISN Query Processing

The ISN is a multi-threaded server, capable of processing several queries concurrently for higher efficiency. Newly arrived queries first join the waiting queue of the ISN. When a thread is idle, it gets a new query from the head of the waiting queue and starts to process it. As there are a fixed number of threads in the ISN, some queries may be delayed in the waiting queue (*i.e.,* queueing time), in addition to their processing time (*i.e.,* execution time) — query response time is the sum of its queueing and execution time.

[1]Though $k$ can be varied to trade search quality (effectiveness) for query execution time (efficiency) [25], we fix $k$ and focus on optimizing efficiency without a quality compromise.




(a) Short-running queries (0–30 ms) (b) medium-running queries (30–80 ms) (c) long-running queries (>80 ms)

**Figure 3: Average execution time and speedup in parallelization for queries classified by their execution time.**

A thread in the ISN searches its web index shard to return the matching documents of a query. The documents inside the index shard are sorted based on static scores (such as PageRank [4]) reflecting the popularity and importance of the document. When an ISN determines that a Web document matches the user query, it computes the relevance score using a ranking function that combines many factors including the document static score, term features (such as term frequency in the document [18]) and other features (such as proximity to user location or history). The query processing operates in conjunctive mode [14] and follows "document-at-a-time (DAAT)" [5]. That is, for a multi-term query, all posting lists are intersected and the score of a document is fully computed before moving to the next document. The ISN employs dynamic pruning to early terminate a query and avoid the scoring of postings for documents that cannot make the top-$k$ retrieved set [26].

# 3. QUERY PARALLELIZATION

## 3.1 Parallelization Design

A basic approach to parallelizing a single keyword query is to partition the entire posting list into the same number of chunks (or groups) of documents as the number of threads allocated to the query, and assign each chunk to a thread. This approach incurs a very large overhead as it processes the entire posting list — documents with low static score are rarely visited during sequential execution as a result of dynamic pruning.

To support dynamic pruning during parallel processing, we use an approach that partitions the posting list into many small chunks and assigns them to threads in the order confirming the static score. We follow the thread pool model, where the chunks constitute a list of work items ordered by static scores of documents. An idle thread requests the first unprocessed chunk from the head of the list to process documents with higher static scores first. This results in an execution that is similar to sequential processing, allowing dynamic pruning while minimizing unnecessary processing of the posting list. Prior work [14] studies a similar approach, and therefore we do not discuss its details further.

## 3.2 Speedup Characteristics

We characterize the ISN query processing workload to show how and when parallelization can reduce the query response time. The results in this section are based on the same experimental setup as discussed in Section 5.

Queries have varying execution times as we discussed for the results in Figure 1: the majority (over 85%) is short-running requiring less than 15 ms, and a few queries are long-running. Moreover, the 99th-percentile execution time is 56 times the median, showing a large variability.

Parallelization is effective only for long-running queries. We classify the queries into three classes based on their execution times and show the speedup, which is the sequential execution time divided by the parallel execution time, with different parallelism degrees in Figure 3. The queries that run longer than 80 ms achieve more than 4 times speedup by using 6 threads. This reduces their mean execution time from 167 ms to 41 ms. In contrast, using 6 threads, the queries that complete within 30 ms achieve just about 1.15 speedups because their execution time is dominated by the sequential part of query processing (*i.e.,* non-parallelized part such as parsing and rescoring of the top results). In addition, for medium-running queries, which run between 30 and 80 ms, the speedup is modest, less than 2 for parallelism degree 6.

We derive two important conclusions from Figure 3. First, it is important to parallelize long-running queries to reduce response time. Without parallelization they are more likely to have a response time over 100 ms, violating the SLA. In addition, they benefit the most from parallelization, showing high speedups. Second, we should execute short-running queries sequentially. Parallelizing short-running queries is a bad strategy because they do not benefit from parallelization. More importantly, they compete for the computational resources, making those limited resources unavailable for parallelizing long-running queries.

In summary, although parallelization holds the promise of reducing query execution times significantly, it has to be applied carefully on a per-query basis. There are enormous benefits for "selectively" parallelizing long-running queries. Parallelizing every query may result in much higher resource utilization than sequential execution and fail to reduce the response time. This work employs the pre-retrieval prediction of query execution time (*i.e.,* before executing the query), and applies it to long-running query parallelization — we call this approach *predictive parallelization*.

# 4. PREDICTION FRAMEWORK

This section explores the design of the learning framework to support predictive parallelization. Specifically, we discuss the following questions:

- What are the requirements of prediction?

- What is the space of features for prediction and which features should we use?
- What is the space of algorithms for prediction and which algorithm should we use?

## 4.1 Requirements

Predictive parallelization imposes four requirements on the prediction framework: tail latency, misprediction cost, prediction overhead, and flexibility. We use the standard metrics of prediction accuracy, namely precision and recall:

$$\text{precision} = \frac{|A \cap P|}{|P|}, \quad \text{recall} = \frac{|A \cap P|}{|A|},$$

where $A$ is a set of true long-running queries, and $P$ is a set of predicted long-running queries.

***1-Tail latency.*** To reduce the tail latency, we must correctly identify a majority of long-running queries and reduce their execution time through parallelization. For example, for our workload, to meet the 99th-percentile response time target, the prediction should achieve a recall of 0.75 or higher (*i.e.,* correctly identifying 75% or more of true long-running queries). To illustrate, the Bing workload shows that the queries running longer than 80 ms are about 4% of the total number of queries. As long as at least 75% of the true long-running queries can be identified, the remaining true long-running queries, which are wrongly predicted as short-running, contribute to less than $4\% \times (1 - 0.75) = 1\%$ of the total queries. This small portion, 1% or less, does not affect the 99th-percentile response time. Therefore, the predictor should have a recall of 0.75 or higher.

***2-Misprediction cost.*** The misprediction cost comes from the prediction error in which short-running queries are predicted as long-running. This overhead is directly related to the precision achieved. When such misprediction happens, the processor resources are wasted to parallelize short-running queries with almost no benefit. Prediction with higher precision incurs lower misprediction cost, which matters little at light load but has a significant impact at heavy load. We elaborate the importance and show which precision values are required in Section 5.

***3-Prediction overhead.*** The overhead involved in performing prediction must be small to keep the interactive nature of web search. Prediction itself adds additional work to query execution, increasing query response time. Therefore, prediction should return the predicted execution time for a query quickly. Since the average query execution time is about 15 ms, adding 5% of it for prediction is an acceptable cost for the potential benefits. Here, we set the goal of less than 0.75 ms to predict query execution time.

***4-Flexibility.*** The ability to adjust the threshold of defining long-running queries allows the predictor to adapt to varying load and to achieve better performance (Section 5). We thus abstract prediction as a regression problem (of estimating the execution time) rather than a classification problem (of deciding whether the query is long-running or not). We empirically show that this flexibility comes without loss in prediction accuracy (Section 4.3).

## 4.2 Features

In this section, we describe the features that can be used for prediction and analyze the importance of the features.

| Category | Feature | Description |
|---|---|---|
| term feature | AMeanScore | Arithmetic mean scores |
| | GMeanScore | Geometric mean scores |
| | HMeanScore | Harmonic mean scores |
| | MaxScore | Maximum of scores |
| | EMaxScore | Estimated maximum scores [15] |
| | VarScore | Variance of scores |
| | NumPostings | # postings |
| | NumMaxima | # local maxima |
| | GAvgMaxima | # maxima greater than average |
| | MaxNumPostings | # postings with maximum score |
| | In5%Max | # postings in 5% of maximum score |
| | IDF | inverse document frequency |
| | NumThres | # postings in 5% of the $k$th score |
| | ProK | # docs ever promoted to the top-$k$ |
| query feature | English | Query in English or not (binary) |
| | NumAugTerm | # augmented requirements |
| | Complexity | Degree of query complexity |
| | RelaxCount | Relax count applied or not (binary) |
| | NumBefore | # terms in the original query |
| | NumAfter | # terms after query rewriting |

**Table 1: Space of the features.**

### 4.2.1 Space of Features

We investigate features that meaningfully correlate with the execution time, which we categorize into *term* and *query* features. Table 1 lists 14 term features and 6 query features.

***Term features.*** Term features capture the "efficiency" of queries by estimating the effects of dynamic pruning. Table 1 presents 14 term features, studied to be good predictors [16].

Most features in the table are self-explanatory, and we explain three features in more details:

- NumMaxima: the number of times that the gradient of the score curve across all postings is 0, *i.e.,* the number of times there is a local maxima in the postings curve.
- GAvgMaxima: the number of documents with the maxima score greater than average. The low score also indicates higher pruning effect.
- IDF: the inverse document frequency of the term. This accounts for the number of documents in the corpus that contain the term [23].

Since a query may contain multiple terms, scores of a term feature across the query terms need to be combined into a single score using an *aggregation* function. For example, for the query "Gold Coast", scores of a term feature for "Gold" and "Coast" are aggregated by each of four functions: maximum, minimum, variance, and summation. In other words, possible feature scores that can be computed from the aggregation is $14 \times 4 = 56$, depending on which of the 14 term features or which of the 4 aggregation functions used. Prior work proposed maximum, variance, and summation as the aggregation functions [16]. In this work, we add the minimum as an aggregation function because the minimum matches the way a conjunctive query is processed.[2]

In general, we find that retrieving term features from a term index is expensive. Although these features can be precomputed and cached in the memory, this requires a large memory footprint. As an example, consider a case

[2]This observation may not be consistent with existing prediction framework [16] that builds on disjunctive WAND queries.

where roughly one hundred million terms are stored within a server. Caching the features requires 4.47 GB memory, which is unacceptable for an index server which benefits more from using this memory to store a larger subset of the inverted web index partition. Moreover, the precomputed term feature information needs to be updated frequently whenever new postings are added or existing postings are deleted, incurring additional overhead. Given the high caching and maintenance cost of term features, we introduce a new type of features, called *query features*, which improve prediction accuracy with much lower cost.

***Query features.*** Query features capture the "complexity" of a query, and this complexity affects the execution time. For example, the number of terms in a query often positively correlates with the execution time. Also, the language of the query, which is related to the size of the corpus to be searched, strongly correlates to the execution time.

We propose to exploit both term and query features, rather than only term features used in the prior work [16], for the following advantages. First, in a modern search engine, a query is frequently rewritten to correct the errors or ambiguity in the user input keywords [6]. For example, in our query logs, we observe that nearly half the queries are rewritten. This rewriting significantly increases the number of terms and the query complexity, which is a dominant factor of overall execution time. Second, query features are conveniently available at runtime with a low cost.

To elaborate how the query features reflect the query rewriting process, consider a rewritten query due to spelling correction. A mistyped query like "facebok" is automatically rewritten into another query, such as "facebok OR facebook" to retrieve more useful results for the user. Here, using term features alone for the given term "facebok" inevitably leads to inaccurate results. In contrast, query features, reflecting the number of terms after such rewriting, more closely estimate the execution time for such queries. Query features reflecting whether the given query requires such additional processing enable more accurate prediction.

Table 1 lists six query features, and we describe three features in more details here:

- NumAugTerm: the number of augmented requirements for the given query. These are used in general for the personalization [19] of search, and the number positively correlates with the execution time.
- Complexity: how complex it is to find a match with query terms. To find matches a query is translated into an execution plan which walks through posting lists of the terms, and its complexity is subject to the number of rewritten terms and the length of phrases. The complexity is a numeric value.
- RelaxCount: whether queries are relaxed to generate more meaningful subqueries. For example, query "Microsoft Office Windows" can be relaxed to "Microsoft Windows" or "Microsoft Office" [21]. Having it enabled correlates with higher execution time.

***Cheap features.*** We develop a memory-efficient feature set (called "cheap features"), which contains all query features and IDF from term features combined using four aggregation functions (*i.e.*, minimum, maximum, summation, and variance). We later show that (1) our predictor using only the cheap features meets the accuracy requirements of predictive parallelism (Section 4.3), (2) the cheap features are practical since they do not require a large memory footprint (Section 4.3), and (3) the cheap features work well for parallelization (Section 5).

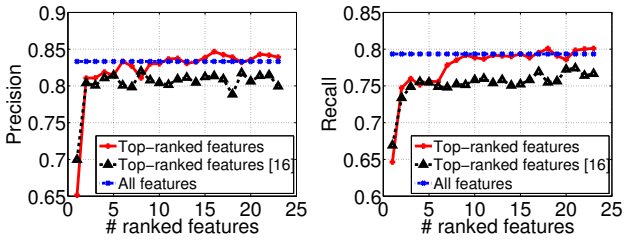
| Feature | Importance |
|---|---|
| **Max-IDF** | 1 |
| Sum-AMeanScore | 0.34823 |
| Min-MaxScore | 0.33350 |
| Min-MaxNumPostings | 0.28197 |
| Sum-HMeanScore | 0.27014 |
| **English** | 0.26460 |
| **RelaxCount** | 0.21890 |
| **Min-IDF** | 0.19128 |
| Max-GMeanScore | 0.18497 |
| **NumAugTerm** | 0.18442 |
| Sum-VarScore | 0.17083 |
| Var-HMeanScore | 0.16660 |
| Var-MaxNumPostings | 0.16227 |
| Var-MaxScore | 0.15709 |
| **Complexity** | 0.13965 |

**Table 2: Top-15 features ranked by the importance obtained from boosted regression tree. Cheap features are shown in bold.**

### 4.2.2 *Feature Analysis*

This section studies which feature is a good predictor of query execution time. As a metric, we first use per-feature gain from boosted regression tree, where the importance of a feature is proportional to the total error reduction per split in the tree. Table 2 shows the top-15 most important features, with each importance normalized to the highest value. We observe that four out of the top-15 features are query features that do not require accesses to term index or need large memory footprint. More importantly, most of the cheap features (shown in **bold**) are ranked high, which indicate that they are good predictors while incurring low prediction cost.

To further support this argument, we assume that all features were available at runtime and we study the overall prediction accuracy changes when a subset of features is selected by the order presented in Table 2. Figure 4 presents both precision and recall with a threshold of 50 ms for identifying long-running queries. The figure shows, when the top-10 most important features in Table 2 are used (the red curve), both precision and recall converge to the case of using all term and query features (the blue curve). In addition,




(a) Precision at 50 ms (b) Recall at 50 ms

**Figure 4: Precision and recall with features added in the order of importance for boosted regression tree. This shows the performance difference between our order (the red curve) and the order suggested in the prior work [16] (the black curve).**

| Threshold | Metric | Algorithm | term features + query features | cheap features | Prior work [16] |
|---|---|---|---|---|---|
| 50 ms | Precision | Linear regression | 0.6062 ±0.0027 | 0.6216 ±0.0045 | 0.6200 ±0.015 |
| | | Gaussian process regression | 0.6897 ±0.0051 | 0.6852 ±0.0098 | 0.6089 ±0.011 |
| | | Boosted regression tree | **0.8306** ±0.013 | **0.7927** ±0.013 | 0.6942 ±0.015 |
| | Recall | Linear regression | 0.6838 ±0.002 | 0.2997 ±0.0022 | 0.4912 ±0.0050 |
| | | Gaussian process regression | 0.7884 ±0.0071 | 0.6655 ±0.0282 | 0.5906 ±0.0203 |
| | | Boosted regression tree | **0.8007** ±0.0057 | **0.7723** ±0.0078 | 0.6280 ±0.0098 |
| 80 ms | Precision | Linear regression | 0.6717 ±0.0029 | 0.7074 ±0.0014 | 0.6716 ±0.0047 |
| | | Gaussian process regression | 0.8123 ±0.0050 | 0.7712 ±0.0503 | 0.7028 ±0.0102 |
| | | Boosted regression tree | **0.8894** ±0.0100 | **0.8567** ±0.0102 | 0.7643 ±0.0143 |
| | Recall | Linear regression | 0.6627 ±0.0010 | 0.1929 ±0.0290 | 0.2817 ±0.0012 |
| | | Gaussian process regression | 0.7568 ±0.0114 | 0.4940 ±0.1344 | 0.5329 ±0.0066 |
| | | Boosted regression tree | **0.8370** ±0.0084 | **0.7961** ±0.0063 | 0.6354 ±0.0153 |

**Table 3: Prediction accuracy of linear regression, Gaussian process regression, and boosted regression tree for two threshold values 50 ms and 80 ms. The accuracy is presented with 95% confidence interval.**

| Algorithm | Training time | Prediction overhead |
|---|---|---|
| Boosted regression tree | 1.805 (sec) | < 0.75 (ms) |
| Linear regression | 0.006 (sec) | < 0.75 (ms) |
| Gaussian process | 539.326 (sec) | < 0.75 (ms) |

**Table 4: Training time and prediction overhead comparisons for the three algorithms.**

| Approach | Query\term feature | Query rewriting | Memory usage |
|---|---|---|---|
| Prior work [16] | - \All | - | 4.47 GB |
| All features | Y \All | Y | 4.47 GB |
| Cheap features | Y \IDF | Y | 0.37 GB |

**Table 5: Design comparison of existing approach and our solutions. Memory usage is calculated based on one hundred million terms stored in a server.**

to put these results in context, we compare to related work. We select a set of features according to the order proposed in the prior work [16], and observe that convergence does not happen even when all features are used (the black curve).

An analysis with the ranked features suggests that the per-feature gain used in this work is a reliable metric for feature selection, making it possible to build the prediction with accuracy comparable to using all features, only with 10 features or less. Since our cheap features contain five out of the top-10 features, we expect that the cheap features should work well.

## 4.3 Empirical Evaluation

***Evaluation of regression algorithms.*** We evaluate the accuracy of three regression algorithms: linear regression, boosted regression tree [12], and Gaussian process regression [20]. The boosted regression tree and Gaussian process are nonlinear regression algorithms. We collect 22,000 user queries from search engine, and perform 5-fold cross validation with 5 repetitions to avoid biased results. Table 3 presents the average of precision and recall with 95% confidence intervals.

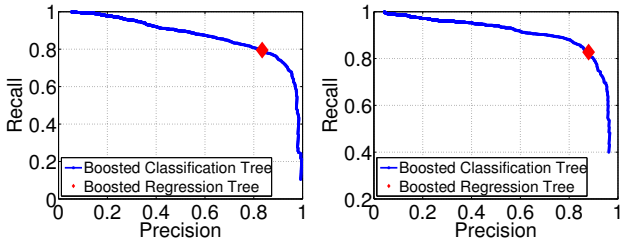
First, we compare the training time and the prediction overhead of the algorithms in Table 4. Linear regression, limited by its function form, is most efficient in terms of the training time. The training time for boosted regression tree is less than 2 seconds, and Gaussian process exhibits a long training time compared to other two algorithms. All approaches are comparable in the prediction overhead, meeting the target goal of 0.75 ms or less.

In general, the nonlinear algorithms demonstrate significantly superior prediction results. In particular, we observe that boosted regression tree outperforms Gaussian process regression, as we explain in Table 3. Note that even though introducing more inducing variables may lead to improvement in the accuracy of the Gaussian process regression (with additional training overhead), we do not pursue this direction as we find that boosted regression tree is sufficiently accurate for predictive parallelization (as we show in Section 5) and has lower training overhead.

***Comparison with prior work [16].*** Table 3 shows the precision and recall of regression algorithms using different sets of features and different thresholds (50 and 80 ms) for long-running query. We propose two prediction solutions, one based on all features (*i.e.,* term features + query features) and the other based on cheap features (*i.e.,* IDF + query features). Table 5 shows how the proposed solutions are compared to prior work [16] in terms of design considerations and cost.

The first two columns in Table 3 implement the proposed solutions that use all features and cheap features, respectively. We compare these to the prior work [16] in the third column that only uses term features from the given query without term rewriting. The results show that our approaches consistently provide higher precision and recall for the two thresholds. This is achieved by two factors. First, query features are important as shown in the previous section. Second, in computing term features, we account for rewritten terms in extracting term features, and this improves the accuracy in our solutions. For example, consider a query with two terms, $X$ and $Y$, and the query is rewritten into ($X$ or $X'$) and ($Y$ or $Y'$). A naive way to extract term features would be to treat the four terms independently and extract the four feature values. However, this approach considers unnecessary relation between $X$ and $Y'$ (and between $Y$ and $X'$) which leads to poor estimation. In contrast, we extract and combine the term features carefully. For example, term features for ($X$ or $X'$) are computed by summing up the values for each term.

Our approach has two key advantages when used for predictive parallelization: (1) We achieve high accuracy to effectively reduce 99th-percentile response time, while the prior work does not. As discussed in Section 4.1, our workload requires us to correctly identify at least 75% of the long-running queries (*i.e.,* demanding a recall of 0.75 or higher), while the predictor from the prior work [16] has a recall of 0.6485 as shown in Table 3. In this case, more than 1% of the queries that are long-running but misidentified as short-running will not be parallelized, causing high 99th-




(a) Threshold = 50 ms (b) Threshold = 80 ms

**Figure 5: Comparisons of precision and recall between boosted classification and regression tree.**

percentile latency. In comparison, our predictor improves recall to 0.7975 with statistical significance, to effectively reduce 99th-percentile response time; it also improves precision to reduce overhead due to false positives (*i.e.,* short-running queries misidentified as long-running ones and executed in parallel). (2) We achieve high accuracy using only cheap features, saving more than 90% of memory space needed for caching the features as shown in Table 5. Moreover, Table 3 also shows the precision of both our proposed features and cheap features is higher than a 95% confidence interval of [16]. These trends were consistent when using either 50 ms or 80 ms as a threshold of long-running queries.

***Regressor versus classifier.*** Lastly, we justify using a regressor for prediction. Using a classifier is less flexible, requiring retraining when the threshold changes. We compare the accuracy of a regressor to classifiers. Figure 5 shows that we can safely rule out classifiers, as the regressor achieves our prediction requirements with the high flexibility in choosing the threshold. The accuracy of the regressor is comparable to the classifiers. For example, in Figure 5(b), the point representing the accuracy of regressor is 0.85 in F1 score, which is the maximum of the entire curve representing classifier accuracy.

## 4.4 Summary

We derive the prediction requirements in terms of recall and precision as well as computational overhead. We discuss using both term and query features, and we identify a set of cheap features that are memory efficient and meet our accuracy requirements, while accounting for query augmentation and rewriting. We find that boosted regression tree shows the highest accuracy in prediction while incurring acceptable prediction overhead. Based on these observations, we employ **cheap features and boosted regression tree** in our predictive parallelization framework.

# 5. EXPERIMENTAL EVALUATION

We implement the boosted regression tree algorithm using cheap features in a commercial search engine to predict query execution time. Using the predicted query execution time, we modify the index serving nodes (ISNs) to selectively parallelize long-running queries only. Our results show that, compared with prior approaches that parallelize all queries, even under moderate-to-heavy load, predictive parallelization reduces the 99th-percentile response time by 50% from 200 ms to 100 ms. This allows us to operate the servers at high load while meeting the SLA, improving server capacity, and to process the same query workload using fewer number of servers. This section describes the experimental setup and results.

## 5.1 Experimental Setup

The ISN used in our evaluation has two 2.27 GHz 6-core Intel 64-bit Xeon processors, 32 GB of main memory, and runs production Bing code on Windows Server 2012. Each core supports 2-way hyperthreading, so the server runs up to 24 concurrent threads. The ISN manages a 160 GB web index partition on an SSD.

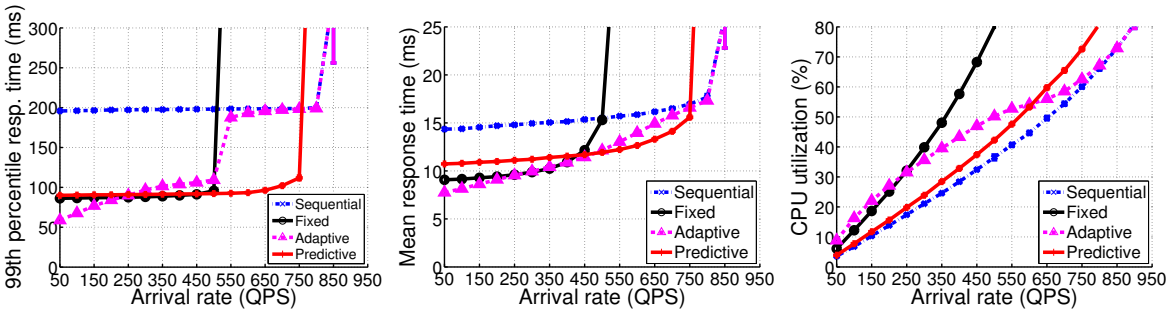
Our setup includes an ISN that answers queries and a client that plays queries from a trace of user queries. The trace contains 100K queries. We run the system by issuing queries following a Poisson distribution. We vary system load by changing the average query arrival rate. The ISN searches its local index and returns the top-4 matching results to the client with relevance scores. This is a standard testing configuration for the Bing search engine.

Our evaluation compares our proposed predictive parallelization strategy with other parallelization strategies that do not predict query execution time to selectively parallelize long-running queries. In particular, we compare with two state-of-the-art parallelization strategies, fixed parallelization and adaptive parallelization. Moreover, we also add sequential execution as a reference. In summary, we implement and evaluate the following four strategies.

- *Sequential execution* is a baseline system, which does not perform any query parallelization.
- *Fixed parallelization* parallelizes each query with 3 threads. This was the production configuration before our prediction framework has been deployed. Parallelism degree 3 is selected to meet the desired 99th-percentile response time of 100 ms.
- *Adaptive parallelization* is a recent approach on search engine query parallelization [14]. It dynamically selects the degree of parallelism on a query-by-query basis considering the current load and average parallelism efficiency of queries. This approach does not predict query execution time. Therefore, it cannot selectively parallelize long-running queries only.
- *Predictive parallelization* parallelizes only those queries that are predicted to be long-running using 3 threads and runs the other queries sequentially. We apply our proposed predictor using cheap features and boosted regression tree. We do not use the predictor from prior work [16] because its accuracy does not meet the selective parallelization requirements as discussed in Section 4.3. We consider a query as long-running if its predicted execution time is longer than a given threshold, for example, 80 ms.

The fixed, adaptive, and predictive parallelization all apply the same implementation to execute a query using multiple threads, as described in Section 3.1.

Query response time is the key performance metric for evaluating the parallelization policies. The response time, including both execution and queueing time, is measured at the ISN from the time that it receives the query to the time that it responds to the client. Both mean response time and 99th-percentile response time are reported. We do not report response quality (*i.e.,* relevance scores) because the




(a) 99th-percentile response time (b) Mean response time (c) CPU utilization

**Figure 6: Response times and CPU utilization of four different policies. We display the response time and CPU utilization with one standard deviation.**

parallelism decision does not affect the response quality [14]. No matter what parallelism degree is used, each query always performs a fixed amount of work for computing the relevance scores, thus producing the same response quality.

## 5.2 Experimental Results

This section explains the impact of prediction on three aspects of the ISN's execution: (1) the prediction overhead, (2) the response time reduction, and (3) the capacity improvement. Further, this section shows how to adapt the threshold in predictive parallelization at runtime for better resource utilization.

### 5.2.1 *Prediction Overhead*

As prediction itself adds additional work to query execution, it must be light-weight. Our measurements show that the running time of the prediction is about 0.6 ms, which is small at 4% of the average query execution time.

### 5.2.2 *Response Time Reduction*

***Comparison to fixed parallelization.*** Figure 6 shows both the 99th-percentile and mean response times as well as the CPU utilization for the four strategies for different loads, where the averaged response times and CPU utilization are obtained by 5 repetition to remove any biased results. For predictive parallelization, we use a threshold of 80 ms, so the queries predicted to execute longer than 80 ms run with 3-way parallelism.

Figure 6(a) compares the 99th-percentile response time for the competing policies. The x-axis represents the system load expressed as the average query arrival rate in queries per second (QPS). We study a wide range of load from very low to very high values. The y-axis represents the 99th-percentile response time of the queries. From the results we make three observations. First, parallelization significantly reduces the tail latency. Specifically, at light load (up to 500 QPS), both parallelization policies reduce the 99th-percentile response time from 200 ms using sequential execution to 100 ms. Second, at moderate to high load (500–750 QPS), predictive parallelization still achieves the same level of reduction, reducing the tail latency by 50% over sequential execution. In contrast, fixed parallelization fails to do so. This clearly shows that predictive parallelization is better than fixed parallelization in reducing the tail latency. Third, at extremely high load (>800 QPS), sequential execution has lower response time than fixed and predictive parallelization as there is no free core available to run queries in parallel. However, we do not operate our servers at such very high load because all policies violate the desired SLA of 100 ms on 99th-percentile response time.

Next we explain why predictive parallelization outperforms fixed parallelization at moderate to high load. Predictive parallelization enables judicious utilization of cores to long-running queries. At light load, there are ample idle cores and some waste due to parallelization overhead is tolerable. However, when the load increases, idle cores become scarce so they must be dedicated to long-running queries. More precisely, two factors contribute to the success of the predictive strategy. (1) The predictive strategy incurs a fairly small overhead because more than 99% of the short-running queries that are most common are not parallelized — recall that our prediction has high precision (>80%). Our predictive approach rarely identifies a short-running query as long-running and parallelizes it unnecessarily. It only parallelizes 3.71% of the total queries, of which about 3.3% are truly long-running queries. Figure 6(c) presents the average CPU utilization for the three policies under varying load, where we see a significant overhead with fixed parallelization. For example, at 400 QPS, this approach increases CPU utilization by 30% (from 28% of sequential execution to 58%), whereas the predictive strategy increases it by 5% only (from 28% to 33%). Using between 5% - 10% of additional CPU is a worthwhile cost to achieve 99th-percentile response time reduction by 50%. Such a small increase of CPU utilization is often affordable because search engine is not designed to operate at extremely high load [14] (in order to avoid queueing delay and quality degradation). Thus, there exist spare resources that parallelization can exploit. (2) True long-running queries, which are rare and show good parallelization speedup, are mostly identified and parallelized by the predictive strategy — the prediction has high recall (>80%). While 4% of all queries are long-running queries, 3.3% are identified and parallelized, successfully reducing the 99th-percentile response time.

Although search engines are optimized to reduce the high-percentile response time, we also present the mean response time results. Figure 6(b) shows that the trends for mean response time are similar to the 99th-percentile, except for the fact that under low load, the predictive approach has a slightly higher mean response time than fixed parallelization. This is expected: At low loads, fixed parallelization executes all incoming queries in parallel to aggressively make use of many idle cores, so it achieves more response time reduction on average. However, when the arrival rate goes up to

450 QPS, the ISN should carefully select which queries to parallelize. The predictive approach makes smart decisions, parallelizing long-running queries only, and thus it outperforms fixed parallelization.

***Comparison to adaptive parallelization.*** Adaptive parallelization is a recent approach on search engine query parallelization [14]. It dynamically selects the appropriate degree of parallelism based on system load: it executes all queries with a high degree of parallelism at low loads and reduces the degree of parallelism with increased load. Specifically, in Figure 6, at 50–100 QPS, the adaptive strategy parallelizes queries aggressively using 5 or 6 threads per query. When the arrival rate increases to 150–250 QPS, it more conservatively uses 3 or 4 threads per query. Under even higher load, it switches to sequential execution. Reducing the parallelism degree with increased load prevents overloading of the system under moderate/high load.

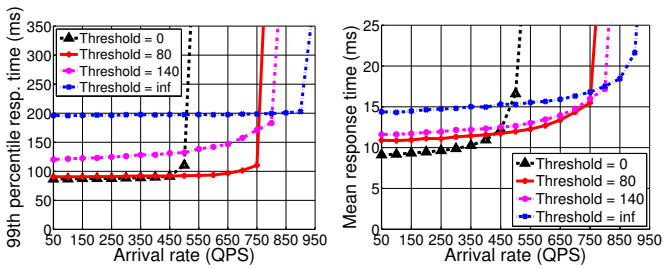
Figure 6(a) compares the 99th-percentile response time between adaptive and predictive parallelization, and it shows the advantage of predictive parallelization. By not wasting processor cores to parallelize short-running queries, the predictive strategy achieves the 99th-percentile response time of 100 ms up to 750 QPS. In contrast, adaptive has 99th-percentile response times exceeding 100 ms from 300 QPS, violating the SLA at fairly light loads. This is because adaptive parallelization does not differentiate long-running and short-running queries, which have different parallelization efficiency and different impact to tail latency. The adaptive strategy has to assign the available cores equally among all types of queries. Thus, as load increases, it reduces parallelism degree for all queries — long-running queries do not get sufficient parallelism, causing the increase in tail latency, while short-running queries waste resources on any parallelism they get, causing the ineffective utilization of cores.

Although adaptive parallelization has its limitations, it is still instructive to adapt query parallelization degree based on system load, *e.g.*, using sequential execution at extremely heavy load. We discuss how to adapt our predictive approach with system load in Section 5.2.4.

### 5.2.3 Capacity Improvement

Figure 6(a) also shows that, while meeting the same response time target, predictive parallelization also supports higher throughput. Assuming a desired 99th-percentile response time target of 100 ms, an ISN using the predictive approach sustains arrival rates up to 750 QPS, while using fixed/adaptive parallelization can only support up to 500 QPS. In other words, the predictive approach increases the server throughput by 50%. This indicates a 50% capacity improvement since we can use the same number of servers to process 50% more query loads.

Another method to show the benefits of the capacity improvement is to compute the number of required servers for a workload. Assume a total workload of $X$ QPS that a search engine needs to serve. Predictive parallelization requires $X/750$ servers while fixed parallelization needs $X/500$ servers: the predictive approach potentially saves $(X/500 - X/750)/(X/500) = 33\%$ of the ISNs to serve the same workload. As major search engines use thousands of servers in production, these savings are significant.




(a) 99th-percentile response time (b) Mean response time

**Figure 7: Latency with different threshold values.**

### 5.2.4 Adapting Threshold Values with Varying Load

To optimize high-percentile response time for all loads, a good adaptation of our predictor is to monitor the system loads and make the threshold a function of the load.

Figure 7 shows the benefits of applying different threshold values at different loads. Under light load, a smaller threshold value is preferred so more queries are parallelized; parallelization overhead is less of a concern under light load. As shown in Figure 7, at light load from 50 QPS to 450 QPS, we shall choose to parallelize all requests, equivalently, using a threshold of 0 ms. With increased load, a larger threshold value is preferred so only the long-running queries are parallelized. As shown in Figure 7, from 450 QPS to 750 QPS, the threshold value to optimize the 99th-percentile response time increases to 80 ms. With the further load increase from 750 QPS to 800 QPS, the best threshold value becomes 140 ms. At extremely heavy load (>800 QPS), sequential execution, which is equivalent to applying a threshold of infinity, produces the minimum latency.

This result shows that we could adapt the threshold values based on the system load to optimize response time for all loads. It also justifies why we choose to use a regression model instead of a classification model in the learning framework, as the regression model offers the flexibility to choose different threshold values.

## 6. RELATED WORK

***Prediction on search queries.*** The primary focus of prior work on predicting the execution time of web search queries is on identifying the traversal of the posting lists in the inverted index, which constitutes a large portion of query processing. Moffat *et al.* [17] show that the execution time of a query is related to the posting list lengths of its constituent query terms. However, under dynamic pruning strategies, the execution time can vary widely even for queries with the same number of postings because not every posting is scored [2]. Macdonald *et al.* [16] incorporate this observation to predict the execution time under dynamic pruning. In particular, various term-level statistics are computed for each term offline. When a query arrives, the term-level features are aggregated into query-level statistics, which are used as inputs to a regression model. These schemes, however, do not consider query rewriting and query features that highly influence query execution. Prior schemes do not investigate the cost of prediction when deployed to real systems. This work addresses these aspects.

Prior research shows how to predict the response quality of a query, and two approaches are proposed. First, pre-

retrieval predictors are calculated based on statistics from the query, without resorting to inverted index access [13]. Second, post-retrieval predictors have more information, exploiting the scores or contents of retrieved documents [1, 7]. These approaches are complementary to our work.

Some of the prediction frameworks are proposed to schedule queries in a replicated retrieval setting [16, 11] or to selectively prune query processing dynamically [25]. No attempt, however, has been made to use prediction combined with parallel query execution to reduce the tail latency.

***Search query parallelization.*** To reduce the response time of some search queries, we focus on parallelization, while index compression and index/result caching are alternative or complementary strategies; see a recent survey [27] for an overview of these techniques. To parallelize a query, Frachtenberg applies multithreading to achieve intra-query parallelization [10], by partitioning data into "equal" sized subsets of document IDs and letting each thread work on one subset. However, the matching documents can be unevenly distributed along the index space causing load imbalance.

The load imbalance problem has been addressed, for example, by using fine-grained partitioning [24] and by using decentralized communication [14]. The latter approach enables early pruning of each thread and deciding the parallelism degree dynamically. However, both approaches parallelize all queries, wasting computational resources. In contrast, this work identifies long-running queries and parallelizes them and runs short-running queries sequentially, leading to a more efficient utilization of the computational resources and to significant reduction in tail latencies.

Frachtenberg [10] proposes a heuristic to predict which queries to parallelize based on runtime information. A query first runs sequentially for a subset of the index partition, and the ratio of hits to documents is determined. If the ratio is above a threshold, the query is assumed to have good parallelization speedup and is theretofore parallelized; otherwise, the query runs sequentially. Although the ratio of hits to documents could correlate with query execution time, it does not capture other important factors contributing to the query execution time such as query complexity, pruning, etc. Moreover, compared with predictive strategy, this approach postpones the parallelism decision by running all queries sequentially at the beginning, and therefore long-running queries do not get the resources to speed up their execution at the earliest possible time.

# 7. CONCLUSIONS

This paper focuses on reducing the tail latencies for web search queries. To achieve this, we propose predictive parallelization, which is based on predicting query execution time. Our contributions include an accurate and efficient predictor for estimating query execution time and an effective parallelization strategy for reducing tail latency. To provide better accuracy and efficiency, we (1) use both query and term features, (2) propose cheap features, and (3) consider query rewriting. We implement this framework and evaluate its performance experimentally on a commercial search engine as compared to two competing strategies, fixed parallelization and adaptive parallelization. Our results show that predictive parallelization reduces the 99th-percentile response time by 50% from 200 ms to 100 ms compared with both approaches, which parallelize all queries. Moreover, it reduces the parallelization overhead and increases system capacity by more than 50%. This potentially saves one-third of production servers, constituting a significant cost reduction in a large-scale system.

# 8. ACKNOWLEDGMENTS

The work of S. Hwang was supported by the MSIP, Korea and Microsoft Research, under IT/SW Creative research program supervised by the NIPA (NIPA-2013-H0503-13-1009).

# 9. REFERENCES

[1] G. Amati, C. Carpineto, G. Romano, and F. U. Bordoni. Query difficulty, robustness and selective application of query expansion. In *ECIR*, 2004.

[2] R. Baeza-Yates, A. Gionis, F. P. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *ACM Trans. Web*, 2(4):20:1–20:28, Oct. 2008.

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

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

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

[6] N. Craswell, B. Billerbeck, D. Fetterly, and M. Najork. Robust query rewriting using anchor data. In *WSDM*, 2013.

[7] S. Cronen-Townsend, Y. Zhou, and W. B. Croft. Predicting query performance. In *SIGIR*, 2002.

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

[9] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, Feb. 2013.

[10] E. Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. *World Wide Web*, 12(4):441–460, Dec. 2009.

[11] A. Freire, C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda. Hybrid query scheduling for a replicated search engine. In *ECIR*, 2013.

[12] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[13] B. He and I. Ounis. Inferring query performance using pre-retrieval predictors. In *SPIRE*, 2004.

[14] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *EuroSys*, 2013.

[15] C. Macdonald, I. Ounis, and N. Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Trans. Inf. Syst.*, 29(4):17:1–17:28, Dec. 2011.

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

[17] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, June 2007.

[18] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, Sept. 1996.

[19] J. Pitkow, H. Schütze, T. Cass, R. Cooley, D. Turnbull, A. Edmonds, E. Adar, and T. Breuel. Personalized search. *CACM*, 45(9):50–55, Sept. 2002.

[20] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2005.

[21] K. M. Risvik, T. Chilimbi, H. Tan, K. Kalyanaraman, and C. Anderson. Maguro, a system for indexing and searching over very large text collections. In *WSDM*, 2013.

[22] E. Schurman and J. Brutlag. Performance related changes and their user impact. In *Velocity*, 2009.

[23] K. Sparck Jones. Document retrieval systems. chapter A Statistical Interpretation of Term Specificity and Its Application in Retrieval, pages 132–142. 1988.

[24] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, 2011.

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

[26] F. Zhang, S. Shi, H. Yan, and J.-R. Wen. Revisiting globally sorted indexes for efficient document retrieval. In *WSDM*, 2010.

[27] J. Zobel and A. Moffat. Inverted files for text search engines. In *ACM Computing Survey*, 2006.