Predictive Parallelization: A Framework for Reducing Tail Latencies of Web Search Queries

by

Myeongjae Jeon

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Scott Rixner, Chair
Associate Professor of Computer Science and Electrical and Computer Engineering

Alan L. Cox
Associate Professor of Computer Science and Electrical and Computer Engineering

Peter J. Varman
Professor in Electrical and Computer Engineering

Houston, Texas
April, 2014
ABSTRACT

Predictive Parallelization: A Framework for Reducing Tail Latencies of Web Search Queries

by

Myeongjae Jeon

We have become dependent on web search in our everyday lives. Web search services aim to provide fast responses to user queries, making the tail latency more important to reduce than the average latency. With modern multicore servers, intra-query parallelization becomes a desirable technique for reducing the query response time. Our workload characterization of commercial search engine servers shows that using parallelization to reduce the tail latency is challenging: (1) The search workload consists of mainly short-running queries that do not benefit from parallelism, and a few long-running queries which significantly impact the tail but exhibit high parallelism speedup. (2) The spare resources available to parallelize queries vary over time.

This thesis presents predictive parallelization, a framework designed for addressing these challenges and reducing tail latencies in web search. There are two fundamental techniques used as key elements of framework design. First, intra-query parallelization of index searching parallelizes each individual query with small overhead. The key idea is for a parallel search to mimic the sequential order of execution that almost never scans the entire index. Second, query execution time predictor identifies a majority of long-running queries through machine learning. The predictor covers
a comprehensive feature set to improve prediction accuracy while avoiding expensive features that have excessive requirements such as large memory footprints. In turn, heuristic algorithms in the framework exploit both query and system load information to decide parallelism degree on a query-by-query basis. At runtime, they selectively parallelize long-running queries with high parallelism efficiency and adapt the parallelism degree to system load. All of the techniques and mechanisms proposed in this thesis have been implemented and evaluated experimentally on production servers and workloads.
I would like to first thank my advisors, Prof. Scott Rixner and Prof. Alan L. Cox, for their guidance and support on my research. Their suggestions and directions over the past year have had a major influence on all aspects of my research and helped me develop professionalism. I would also like to thank Prof. Peter Varman for his helpful advice and valuable comments (especially for adaptive parallelism part) on this thesis.

I would like to thank my collaborators at Microsoft Research and Microsoft Bing. Dr. Yuxiong He and Dr. Sameh Elnikety at Microsoft Research have helped me with ideas, suggestions, and active discussions. I have no doubt that they influenced me similarly to my advisors. I am also grateful that they supported me with four times of internship at Microsoft Research. Dr. Chenyu Yan and its team members at Bing helped me use Bing platforms.

I would like to thank my friends at Rice CS including Mehul Chadha, Xu Liu, Shams Imam, Conglong Li, Yunming Zhang, and Linge Dai for their feedback and advice at the practice talk. Although I do not mention specific names, there are numerous faculty members and fellow students whom I have learned from directly and indirectly. Finally, I would like to thank my parents for their unwavering support throughout the years of my study.
## Contents

Abstract ii  
Acknowledgments iv  
List of Illustrations ix  
List of Tables xi  

1 Introduction 1  
  1.1 Contributions .................................................. 5  
  1.2 Organization ...................................................... 8  

2 Background: Web Search Engines 10  
  2.1 Requirements ................................................... 10  
  2.2 Search Architecture ........................................... 11  
  2.3 Query Processing in the ISN ................................. 13  
    2.3.1 Workflow .................................................. 13  
    2.3.2 Data Layout and Query Processing ....................... 13  
    2.3.3 Query Termination ...................................... 15  

3 Related Work 16  
  3.1 Adaptive Parallelism ........................................ 16  
  3.2 Reducing Response Time in Web Search .................... 18  
  3.3 Prediction on Search Queries ................................ 19  

4 Parallelizing the Processing of an Individual Search Query 21
4.1 Introduction ......................................................... 21
4.2 Opportunities ....................................................... 23
  4.2.1 Computationally-Intensive Workload ......................... 23
  4.2.2 Little Interference on Query Computation ................. 24
  4.2.3 Latency Variability ......................................... 25
4.3 Problems with Static Data Partitioning ....................... 26
4.4 The Design of Query Parallelization ......................... 28
  4.4.1 Index Data Partitioning and Processing ..................... 29
  4.4.2 Thread Communication ..................................... 31
4.5 Evaluation ......................................................... 32
  4.5.1 Experimental Setup ....................................... 32
  4.5.2 Speedup with Query Parallelization ....................... 34
  4.5.3 Response Quality with Query Parallelization .......... 36
4.6 Summary .......................................................... 37

5 Predicting the Execution Time of Individual Query ............. 39
  5.1 Introduction ..................................................... 39
  5.2 Requirements of Prediction Framework ....................... 41
  5.3 Features for Prediction Framework .......................... 43
    5.3.1 Term Features ........................................... 43
    5.3.2 Query Features ........................................ 46
    5.3.3 Cheap Features ......................................... 47
    5.3.4 Feature Analysis ....................................... 47
  5.4 Empirical Evaluation of Regression Algorithms ............. 50
  5.5 Evaluation ...................................................... 56
    5.5.1 Experimental Setup ..................................... 56
    5.5.2 Prediction Overhead ..................................... 58
    5.5.3 Response Time Reduction ............................... 58
5.5.4 Capacity Improvement ........................................... 61
5.5.5 Adapting Threshold Values with Varying Load .............. 62
5.6 Summary .......................................................... 64

6 Predictive Parallelization Framework ............................. 65
6.1 Introduction ......................................................... 65
6.2 Factors Impacting Predictive Parallelization Design ............ 68
  6.2.1 Query Execution Time ......................................... 69
  6.2.2 System Load .................................................... 70
  6.2.3 Parallelization Efficiency ................................. 71
6.3 Heuristic Algorithms ............................................... 72
  6.3.1 AP-Pred ..................................................... 72
  6.3.2 Timeline ..................................................... 76
6.4 Summary .......................................................... 80

7 Evaluation .......................................................... 81
7.1 The Importance of System Load and Parallelization Efficiency . . 83
  7.1.1 Response Time ................................................ 84
  7.1.2 Cost of Parallelism ......................................... 87
  7.1.3 Dynamic Changes in Query Arrival Rates ................. 88
  7.1.4 Comparison to Other Algorithms ......................... 89
7.2 The Power of Predictive Parallelization Strategies ............. 93
  7.2.1 Power of Information ....................................... 95
  7.2.2 AP-Pred and Timeline ..................................... 97
  7.2.3 Alternative Way to Selectively Parallelize Long-Running Queries 99
  7.2.4 Reducing Extreme Tail Latency ............................. 101
  7.2.5 Sensitivity Studies ......................................... 101

8 Conclusions ......................................................... 104
8.1 Future Directions ........................................... 106

Bibliography ....................................................... 109
# Illustrations

1.1 Simplified version of web search workflow stages. ........................................ 2
1.2 Histogram of the sequential query execution time of 70K queries. The x-axis is in 5 ms bins, and the y-axis is in log scale. Measurement setups are provided in Section 5.5. .................................................. 4

2.1 Partition-aggregate architecture of search engine. ................................. 11
2.2 Query processing in an ISN. ................................................................. 12
2.3 Sorted web documents and inverted index. ............................................ 14

4.1 CDF of the fraction of index data processed. ............................................. 26
4.2 An example of static coarse-grain processing with 2 threads. Both web documents and inverted index are accordingly bipartite. .............. 27
4.3 Comparison between dynamic fine-grain sharing and static fine-grain alternating using 3-way parallelism. ................................................. 30
4.4 Average execution time and speedup in parallelization for queries classified by their execution time. .................................................. 35

5.1 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 [1] (the black curve). .................................................. 49
5.2 Comparisons of precision and recall between boosted classification and regression tree. .................................................. 55

5.3 Response times and CPU utilization for sequential execution, fixed parallelization, and adaptive parallelization. ...................... 59

5.4 Response times with different threshold values. .......................... 63

7.1 Response time of sequential execution, fixed parallelism, and adaptive parallelism. The adaptive parallelism performs better than any of fixed parallelism configurations. It also achieves significant reductions in both mean and 95th-percentile latencies than sequential execution. 85

7.2 Distribution of selected degree in adaptive parallelism at 60 QPS. . . 86

7.3 Mean response time with changing arrival rates between adaptive parallelism and sequential execution. Moving average of 30 sec is used to compute mean. .................................................. 88

7.4 Response time comparison with other schemes, Binary and Linear, from prior work [2]. Adaptive performs better than both of them. . . 90

7.5 99th-percentile latency for Sequential, Fixed, Pred, AP, and AP-Pred. 95

7.6 AP-Pred and Timeline with two workloads where long-running queries have distinct parallelization speedups. ...................... 98

7.7 Timeline, RampUp, and Timeline /w correction for reducing tail latencies. .................................................. 100

7.8 AP-Pred with different groups for speedup curves. ...................... 102

7.9 CPU utilization vs. number of active threads vs. number of active threads for long-running queries with Timeline. ...................... 103
Tables

4.1 Response quality of query parallelism over sequential execution when using 6-way parallelism. Positive quality values indicate that query parallelism achieves a better quality compared with sequential execution. 37

5.1 Space of the features. 44

5.2 Top-15 features ranked by the importance obtained from boosted regression tree. Cheap features are shown in bold. 48

5.3 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. 51

5.4 Training time and prediction overhead comparisons for the three algorithms. 52

5.5 Design comparison of existing approach and our solutions. Memory usage is calculated based on one hundred million terms stored in a server. 53

7.1 Decomposition of all the query parallelism strategies according to parallelism change and information use, and their performance targets. Note that Pred, AP-Pred, and Timeline use prediction to estimate query execution time. 82
7.2 Comparison of CPU utilization. A policy & QPS with a **bold** value indicates worse response times than Sequential.
Chapter 1

Introduction

We have become dependent on web search in our everyday lives. Moreover, we have come to expect that the search results will be returned quickly to us and will be highly relevant to our search query. More formally, web search operates under a service level agreement (SLA) requiring short response times (e.g., 300 ms) and high result relevance.

Query processing in web search consists of multiple stages [3], as shown in Figure 1.1. It has a document index search that returns several hundreds of good matching documents for the keywords in the query. These documents are delivered to a second-phase ranking [4], where relevances of the documents are refined by a more complex ranking function. Out of hundreds of documents, a few tens of the “best” matching documents are selected for the user. This stage is followed by a snippet generator that makes the response for the user by adding a few sentence snippet for each of the final documents. The second phase consumes almost a fixed amount of time and is embarrassingly parallel, and the third stage consumes little time. The first phase, however, is along the critical path for query processing, accounting for more than two-thirds of the total processing time [5]. The focus of this thesis is to reduce response times of the first stage, which will significantly benefit search engines in many ways. For example, this saved time can be used to host a more powerful second-phase ranking, serve larger indices, and even add additional stages, which would improve the user experience. In this thesis, for brevity, we henceforth use the
In order to reduce response times, web search engines maintain massive indices of documents, which are partitioned across hundreds or thousands of servers. Quickly finding relevant documents in such a massive index commonly relies on the use of large-scale parallelism in two complementary ways — web search engines process multiple search queries concurrently, and they distribute the processing of each query over hundreds or thousands of servers. Nonetheless, in practice, achieving both high

Figure 1.1: Simplified version of web search workflow stages.
responsiveness and high quality is still challenging because these two requirements are often at odds with each other.

Web search engines have primarily been optimized to reduce the high-percentile (e.g., 99th-percentile) response time to consistently provide fast responses to almost all user queries [3, 6] — the 99th-percentile response time of $X$ means that 99% of requests have response time smaller than $X$. Long-running (or slow) query response times directly degrade user satisfaction and reduce revenues [7]. Reducing the high-percentile response time (also called the tail latency) is, however, a challenging task because the search engine workload exhibits high variability. Our measurements with a Microsoft Bing search server in Figure 1.2 show that most queries are short-running (or fast), with more than 85% taking less than 15 ms, and few queries are very long-running, taking up to 200 ms. In particular, the average of the execution time is 13.47 ms while the 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 long-running queries.

Parallelizing the processing of each individual query (i.e., intra-query parallelization) is a promising solution to reduce query execution time [8,9]. This is motivated by current hardware trends that a modern server has processors with more cores rather than faster clock speeds. With parallelization, multiple threads execute a query concurrently, using the available cores to reduce query execution time. When servers are lightly loaded and there are sufficient number of available cores, parallelizing the execution of all queries with high parallelism degree reduces their execution time, thereby reducing the response time.

However, parallelizing all queries with high parallelism degree is ineffective under
Figure 1.2: Histogram of the sequential query execution time of 70K queries. The x-axis is in 5 ms bins, and the y-axis is in log scale. Measurement setups are provided in Section 5.5.

moderate and high load because it comes with an overhead. As load increases, the system becomes saturated and latency increases rapidly. This motivates both the need for parallelism to reduce latency and the need for adaptation to prevent system saturation, depending on the system load. However, existing adaptive techniques [2] decide the request parallelism degree using only system load without considering request parallelization efficiency. This results in improvements over using a fixed degree of parallelism. However, it is hard to decide how to decrease the degree of parallelism with increased load without considering the efficiency with which an individual request can be parallelized. Either the scheme will be too conservative and not reduce latency as much as possible, or the scheme will be too aggressive and latency will increase beyond sequential at higher loads.
Moreover, parallelization is effective mostly for long-running queries because the overhead is small, achieving better speedup. In contrast, parallelizing short-running queries is ineffective, giving little performance benefit while consuming additional resources. Consequently, under moderate and high load, it is highly beneficial to selectively parallelize long-running queries to reduce the high-percentile response time, and to execute short-running queries sequentially to reduce parallelization overhead. This indicates that accurate prediction of the query execution time is essential for implementing effective selective parallelization schemes.

The search engine architecture is a large-scale distributed system, and speeding up long-running queries at each web search server is imperative to reducing the average latency as well as the tail latency in the cluster. A search query runs in parallel on a large number of servers, and the results are aggregated to respond to the user. Therefore, long latencies at any of the servers manifest as slow responses in the cluster, affecting both the average and the tail latency users perceive [3,6] — in fact, this is a general issue in applications based on the partition-aggregate model, such as MapReduce [10]. For example, if we want 50% users to receive a query response within 100 ms and the cluster has 70 servers, each server needs to reply within 100 ms with probability 0.99, i.e., the server needs to provide a 99th-percentile response time less than 100 ms. This result shows the importance of reducing tail latencies of each ISN, which is the focus of this thesis.

1.1 Contributions

This thesis presents new mechanisms for reducing high-percentile query response times in web search engines. The proposed mechanisms have been deployed in the Bing search engine, delivering performance improvements in a production environ-
ment that serves millions of user queries daily. The contributions of this thesis are as follows.

**Characterization of web search workload.** This thesis is the first to analyze sources of delay that compose the response time of a search query and to show how and when parallelization can reduce query response time. In particular, it describes why speeding up computation to reduce query response time makes good sense in web search — computation occupies the largest fraction in query processing and shows little interference among queries. Moreover, this thesis describes why it is crucial to only speed up long-running queries in order to effectively deliver low-latency responses to users — search queries have highly varying computational demands.

**Search query parallelization.** This thesis proposes a query parallelization approach called *dynamic fine-grain sharing* that partitions the indices of web pages and parallelizes query processing with little wasted work and good load balancing. When performing a web search, web pages that are more important are searched first for relevant results, and the search terminates early if further processing of remaining web pages is unlikely to yield any better results. The key idea of the proposed technique is for a parallel search to mimic this (sequential) order of execution on a number of small data chunks. Effectively, this ordering incurs a small amount of speculative execution and is less susceptible to thread scheduling or load imbalance among the chunks. Moreover, there is no degradation in the relevance of the search results.

**Query execution time predictor.** To reduce tail latencies in web search, where a small delay (*e.g.*, 10 ms) is critical, identifying long-running queries at the earliest possible time is highly desirable. This thesis proposes a framework for query execution
time prediction that allows identifying such queries early, even before query processing starts. This thesis systematically explores the design space of prediction features and learning algorithms to construct a predictor that incorporates query rewriting and uses a cheap set of features. Unlike a prior proposal [1], our predictor is accurate enough to identify a majority of long-running queries and efficient enough that it can be implemented in real search servers. The predictor is trained and evaluated using real user query logs and a production web index, reflecting many important aspects that have been neglected in the previous studies.

**Resource management algorithms.** This thesis proposes an adaptive parallelism strategy that decides the degree of parallelism for each query at runtime using system load and parallelization efficiency. When deciding the parallelism degree of a query, the key idea is to find the minimum cost that amounts to a combination of the estimated benefits to the parallelized query and the estimated delays on subsequent waiting queries. This enables the servers to execute queries with a high degree of parallelism at low loads, gracefully reduce the degree of parallelism with increased load, and choose sequential execution under high load. The adaptive strategy works no matter whether the search engine has prior knowledge of the computational cost of a query or not.

Further, this thesis presents new online algorithms that select the degree of parallelism for reducing tail latencies, not the average. The algorithms decide which queries impact the tail and how aggressively to parallelize them. Specifically, the algorithms exploit query execution time predictor to identify long-running queries (that directly affect tail latencies), and parallelize them with high efficiency and run other short-running queries sequentially. Further, the algorithms estimate parallelism efficiency
of long-running queries through system calibration, and keep tracking the amount of currently available processor resources. Our experimental evaluation shows that, compared to other competing strategies, the proposed algorithms reduce tail latencies considerably over a wide range of system loads. Additionally, this thesis presents an efficient fallback mechanism to recover from prediction errors and effectively reduce the extreme latency tail, e.g., 99.9th-percentile.

1.2 Organization

This thesis is organized as follows:

- Chapter 2 provides background information on the current state-of-the-art web search engine. It presents the common architecture of an index serving system in search engine, including its data layout and query processing.

- Chapter 3 discusses prior research that closely relates to the contributions of this thesis.

- Chapter 4 describes how to parallelize the processing of an individual query within the server. It describes the issues that make the parallelization of an individual query within a server challenging, and it presents a parallelization approach that effectively addresses these challenges.

- Chapter 5 introduces accurate and efficient prediction of query execution times, which is essential for implementing an effective selective parallelization scheme. It primarily explores the design of the learning framework including the requirements of prediction, the space of features, the space of regression algorithms.
• Chapter 6 introduces predictive parallelization, a framework designed for reducing tail latencies within a search server. It explains information that the framework exploits and resource management algorithms that choose the degree of parallelism at runtime for each query.

• Chapter 7 presents extensive evaluation of query parallelization policies including those proposed in this thesis.

• Chapter 8 explains conclusions and future research directions.
Chapter 2

Background: Web Search Engines

The focus of this thesis is on the index serving part of the web search engine that processes user search queries (interactive processing), rather than on the web crawler and indexer (batch processing). This chapter presents a common architecture of an index serving system, its data layout and query processing.

2.1 Requirements

Response time. Achieving consistently low response times is a primary design requirement for web search engines. A web search service is required to respond to a user query within a bounded amount of time. Since users are highly sensitive to the server’s response time, a small increase in query response time can significantly degrade user satisfaction and reduce revenue [7, 11, 12]. Therefore, it is important for service providers to lower the mean and high-percentile response times for their services.

Relevance. A web search engine must provide results that are relevant to the search keywords. Relevance is a complex metric to which many factors contribute. In general, the relevance of results improves with a larger web index containing more documents, with more features extracted from the documents, and with more sophisticated ranking functions. However, all of these factors contribute to greater resource demands and longer query processing times.
2.2 Search Architecture

The index serving system is a distributed service consisting of aggregators (also known as brokers) and index serving nodes (ISNs). An entire web index, containing information about billions of web documents, is document-sharded [13] and distributed among a large number of ISNs. When a user sends a query and its results are not cached, the index serving system processes and answers the query. Aggregators propagate the query to all ISNs hosting the web index, ISNs find matching documents for the query in their part, and aggregators collect the results to compute the response for the user query.

Figure 2.1 illustrates the partition-aggregate architecture, consisting of an aggregator and its ISNs. Due to the large number of ISNs, several levels of aggregators can be used. When a query arrives, an aggregator broadcasts it to its ISNs. Each ISN searches a disjoint partition of the web index and returns the top-\(k\) most relevant re-
results (currently $k = 4$) to the aggregator. The aggregator receives the results from the ISNs, and merges them to compute the response for the user query. Since a large number of ISNs are used to compute the query results, the aggregator imposes a deadline, typically 200 ms, to receive responses. Any ISN results received after the deadline are discarded, which potentially reduces the response quality and wastes processor and network resources. This architecture is consistent with the others discussed in the literature [5, 14].

ISNs are the workhorse of the index serving system. They constitute over 90% of the total hardware resources. Moreover, they are along the critical path for query processing and account for more than two-thirds of the total processing time.
2.3 Query Processing in the ISN

2.3.1 Workflow

Figure 2.2 shows the workflow for query processing in an ISN. The ISN is a multi-threaded server, capable of processing several queries concurrently for higher efficiency. Newly arrived requests join the waiting queue. When the waiting queue becomes full, new requests will be dropped. An ISN manages a number of worker threads. Each worker processes a single request at a time. When a worker completes a request, 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 experience delay in the waiting queue (i.e., queueing time), in addition to their processing time (i.e., execution time). The response time of a query consists of its queueing time and execution time.

The number of workers is at least equal to the number of cores in the system. Typically, there are a larger number of workers than the number of cores, to account for workers who block on I/O operations. However, blocking on I/O is rare because the indices are partitioned, which promotes locality at the ISNs.

2.3.2 Data Layout and Query Processing

When an ISN worker thread processes a query, it searches its web index to produce a list of documents matching the keywords in the query. It then ranks the matching documents. This is the most time consuming part of the ISN’s work because the ranking function extracts and computes many features of the documents.

The ISN uses an inverted index [15] (also called posting list) to store information about documents. Figure 2.3 shows an example of the data layout of the inverted
index, where the documents are sorted based on their static rank. The static rank of a document, similarly to PageRank [16], depends only on its own features such as its popularity or importance; it does not depend on the query. When a worker matches and ranks the documents by following the inverted indices, it processes the documents with higher static ranking first as they are more likely to contribute to the top matching results of the query. During the ranking process, each match is also “dynamically” scored to compute a relevance score. The relevance score of a document depends on its static rank as well as on many other features, including for example, term frequency in the document [17], the proximity of the keywords, user location and context, etc.

When the search query has multiple keywords, the ISN finds matches with all/many keywords appearing in the same document. As queries are normally processed in the conjunctive mode [5], this is done by intersecting inverted indices for all keywords. The inverted indices are sorted based on document static rank, so this is a simple merge-join process on sorted data. The following explains a common algorithm for performing the intersection. For each element in the shortest index, a search method (e.g., binary or interpolation search) is performed in the other indices.
A matching document is found if the element belongs to all/many indices. Otherwise, the searching immediately stops, ignoring remaining indices, and moves to the next element in the shortest index. After iterating over all elements in the shortest index, the intersection terminates.

The inverted index is not implemented as a conventional array. Its implementation is similar to that of a skip list. The seek cost from one element to another is not constant but logarithmic on the size of the inverted index for a particular keyword. Moreover, the inverted indices are compressed, exploiting more advanced data structures that trade-off among space, decompression cost, and sequential and random access efficiency.

### 2.3.3 Query Termination

Once an ISN finds results for a query that are good enough, it terminates the search and returns the results. This is called *early termination* or *stop early* \([4,17-20]\). The ISN predicts how likely the results will improve from scoring the remaining documents and terminates the execution when the probability is low. Early termination is effective because the inverted indices are sorted based on the static rank of the web pages, which lists important and popular web pages first. Web pages processed earlier are more likely to rank higher and contribute to the top results of the query. The ISN has an algorithm that takes the current top documents and the static ranking of the later documents as inputs, and it decides if the termination condition is satisfied. The specific details of this algorithm are beyond the scope of this thesis. With early termination, there is a small, but non-zero, probability that more relevant results might be missed.
Chapter 3

Related Work

This chapter discusses prior research that closely relate to the contributions of this thesis. It is organized as follows. Section 3.1 describes various proposals to perform adaptive parallelism in other application domains. Section 3.2 presents approaches dealing with response time in web search, including query parallelization and general optimizations in the search server. Finally, Section 3.3 includes some related work on prediction attempts on search queries and use cases.

3.1 Adaptive Parallelism

Many interfaces and associated run-time systems have been proposed that adapt parallel program execution to run-time variability and hardware characteristics [21–28]. They focus on improving the execution of a single job with respect to various performance goals, such as reducing job execution time and improving job energy-efficiency. However, they do not consider a server system running concurrent jobs. Simply applying parallelism to minimize the execution time of every single job will not minimize the mean response time across all jobs. Our work focuses on such an environment with many jobs where the parallelization of one job may affect others. We decide the degree of parallelism for a job based on the impact both on the job itself and on the other jobs.

Sharing resources adaptively among parallel jobs, which is often referred to as
adaptive job scheduling, has been studied both empirically [29, 30] and theoretically [31, 32]. However, they focus on a multiprogrammed environment instead of an interactive server with latency constraints. In a multiprogrammed environment, it is common for jobs to have different characteristics that the scheduler does not know a priori. Thus, work in this area uses non-clairvoyant scheduling wherein nothing is assumed or known about the job before executing it. The scheduler learns the job’s characteristics and adjusts the degree of parallelism as the job executes. In contrast, in the web search server that we study, requests share a lot of similarities because they process user queries using the same procedure. So the scheduler can exploit more information, such as the average execution profile for the requests, to improve scheduling decisions. Moreover, as interactive requests often complete quickly, there is limited time for the scheduler to learn and adapt to the characteristics of an individual request. Instead, we use a predictive model that makes the decision before executing a job using the available job information.

Adaptive resource allocation for server systems has been explored; most of this work, however, focus on dynamically allocating resources to different components of the server while the individual requests still execute sequentially [33,34]. In contrast, Raman et al. have proposed the Degree of Parallelism Executive (DoPE), an API and run-time system for adaptive parallelism [2]. The API allows developers to express parallelism options and goals such as maximizing throughput and minimizing mean response time. The run-time has mechanisms to dynamically decide the degree of parallelism to meet the goals. Like this paper, one goal of parallelization in their work is to minimize response time of requests in a server system. They present two heuristic algorithms for deciding the degree of parallelism. We implement both algorithms, called Binary and Linear in Section 7.1, and we compare experimentally
them to our adaptive algorithm.

## 3.2 Reducing Response Time in Web Search

To reduce the latency of semantic web search queries, Frachtenberg applies multithreading to achieve intra-query parallelism [8]. To parallelize a query, the ISN partitions the data into equal-sized subsets of document IDs and each thread works on one subset. This is identical to the static coarse-grained processing approach that is discussed in Section 4.3. A key assumption of that work [8] is the following: Matching documents of the query in web index are evenly distributed and thus load balancing among threads is not of great concern. However, this is not always the case because matching documents of a query may not be evenly distributed along the index space. Techniques that address load imbalance among threads when intra-query parallelism is exploited have been proposed in [9, 35].

Tsirogiannis et al. explore efficient algorithms that partition the inverted indices of a search query to balance the load among threads within a guaranteed error bound [35]. The algorithms further exploit good probing orders on the partitioned indices to reduce the intersection overhead. Their techniques can be combined with ours to enhance the load balancing.

Tatikonda et al. propose a fine-grained intra-query parallelism approach in web search [9], which has some similarities to our query parallelization technique. Both approaches partition the indices into a number of fine-grained tasks, which enables good load balancing among threads. Moreover, both maintain a shared data structure among all threads of a query to store the top ranked results. The two approaches have several fundamental differences. First, while they propose to use a producer-consumer model to generate and assign tasks, we employ a decentralized method in
which threads coordinate and acquire its next index partition using a single atomic fetch-and-increment operation and the top results are kept in each thread’s context. This enables each thread to promptly invoke early termination to reduce the overhead of speculative execution with small synchronization overhead. Moreover, another key focus of our work is to decide the parallelism of a query adaptively based on query execution profile and system load, which is not studied in the prior work [9]. Lastly, we implement our approach and evaluate it using more recent data sets.

Graphics processors (GPU) [36] and SIMD instructions [37] have been used to parallelize the processing of an individual search query. Also, there are studies on reducing the response time for web search queries across different system components, for example, optimizing caching [38–40] and prefetching [41] to mitigate I/O costs and improving network protocols between ISNs and aggregators [42, 43]. Moreover, response quality can be traded off to reduce response time, especially under heavy load or other exceptional situations in which the servers could not process queries fast enough [44]. These studies are complementary to our work.

3.3 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. [45] 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 [46]. Macdonald et al. [1] 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 thesis 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 [47]. Second, post-retrieval predictors have more information, exploiting the scores or contents of retrieved documents [48, 49]. These approaches are complementary to our work.

Some of the prediction frameworks are proposed to schedule queries in a replicated retrieval setting [1, 50] or to selectively prune query processing dynamically [51]. No attempt, however, has been made to use prediction combined with parallel execution of query in order to reduce the tail latency.

Frachtenberg [8] 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 therefore 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 our 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.
Chapter 4

Parallelizing the Processing of an Individual Search Query

4.1 Introduction

Today, web search engines commonly achieve large-scale parallelism in two complementary ways. First, the entire web index containing billions of documents is partitioned across hundreds or thousands of servers, and each query is processed by these index servers concurrently. The final result of a query is aggregated from the index servers and sent back to the users. Second, with modern multicore servers, multiple requests are processed concurrently within each server. However, this inter-query parallelization within a server does not reduce the execution time of an individual query, because each request is still executed sequentially within the server. Nonetheless, processing multiple requests in parallel is valuable because it improves the server’s throughput and potentially reduces the time that a query waits for execution.

This chapter illustrates a third complementary way of achieving parallelism, *i.e.*, intra-query parallelization of index searching within a multicore server. Specifically, the focus of this chapter is on how to parallelize a single query within one server that hosts a fragment of the web index, such that multiple processor cores cooperate to service the query. This is motivated by having processors with more cores rather than faster clock speeds, making intra-query parallelization an effective technique to reduce query response time. Our study shows that parallelization is effective for reducing
query response time because computation occupies the largest fraction of the query execution time when compared with other factors such as network, queueing, and I/O latency. The computation time can be greatly reduced using parallelization.

At the query level, web search is embarrassingly parallel. Each server searches its fragment of the index. When performing a web search, documents with a higher static rank are searched first for relevant results. A web search can terminate early if it discovers that scoring the remaining documents is unlikely to yield any better results for the current query. Therefore, a sequential search of the ordered index almost never scans the entire index. In fact, about 30% of queries need only search 10% of the index.

Unfortunately, this early termination of web searches makes intra-query parallelization challenging. It is not obvious how to partition tasks within a single query in order to effectively search the index without performing large amounts of unnecessary work. Since a sequential search can terminate the search early, a parallel search will almost always scan more of the index. Documents with a higher static priority are being scanned concurrently with documents with a lower static priority. So, the time spent looking at the lower priority documents is wasted if enough relevant results are found in the higher priority documents.

To reduce such wasted work, this chapter presents a dynamic fine-grain sharing technique that parallelizes each request while preserving the sequential order of execution. This technique limits wasted work and achieves good load balancing.

The rest of this chapter is organized as follows. Section 4.2 and 4.3 introduce opportunities and challenges, respectively, of query parallelization. Section 4.4 addresses challenges: how to partition and process index data. Section 4.5 presents our experimental results, and finally, Section 4.6 summarizes the chapter.
4.2 Opportunities

We introduce intra-query parallelism into the ISN to reduce tail latencies. This section discusses important workload characteristics of web search that create opportunities for such query parallelization. The methods for collecting the data are the same as described in Section 5.5. For brevity, we henceforth use the term query parallelization to mean intra-query parallelization, unless specified otherwise.

4.2.1 Computationally-Intensive Workload

There are mainly four sources of latency that compose the response time of a query in the request-response flow between an ISN and an aggregator: (bidirectional) network delay, queueing delay, reading index data, and computation. By investigating how much each factor contributes to the response time, this section supports the optimization of computation rather than the others to reduce query response time.

Network. Network impacts little on query response time. Network latency is measured at a production cluster from the time that an ISN completes the processing of a query to the time that the aggregator receives its response. The measurements show that the latency is 2.13 ms on average and 3.47 ms for the 99th-percentile. Since the request-response flow is bidirectional, the actual amount comes to roughly twice the measured latency. Nonetheless, it is small considering queries that run up to 200 ms.

Queueing. Many commercial data centers have server utilization less than 50% for certain production workloads [52–54]. In Bing, search servers operate at between 30% and 50% of the maximum level for most of the time. Search engines have low CPU utilization by design, to avoid queueing and to provide consistently high quality
responses; the servers are also over-provisioned for the events like a failure of a cluster or an entire data center. We observe that under even a higher CPU utilization at 73%, the queueing time at the ISN is only 0.35 ms on average.

**I/O.** The inverted indices are partitioned across a large number of ISNs, each holding 10s of GB of DRAM for index caching, to create locality and reduce disk I/O accesses. In our production servers, I/O bandwidth consumption is only around 0.3 KB/s. More importantly, latency for I/O operations is minimal for two reasons. First, the web index is stored on an SSD to enable fast index data fetching and to avoid performance degradation from interleaved I/O accesses. Second, a worker thread is busy waiting for I/O completion [55,56] in order to react to it promptly and to avoid delay with context switching. Hence, I/O is not a limiting factor in the response time.

**Computation.** In contrast, computation occupies the largest fraction in query response processing. The amount is much larger for long-running queries that contribute to high-percentile latencies. Specifically, the average of the service demand is 13.47 ms, which is 3.16 times the average of the network delay. The 99th-percentile service demand is 200 ms, which is far larger than any other factors. Therefore, the query latency reduction is best achieved by speeding up the computation.

### 4.2.2 Little Interference on Query Computation

Computationally-intensive workload with little interference among queries indicates a good opportunity for query parallelization. Moreover, background services running on a server have almost no impact on the query processing.

The ISN performs complex calculations, such as computing the relevance of matches, which makes the ISNs computationally intensive. In fact, web search ex-
hibits higher instructions per cycle (IPC) than traditional enterprise workloads [57]. This also means that processor cache misses are relatively infrequent. Therefore, we would expect queries running in parallel on an enterprise-class server to exhibit little slowdown due to memory contention.

Our measurement data confirm our expectation. A query only incurs less than 5% slowdown while running together with other queries at 50% CPU utilization compared with running alone. It demonstrates that the interference among concurrently running queries is negligible.

An ISN runs several background services performing maintenance activities (such as updating index, logging server events, and monitoring hardware failures). These services are infrequently invoked and consume mainly I/O bandwidth, thus having almost no impact on interactive query latency. Moreover, some effort has been made to safely collocate latency-sensitive applications with other tasks [58–60].

4.2.3 Latency Variability

Search queries have varying computational demands. Figure 1.2 shows the distribution of request service demands from a Bing ISN. Most queries are short-running, with more than 85% taking below 15 ms. However, few queries are very long-running taking up to 200 ms. This variability is observed fairly consistent across a few hundred ISNs in a Bing index serving cluster, and it shows the importance of reducing tail latency of each ISN to deliver low-latency responses to users.

Many factors affect execution time of queries, and two typical factors are responsible for the long time. First, these queries tend to have more documents to match and score. This requires frequent invocations of ranking inference engines, which usually consume a significant number of processor cycles for scoring each match. Second,
these queries involve the intersection of a larger number of inverted indices. It is known that the average latency of queries with ten keywords is approximately an order of magnitude greater than that of queries with only two keywords [9].

4.3 Problems with Static Data Partitioning

As a result of early termination, it is common that documents with low importance are rarely visited during sequential execution. Figure 4.1 shows the cumulative distribution of the fraction of inverted indices visited by Bing ISNs. The measurements indicate that about 30% of the queries use only 10% of the documents. Moreover, more than 38% of the queries never need to score matches on the second half of the documents. This poses challenges to query parallelization. We present two basic approaches and discuss their performance limitations.
Figure 4.2: An example of static coarse-grain processing with 2 threads. Both web documents and inverted index are accordingly bipartite.

One possible approach, static coarse-grain processing, partitions the data into the same number of chunks as there are threads assigned to the query. This approach may, however, introduce a large amount of speculative execution because all matching documents will be processed. Consider a simple example: the inverted index of a query is partitioned into two chunks, and each chunk is processed by a different thread, as shown in Figure 4.2. Much of the work performed by the second thread is likely to be useless, since 38% of the queries (Figure 4.1) are unlikely to yield any relevant results from the second half of the index.

Another approach, static fine-grain alternating, partitions the data into many, smaller chunks, and assigns these chunks to the threads in an alternating fashion. For example, the partitioned chunks of the inverted index are labeled based on their static order to have chunks 1 to $N$. When two threads are used, one thread processes odd chunks while the other thread processes even chunks, and each thread processes their chunks based on their static order so more important chunks are processed first. While processing the chunks, the threads communicate with each other to merge
the top results they have found so far. Thus, a thread can terminate early without processing all assigned chunks once the current top results are good enough. When concurrent threads of a query advance at a similar rate, static fine-grain alternating behaves similarly to the sequential order of execution and it helps to reduce the amount of speculative execution.

The static fine-grain alternating approach still has two issues: (1) The amount of computation per chunk and per thread may be uneven, resulting in unbalanced load among threads. (2) Multiple threads of a query may not always be co-scheduled since there are more threads than processor cores. This can lead to different completion times for the threads. Moreover, when one thread is delayed and does not process its most important chunks, other threads may perform a larger amount of speculative work because early termination does not occur promptly.

Therefore, to parallelize web search, the primary challenge is to partition and process the index data efficiently to reduce additional work.

4.4 The Design of Query Parallelization

This section presents how to partition and process the index data in parallel to reduce wasted, speculative work given a fixed parallelization degree. This section presents a parallelization technique — *dynamic fine-grain sharing* — that mimics the sequential order of execution with small synchronization overhead and good load balancing among threads. This keeps the overhead of parallelization low while producing results with comparable quality to sequential execution. This section first describes how to partition and process the index data using concurrent threads. Then, it discusses the communication among threads and how to reduce the overhead.
4.4.1 Index Data Partitioning and Processing

Dynamic fine-grain sharing partitions the index into small chunks. When a thread becomes idle, it grabs the most important unprocessed data chunk. To implement this technique, a counter is only needed to represent the ID of the next most important unprocessed chunk. An idle thread uses an atomic fetch-and-increment operation to claim this chunk and update the counter to reference the next most important chunk. The threads communicate with each other to merge the top results they have found so far. When a thread finds that the current top results are good enough and the later chunks are unlikely to produce better results, the thread stops processing the query, reducing computation overhead.

Compared to the static fine-grain alternating approach in Section 4.3, dynamic sharing more closely mimics the sequential order of execution regardless of thread scheduling. It also achieves better load balancing. Even when threads of a query are not co-scheduled, the active threads of the query still process the chunks based on their sequential order. Moreover, as the dynamic sharing is similar to having a shared global queue of tasks, good load balancing is attained. Nonetheless, the synchronization overhead is quite small, because obtaining a task only requires a single atomic fetch-and-increment operation.

Figure 4.3 compares the performance between the static alternating and dynamic sharing techniques. The experimental setup is the same as that in Section 4.5. The figure shows that dynamic sharing always outperforms static alternating with lower mean and 95th-percentile response time. Under light load from 10 to 50 QPS, the performance gap between the static and dynamic approaches is smaller: the static approach takes 3% to 12% longer than dynamic sharing for both the mean and the 95th-percentile response time. The inefficiency of static alternating is mostly caused
by the uneven work distribution among chunks. Under heavy loads, however, both thread scheduling and uneven work distribution among chunks affect response time. Specifically, due to the imbalance of OS thread scheduling, when a delayed thread did not get a chance to process its most important chunks, other threads may incur a
larger amount of wasted, speculative work. These threads evaluate lower static-rank documents that are unlikely to be the top results, increasing response time. As shown in the figure, under 70 QPS, the mean and 95th-percentile response times for static alternating are 75% and 78% higher respectively than dynamic sharing.

4.4.2 Thread Communication

Threads must communicate to determine when to terminate the search. There are different approaches for threads to communicate and share information. A simple way is for all threads to maintain one global data structure and update this data structure whenever a thread finds new results. This approach reduces the size of critical section as it does not necessarily hold a lock for all threads. However, the problem with this simple approach is the potential for contention among threads. Processing of a query often evaluates thousands of matches, and thus synchronizing for every new match is expensive.

Our approach uses batched updates. Each query maintains a global heap and each thread executing the query maintains a local heap. A thread periodically synchronizes the information in its local heap with the global heap, not directly with other threads. Specifically, a thread’s local heap is synchronized with the global heap only when a thread completes its computation for a chunk. Batch updates allow threads to see the results of other threads soon enough to facilitate effective early termination without introducing much synchronization overhead. The chunk size is tunable to tradeoff between the synchronization overhead and the amount of speculative execution, but data compression limits the smallest chunk size. The appropriate chunk size is determined empirically.
4.5 Evaluation

4.5.1 Experimental Setup

**Machine setup and workload.** The ISN used in this 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. The ISN manages a 90 GB web index partition on an SSD and uses 22 GB of memory to cache recently accessed web index data. The web index was created in March 2012. The number of worker threads is set to twice the number of cores, considering workers sometimes blocking on I/O accesses. The OS scheduler assigns the threads of a query to the available cores.

Our setup includes an ISN that answers queries and a client that plays queries from a production trace of Bing user queries. The trace contains 100K queries and was obtained in June 2012. We run the system by issuing queries following a Poisson distribution in an open loop. We vary system load by changing the average arrival rate of queries, *i.e.*, queries per second (QPS). 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. With more results returned from each ISN, the processing time of queries may increase. Our parallelism techniques work in all cases.

**Index chunk size.** We empirically explore various chunk sizes and use the one resulting in the smallest average query execution time. In our experiments, the index space in the ISN is partitioned into 200 chunks. A query processes 117 chunks on average as some queries terminate early. We observe that the performance is good overall and stable if the average number of chunks processed per query is between 100 and 200. With larger chunk sizes, the overhead due to speculative execution is too
high, and the load among the threads is not balanced. These are no longer a problem with smaller chunk sizes, but the benefits are offset by the synchronization cost and the seek cost across chunks.

**Performance metrics.** This section compares query parallelism to sequential execution and presents the following performance metrics:

- *Parallelism speedup.* Execution time and speedup results are measured at the ISN by using our parallelization technique. To ensure no interference among queries, the execution of one query is isolated from that of other queries.

- *Response quality.* To evaluate the quality of a response to a parallelized search query, the relevance scores are compared in the response to the relevance scores in sequential execution. The details of this methodology will be explained next.

The performance metrics are collected after the ISN index cache is warmed to obtain steady state measurements.

**Response quality metric.** The quality of the response is an important measure for search engines, and parallelism is desired to provide very similar response quality to sequential execution. The quality metric we use is a relative comparison of the quality of the results from a *baseline* run (sequential) to a *test* run (parallelized).

To compare the quality of search results from two different runs, do the following. First, perform a pairwise comparison between relevance scores on the sorted results from the two runs. So, the “best” result from the baseline run will be compared to the “best” result from the test run, and so on. Then, for each pairwise comparison, the quality metric is incremented, decremented, or unchanged by a given weight (explained below) based on whether the test run has higher, lower, or the same
relevance. This will produce a final “relative quality” metric that tells us how much better (or worse) the results from the test run are.

In the evaluation, the weights for the increment or decrement are assigned in two ways. First, proportional weights gives each of the top-4 documents the same weight. Second, another quality metric, exponential weights, assigns higher weights to higher ranking documents, as higher ranking documents are more important, so they are more likely selected by the top-level aggregator to return to the user. 8/15, 4/15, 2/15, and 1/15 are assigned for the highest to lowest results.

4.5.2 Speedup with Query Parallelization

Query workload is not homogenous, and different queries react differently to intra-query parallelization. While there are many short-running queries that do not benefit from parallelization, there are still a significant number of long-running queries that parallelize well, achieving good speedup. In particular, the parallelization of these latter queries will significantly reduce the tail latency.

To show that parallelization is effective only for long-running queries, queries are classified into three classes based on their execution times, and we show the speedup, which is the sequential execution time divided by the parallel execution time, with different parallelism degrees in Figure 4.4. 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. In addition, for medium-running queries, which run between 30 and 80 ms, the speedup is modest, less than 2 for parallelism degree 6.

Long-running requests achieve good speedup for two reasons. First, the dynamic
Figure 4.4: Average execution time and speedup in parallelization for queries classified by their execution time.

(a) Short-running queries (0–30 ms)

(b) Medium-running queries (30–80 ms)

(c) Long-running queries (>80 ms)

data partitioning and processing scheme effectively controls the amount of speculative execution. As compared to a sequential execution, a parallel execution with degree \( p \) is unlikely to visit more than \( p - 1 \) extra chunks. Since sequential executions of the long-running requests visit a few hundred chunks on average, the additional \( p - 1 \) chunks constitute a small percentage of the overhead. Second, our parallelization
focuses on the matching and ranking of the inverted indices to find the top documents of a query. However, there are parts of the query processing that are not parallelized, e.g., query parsing and rescoring of the top results. For the longest-running queries, the non-parallelized part constitutes a very small part of the total execution time.

Although reducing the chunk size can reduce the amount of speculative work, it may add to other overheads. For example, reducing the chunk size will increase the frequency of moving between non-adjacent chunks. This has a non-trivial cost because the inverted index cannot be accessed as an array. Also, it will increase the synchronization overhead.

There are two important conclusions from Figure 4.4. First, it is important to parallelize long-running queries to reduce response time. Without parallelization, they are more likely to have long response times. 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.

4.5.3 Response Quality with Query Parallelization

Table 4.1 presents the response quality of query parallelism against sequential execution when using 6-way parallelism. Sequential execution becomes a baseline run to compute the response quality of query parallelism (which is the test run). Therefore, if query parallelism returns better relevance scores of search query responses than sequential execution under a weighting strategy, the figure will report a positive value at that policy.
Table 4.1 shows that query parallelism produces slightly better response quality than sequential execution in both proportional weights and exponential weights. Query parallelism returns positive qualities for both cases, meaning that its relevance scores for search queries are overall higher. This is because parallel execution may cover a (slightly) longer prefix of the inverted indices due to speculative execution, having more documents scored for a search query than when it is executed sequentially.

### 4.6 Summary

A web search query made to Microsoft Bing has been parallelized by distributing the query processing across many servers. Within each of these servers, the query is, however, processed sequentially. Although each server may be processing multiple queries concurrently, with modern multicore servers, parallelizing the processing of an individual query within the server may nonetheless improve the users experience by reducing the response time.

This chapter explored effective techniques to parallelize the query execution in web search servers to reduce their mean and high-percentile response time. This chapter introduced a dynamic fine-grain sharing technique that mimics the sequential order.
of execution to parallelize each individual request with small speculative execution and good load balancing. Moreover, with the proposed parallelization technique, no degradation in the relevance of the search results was observed as compared to sequential execution.

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. Therefore, this thesis 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*. 
Chapter 5

Predicting the Execution Time of Individual Query

5.1 Introduction

Web search engines are optimized to reduce the high-percentile (e.g., 99th-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, this chapter introduces a framework for accurate prediction of the query execution time which makes query parallelization effective [61]. The primary goal is to identify and speed up long-running queries only to lower the high-percentile response time efficiently.

In particular, 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 [1] 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.

To show the power of using the prediction for query 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 show that under moderate to heavy load, this predictive parallelization strategy reduces the 99th-percentile response time by 50% (from 200 ms to 100 ms) compared with prior approaches that parallelize all queries. In this chapter, this strategy is referred to as predictive parallelization (or predictive parallelism framework) for brevity. However, note that predictive parallelization indicates a framework designed for reducing tail latencies, which will be discussed in detail in Section 6.

The contribution of this chapter is the design and evaluation of the prediction framework:

- Improved prediction: We systematically explore the design space of prediction features and learning algorithms to construct a predictor that incorporates
query rewriting and use a cheap set of features that achieve 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. This predictive parallelization is evaluated in a production environment of Bing search engine. The results show that the proposed solution significantly reduces the 99th-percentile latency compared with sequential execution and a state-of-the-art parallelization policy.

The rest of this chapter is organized as follows. Section 5.2 explains the requirements of prediction framework. Section 5.3 and Section 5.4 present the space of features and regression algorithms, respectively, for prediction, and conclude which ones to use. Predictive parallelization is evaluated experimentally in Section 5.5. Section 5.6 summaries this chapter.

### 5.2 Requirements of Prediction Framework

Predictive parallelization imposes four requirements on the prediction framework: tail latency, misprediction cost, prediction overhead, and flexibility. The following is 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.
**Tail latency.** As explained in Section 5.1, 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 guarantee a reduction of the 99th-percentile response time, 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 workload of Bing search 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.

**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. Section 5.5 elaborates the importance and shows which precision values are required.

**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.

**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 5.5). 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 5.4), which supports our design decision.

### 5.3 Features for Prediction Framework

This section describes the features that can be used for prediction and analyzes the importance of the features. This section begins with investigating features that meaningfully correlate with the execution time, which can be categorized into term and query features. Table 5.1 lists 14 term features and 6 query features.

#### 5.3.1 Term Features

Term features capture the “efficiency” of queries by estimating the effects of dynamic pruning. Table 5.1 presents 14 term features, studied to be good predictors [1].

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.
<table>
<thead>
<tr>
<th>Category</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>term feature</td>
<td>AMeanScore</td>
<td>Arithmetic mean scores</td>
</tr>
<tr>
<td></td>
<td>GMeanScore</td>
<td>Geometric mean scores</td>
</tr>
<tr>
<td></td>
<td>HMeanScore</td>
<td>Harmonic mean scores</td>
</tr>
<tr>
<td></td>
<td>MaxScore</td>
<td>Maximum of scores</td>
</tr>
<tr>
<td></td>
<td>EMaxScore</td>
<td>Estimated maximum scores [62]</td>
</tr>
<tr>
<td></td>
<td>VarScore</td>
<td>Variance of scores</td>
</tr>
<tr>
<td></td>
<td>NumPostings</td>
<td># postings</td>
</tr>
<tr>
<td></td>
<td>NumMaxima</td>
<td># local maxima</td>
</tr>
<tr>
<td></td>
<td>GAvgMaxima</td>
<td># maxima greater than average</td>
</tr>
<tr>
<td></td>
<td>MaxNumPostings</td>
<td># postings with maximum score</td>
</tr>
<tr>
<td></td>
<td>In5%Max</td>
<td># postings in 5% of maximum score</td>
</tr>
<tr>
<td></td>
<td>IDF</td>
<td>inverse document frequency</td>
</tr>
<tr>
<td></td>
<td>NumThres</td>
<td># postings in 5% of the kth score</td>
</tr>
<tr>
<td></td>
<td>ProK</td>
<td># docs ever promoted to the top-k</td>
</tr>
<tr>
<td>query feature</td>
<td>English</td>
<td>Query in English or not (binary)</td>
</tr>
<tr>
<td></td>
<td>NumAugTerm</td>
<td># augmented requirements</td>
</tr>
<tr>
<td></td>
<td>Complexity</td>
<td>Degree of query complexity</td>
</tr>
<tr>
<td></td>
<td>RelaxCount</td>
<td>Relax count applied or not (binary)</td>
</tr>
<tr>
<td></td>
<td>NumBefore</td>
<td># terms in the original query</td>
</tr>
<tr>
<td></td>
<td>NumAfter</td>
<td># terms after query rewriting</td>
</tr>
</tbody>
</table>

Table 5.1: Space of the features.
• IDF: the inverse document frequency of the term. This accounts for the number of documents in the corpus that contain the term [63].

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 [1]. In this work, we add the minimum as an aggregation function because the minimum matches the way a conjunctive query is processed.

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 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, adding 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.
5.3.2 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.

In this thesis, we propose to exploit both term and query features, rather than only term features used in the prior work [1], 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 [64]. 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 5.1 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 [65] 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 sub-queries. For example, query “Microsoft Office Windows” can be relaxed to “Microsoft Windows” or “Microsoft Office” [66]. Having it enabled correlates with higher execution time.

5.3.3 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 5.4), (2) the cheap features are practical since they do not require a large memory footprint (Section 5.4), and (3) the cheap features work well for parallelization (Section 5.5).

5.3.4 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
Table 5.2: Top-15 features ranked by the importance obtained from boosted regression tree. Cheap features are shown in bold.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max-IDF</td>
<td>1</td>
</tr>
<tr>
<td>Sum-AMeanScore</td>
<td>0.34823</td>
</tr>
<tr>
<td>Min-MaxScore</td>
<td>0.33350</td>
</tr>
<tr>
<td>Min-MaxNumPostings</td>
<td>0.28197</td>
</tr>
<tr>
<td>Sum-HMeanScore</td>
<td>0.27014</td>
</tr>
<tr>
<td>English</td>
<td>0.26460</td>
</tr>
<tr>
<td>RelaxCount</td>
<td>0.21890</td>
</tr>
<tr>
<td>Min-IDF</td>
<td>0.19128</td>
</tr>
<tr>
<td>Max-GMeanScore</td>
<td>0.18497</td>
</tr>
<tr>
<td>NumAugTerm</td>
<td>0.18442</td>
</tr>
<tr>
<td>Sum-VarScore</td>
<td>0.17083</td>
</tr>
<tr>
<td>Var-HMeanScore</td>
<td>0.16660</td>
</tr>
<tr>
<td>Var-MaxNumPostings</td>
<td>0.16227</td>
</tr>
<tr>
<td>Var-MaxScore</td>
<td>0.15709</td>
</tr>
<tr>
<td>Complexity</td>
<td>0.13965</td>
</tr>
</tbody>
</table>

Table 5.2 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.
Figure 5.1: 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 [1] (the black curve).

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 5.2. Figure 5.1 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 5.2 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, 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 [1], 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 fewer. Since our cheap features contain five out of the top-10 features, we expect that the cheap features should work well.

5.4 Empirical Evaluation of Regression Algorithms

We evaluate the accuracy of three regression algorithms: linear regression, boosted regression tree [67], and Gaussian process regression [68]. 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 5.3 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 5.4. 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
<table>
<thead>
<tr>
<th>Threshold</th>
<th>Metric</th>
<th>Algorithm</th>
<th>term features + query features</th>
<th>cheap features</th>
<th>Prior work [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 ms</td>
<td>Precision</td>
<td>Linear regression</td>
<td>0.6062 ±0.0027</td>
<td>0.6216 ±0.0045</td>
<td>0.6200 ±0.015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gaussian process regression</td>
<td>0.6897 ±0.0051</td>
<td>0.6852 ±0.0098</td>
<td>0.6089 ±0.011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boosted regression tree</td>
<td><strong>0.8306 ±0.013</strong></td>
<td><strong>0.7927 ±0.013</strong></td>
<td>0.6942 ±0.015</td>
</tr>
<tr>
<td>Recall</td>
<td>Linear regression</td>
<td></td>
<td>0.6838 ±0.002</td>
<td>0.2997 ±0.0022</td>
<td>0.4912 ±0.0050</td>
</tr>
<tr>
<td></td>
<td>Gaussian process regression</td>
<td></td>
<td>0.7884 ±0.0071</td>
<td>0.6655 ±0.0282</td>
<td>0.5906 ±0.0203</td>
</tr>
<tr>
<td></td>
<td>Boosted regression tree</td>
<td></td>
<td><strong>0.8007 ±0.0057</strong></td>
<td><strong>0.7723 ±0.0078</strong></td>
<td>0.6280 ±0.0098</td>
</tr>
<tr>
<td>80 ms</td>
<td>Precision</td>
<td>Linear regression</td>
<td>0.6717 ±0.0029</td>
<td>0.7074 ±0.0014</td>
<td>0.6716 ±0.0047</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gaussian process regression</td>
<td>0.8123 ±0.0050</td>
<td>0.7712 ±0.0503</td>
<td>0.7028 ±0.0102</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boosted regression tree</td>
<td><strong>0.8894 ±0.0100</strong></td>
<td><strong>0.8567 ±0.0102</strong></td>
<td>0.7643 ±0.0143</td>
</tr>
<tr>
<td>Recall</td>
<td>Linear regression</td>
<td></td>
<td>0.6627 ±0.0010</td>
<td>0.1929 ±0.0290</td>
<td>0.2817 ±0.0012</td>
</tr>
<tr>
<td></td>
<td>Gaussian process regression</td>
<td></td>
<td>0.7568 ±0.0114</td>
<td>0.4940 ±0.1344</td>
<td>0.5329 ±0.0066</td>
</tr>
<tr>
<td></td>
<td>Boosted regression tree</td>
<td></td>
<td><strong>0.8370 ±0.0084</strong></td>
<td><strong>0.7961 ±0.0063</strong></td>
<td>0.6354 ±0.0153</td>
</tr>
</tbody>
</table>

Table 5.3: 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.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Training time</th>
<th>Prediction overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boosted regression tree</td>
<td>1.805 (sec)</td>
<td>&lt; 0.75 (ms)</td>
</tr>
<tr>
<td>Linear regression</td>
<td>0.006 (sec)</td>
<td>&lt; 0.75 (ms)</td>
</tr>
<tr>
<td>Gaussian process</td>
<td>539.326 (sec)</td>
<td>&lt; 0.75 (ms)</td>
</tr>
</tbody>
</table>

Table 5.4: Training time and prediction overhead comparisons for the three algorithms.

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 5.3. 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 5.5) and has lower training overhead.

Comparison with using only term features. The prior work [1] does not consider query rewriting and query features. Instead, it uses only term features, which are aggregated to be used as inputs to a regression model. See Section 3.3 for further details.

Table 5.3 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 5.5 shows how the proposed solutions are compared to prior work [1] in terms of design considerations and cost.

The first two columns in Table 5.3 implement the proposed solutions that use all features and cheap features, respectively. We compare these to the prior work [1] 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 \((X \text{ or } X')\) and \((Y \text{ or } Y')\). 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 \((X \text{ or } X')\) 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 5.2, our workload requires us to correctly identify at least 75% of the long-running queries \( (i.e., \text{demanding a recall of 0.75 or higher}) \), while the predictor from the prior work \([1]\) has a recall of 0.6485 as shown in Table 5.3. 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-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 \( (i.e., \text{short-running queries misidentified as long-running ones and executed in parallel}) \). (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 5.5. Moreover, Table 5.3 also shows the precision of both our proposed features and cheap features is higher than a 95% confidence interval of \([1]\). 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 5.2 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 5.2(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.
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 accu-
racy 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.

5.5 Evaluation

The boosted tree regression algorithm using cheap features is implemented in the production Bing index server to predict query execution time. Using the predicted query execution time, the index serving node (ISN) is modified to parallelize long-running queries only and run the short-running queries sequentially. Experimental results show that, even under moderate-to-heavy load, compared with parallelizing all queries unselectively, the proposed approach 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 reducing the tail latency, improving server capacity, and to process the same query workload using fewer number of servers. This section describes the experimental setup and results.

5.5.1 Experimental Setup

Machine setup and workload. The ISN used in this 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 and uses 17 GB of memory to cache recently accessed web index data. The web index was created in March 2013. The number of worker threads is
set to 28, considering workers sometimes blocking on I/O accesses. The OS scheduler
assigns the worker threads of a query to the available cores.

Our setup includes an ISN that answers queries and the same client as in Sec-
tion 4.5.1 that plays queries from a production trace of Bing user queries. The trace
contains 100K queries and was obtained in March 2013. Another key change to the
setup in Section 4.5.1 is that the amount of query computational demand is bounded
to 200 ms to be able to sustain higher QPSes.

Policies for comparison. The following policies are compared in the evaluation.

- **Sequential execution** is a baseline system, which does not perform any query
  parallelization.

- **Fixed parallelization** parallelizes each query with 3 threads. Parallelism degree
  3 is selected to produce the 99th-percentile response time of 100 ms.

- **Predictive parallelization** parallelizes only those queries that are predicted to
  have a long execution time using 3 threads and runs the other short-running
  queries sequentially. Our proposed predictor using cheap features and boosted
  regression tree is applied. The predictor from prior work [1] is not used because
  its accuracy does not meet the selective parallelization requirements as discussed
  in Section 5.2. A query is considered as long-running if its predicted execution
time is longer than a given threshold, for example, 80 ms.

Performance metrics. Query response time is the key performance metric for
evaluating the three competing 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 presented. Response quality (i.e., relevance scores) is not reported in the evaluation because the parallelism decision does not degrade the response quality, as explained in Section 4.5.3. No matter what parallelism degree is used, each query always performs at least the same amount of work to sequential execution for computing the relevance scores, thus producing at least the same response quality. The performance metrics are collected after the ISN index cache is warmed to obtain steady state measurements.

The rest of 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.

5.5.2 Prediction Overhead

As prediction itself adds additional work to query execution, it must be lightweight. 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. This prediction time is mostly spent parsing the rewritten keywords.

5.5.3 Response Time Reduction

Figure 5.3 shows both the 99th-percentile and mean response times as well as the CPU utilization for sequential execution, fixed parallelization, and predictive parallelization at different loads. For predictive parallelization, a threshold of 80 ms is used, so the queries predicted to execute longer than 80 ms run with 3-way parallelism. The performance impact of using different threshold values is discussed in Section 5.5.5.
Figure 5.3: Response times and CPU utilization for sequential execution, fixed parallelization, and adaptive parallelization.

Figure 5.3(a) compares the 99th-percentile response time for the three competing policies. The x-axis represents the system load expressed as the average query arrival rate in queries per second (QPS). The figure shows a wide range of load from very low to very high values. The y-axis represents the 99th-percentile response time of the queries. The results lead to 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 reducing the tail latency through query parallelization is more effective if combined with our prediction framework. 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, ISNs hardly ever operate at such very high load, and a simple technique like having a threshold for queue length can solve the issue.

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 5.3(c) presents the average CPU utilization for the three policies under varying load, where one can 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%. (2) True long-running queries, which are rare and show good speedup from parallelization, 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 5.3(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. It once again shows the importance of predicting query execution time.

5.5.4 Capacity Improvement

Figure 5.3(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 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 Bing uses thousands of servers in production, these savings are significant.

5.5.5 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 5.4 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 5.4, at light load from 50 QPS to 450 QPS, all requests are parallelized equivalently using a threshold of 0 ms. With an increase in the load, a larger threshold value is preferred so only the long-running queries are parallelized. As shown in Figure 5.4, from 450 QPS to 750 QPS, the threshold value to optimize the 99th-percentile response time increases to 80 ms. With the further increase of the load from 750 QPS to 800 QPS, the best threshold value becomes 120 ms. At extremely heavy load (>800 QPS), sequential execution, which is equivalent to applying a threshold of infinity, produces the minimum response time.

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
Figure 5.4: Response times with different threshold values.

(a) 99th-percentile response time

(b) Mean response time

A regression model instead of a classification model is used in the learning framework. The regression model offers the flexibility to choose different threshold values.
5.6 Summary

To reduce the tail latency, it is crucial to parallelize mainly long-running queries, and not short-running queries. Short-running queries do not contribute to the tail at all. Moreover, parallelizing short-running queries gives little benefits and consumes resources that could have been used to speed up the processing of long-running queries. To this end, the prediction of query execution time is essential to differentiate queries and to support selective parallelization.

This chapter presented an accurate and efficient predictor for estimating query execution time. To provide better accuracy and efficiency, this chapter mainly proposed to (1) use both query and term features, (2) compile cheap features, and (3) consider query rewriting. As a useful case study for combining the predictor with parallelization, this chapter introduced a parallelization technique, which parallelizes long-running queries and processes short-running queries sequentially based on the predicted query execution time.

The performance of this parallelization approach was evaluated experimentally on Bing search engine as compared to parallelizing all queries. Our results show that under moderate to heavy load, using predictor reduces the 99th-percentile response time by 50% from 200 ms to 100 ms compared with parallelizing 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.
Chapter 6

Predictive Parallelization Framework

6.1 Introduction

The query execution time can be greatly reduced using parallelization. However, using parallelization to reduce the tail latency faces two key challenges: (1) Queries cannot be blindly or equally parallelized because the query workload is not homogeneous and different queries react differently to intra-query parallelization. While there are many short-running queries that do not benefit from parallelization, there are still a significant number of long-running queries that are parallelized well, achieving good speedup. In particular, the parallelization of these latter queries significantly reduces the tail latency. (2) The appropriate degree of parallelism depends on the available spare resources, which vary over time [69].

Predictive parallelization framework exploits both query and system load information to select the degree of parallelism on a per-query basis. Specifically, prediction estimates the query execution time before query processing starts by providing certain features from the query and the inverted index to a trained regression model. Since our workload characterization shows a close correlation between query execution time and parallelism efficiency — longer-running queries achieve better speedup, query parallelism efficiency can be estimated using the predicted execution time. Combining the predicted query execution time and parallelism efficiency information together with the instantaneous system load, predictive parallelization works based
on two online algorithms that decide the degree of query parallelism to reduce the tail latency of web search.

The proposed algorithms selectively parallelize long-running queries and run short-running queries sequentially. The algorithms decide which queries impact the tail and how aggressively to parallelize them. How aggressively we can parallelize a long-running query depends on the system load: Too little parallelism leaves resources idle, where tail latency may be reduced further; too much parallelism overloads a system, prolonging the waiting time of the pending queries in the queue. The algorithms strike a balance by minimizing the summation of the execution time of the current request and the waiting time of the requests in the queue. How aggressively we want to parallelize a long-running query also depends on its impact on the tail and its parallelism efficiency. Without sufficient parallelism, a long-running query may not complete fast enough, increasing the tail latency; with too much parallelism, a query completes much faster than the tail but consumes more resources than necessary, and those resources are better used to tame the latency of other queries. The algorithms therefore choose a degree of parallelism that meets the tail latency target with minimum resources.

Predictive parallelization framework, along with the proposed algorithms, is implemented and evaluated in the context of the Microsoft Bing search engine. When compared to sequential execution of a query, it reduces the 99th-percentile response times by 70% under low-to-moderate load and by 45–70% under moderate-to-high load. Moreover, we develop an efficient fallback mechanism to handle the most important prediction errors. When a long-running query is mispredicted as short-running, its parallelism degree ramps up dynamically when the prediction error is recognized. This effectively reduces the extreme tail, i.e., the 99.9th-percentile response time.
Finally, this chapter evaluates how our algorithms perform on reducing tail latency under different workloads.

**Comparison to other designs.** There are of course other design choices for query parallelization.

*Dynamic parallelization* changes parallelism degree for a query in the middle of computation. For example, a policy may increase the degree over time, starting from sequential execution. As a result, this approach gives no additional cores to parallelizing short-running queries, whereas it allocates more resources to parallelizing long-running queries, thus targeting mainly on tail latency reduction. To prevent the machine from saturation, no further allocation is performed once the number of active threads reaches the number of cores. There are two facts supporting this strategy in the context of web search. First, parallelization is effective only for long-running queries. Second, query execution is computationally intensive, requiring processor resources for better performance, and causes little contention for off-chip bandwidth.

A fundamental limitation of this approach is that it postpones the use of right degree of parallelism by running all queries sequentially in the beginning. In particular, long-running queries do not get enough resources to speed up their execution at the earliest possible time. Although managing ramp-up periods may help, it does not address the problem entirely. Section 7.2 evaluates dynamic parallelization and explains its pros and cons as compared to predictive parallelization.

Fixed parallelization parallelizes each query always with a certain fixed number of threads [8, 9]. This is the simplest technique that makes no use of information in determining the parallelism degree. A query parallelized always with a certain degree achieves its own speedup primarily according to its service demand, resulting in a
stable latency reduction unless the system is saturated.

A downside is that a fixed degree of parallelism cannot reduce response times for all system loads [5]. At light load, there are ample idle cores and some waste due to parallelization overhead is tolerable. Therefore, using a high degree of parallelism pays off under low load, dramatically reducing response latency. However, under high load, where spare processor resources become limited, parallelization adversely delays waiting queries and degrades the system performance.

In contrast, predictive parallelization dynamically selects the degree parallelism on a query-by-query basis. This strategy exploits runtime information, such as system load and query execution time, to promote adaptive scheduling of spare processor resources. For example, adaptation in response to the system load can be such that queries are parallelized inversely proportional to the number of waiting queries. Section 7.1 evaluates the importance of performing such adaptation.

**Organization.** The rest of this chapter is organized as follows. Key factors impacting predictive parallelization design are identified and explained in Section 6.2. Section 6.3 develops two new resource management algorithms employed in our framework. Section 6.4 summarizes this chapter.

### 6.2 Factors Impacting Predictive Parallelization Design

Predictive parallelization makes the parallelism decision before executing a query. This strategy has two key properties: (1) it dynamically selects the degree of parallelism on a query-by-query basis; and (2) parallelism decided for a query does not change. The reason for (1) is to parallelize queries differently based on their execution time and system load. The key insight behind (2) is that as interactive requests often
complete quickly, there is limited time for the scheduler to learn and adapt to the characteristics of an individual request [5].

This section presents three types of important information that an effective predictive parallelism approach should obtain as inputs to make parallelism decision — query execution time, system load, and parallelization efficiency. Section 6.3 explains how predictive parallelism algorithms exploit such information in detail.

6.2.1 Query Execution Time

To reduce the tail latency, it is important to parallelize only long-running queries, and not short-running queries. Short-running queries do not contribute to the tail, and long-running queries benefit the most from parallelization with low overhead. Parallelizing short-running queries gives little benefits and consumes resources that could have been used to parallelize long-running queries. To this end, the prediction of query execution time is essential to differentiate queries and to support selective parallelization.

Prediction estimates query execution time before executing a query to allocate sufficient resources to long-running queries at the earliest possible time. In the web search server, execution of requests shares a lot of similarities because user queries are processed using the same procedure. Therefore, the prediction here is more feasible and effective than in common multiuser multiprogrammed environments, in which schedulers know little about the requests.

Our prediction framework incorporates more effective features, query rewriting, and boosted-tree regressor, resulting better accuracy [61]. Across 44 server nodes we tested, (precision, recall) is stable and high with (0.88, 0.85), (0.91, 0.86), and (0.89, 0.84) on average for identifying queries running longer than 60, 90, and 120 ms,
respectively. Moreover, the space overhead to cache the prediction features is small, around 38 MB per server for ten million keywords, and the running time of the prediction is about 0.6 ms, which are about 4% of the average query execution time.

Prediction accuracy, which is measured by recall and precision, impacts the tail latency. High recall means we identify the majority of long-running queries. We can reduce their response time using parallelization, and thus the tail latency. Recall that our predictor has a recall of 0.75 or higher and thus effectively reduces the 99th-percentile response time in our workload. Precision is directly related to how many short-running queries are mispredicted as long-running. This misprediction results in an overhead. This overhead is small in our case.

6.2.2 System Load

Instantaneous load on search servers varies over time [69], impacting the availability of idle resources. This motivates the need for adaptation of parallelism to prevent overloading the system and to exploit the available idle resources. To illustrate, consider two extreme cases: First, at low load with only one query in the system, it is better to use as many cores as possible to process the query, exploiting all idle cores for a shorter response time. Second, at heavy load, a high degree of parallelism introduces substantial overhead. Here resources are scarce and the parallelization overhead takes resources away from waiting requests.

A variety of metrics can be used for system load, including query arrival rate, waiting queue length, number of active threads, and processor utilization. Because hundreds of interactive requests come in and out within a second, it is imperative to model the instantaneous system load. Experimental evaluation shows that both waiting queue length and the number of active threads are good metrics capturing fine-
grained information for system load — even at the same average query arrival rate, the change of queue length due to transient overload or underload is observed. Processor utilization is a very intuitive metric and can be obtained easily from OS-provided performance counters. However, it has a limitation on accurately representing the current load because it is a moving average heavily weighted from the past. OS accounts processor utilization for each tick to reduce overhead. For example, Windows sets 10 ms or larger for the tick. This is in fact too coarse-grained considering the lifetime of search queries, which is typically around tens of milliseconds. Section 7.2 compares these metrics.

6.2.3 Parallelization Efficiency

A good degree of parallelism depends on the parallelization efficiency. To illustrate this relation, consider the case of perfect parallelization efficiency, in which the response time decreases linearly with increased parallelism. Here one should use as many cores as possible. In contrast, if parallelization efficiency is worst, response time does not decrease with increased parallelism, making sequential execution the best alternative. Realistic requests fit between these extremes, and better efficiency allows higher parallelism degree.

Our framework uses a request parallel execution time profile to model request parallelization efficiency. The request parallel execution time profile maps the degree of parallelism to an execution time. We denote it by \( \{T_i | i = 1...P\} \), where \( T_i \) is the request execution time with parallelism degree \( i \) and \( P \) is the maximum degree. Depending on how much we know about the request at runtime, \( T_i \) represents values with different accuracy. If we have exact information on a query \( q \), \( T_i \) represents \( T_i^q \), the execution time of request \( q \) using \( i \) threads. In practice, such parallel execution
time profile is hard to predict. In our framework, queries can be classified into groups based on their sequential execution time, and long-running queries exhibit better speedup than the short-running ones (Figure 4.4 in Section 4.5.2). Since sequential execution time of a query can be predicted, we can use it to find the right group of speed up profiles for the query.

When queries are not differentiated based on their execution time, $T_i$ represents $T^\text{AVE}_i$, which is the average execution time of all jobs using $i$ threads.

### 6.3 Heuristic Algorithms

This section introduces two heuristic algorithms for predictive parallelism, $AP$-$Pred$ and $Timeline$, which exploit per-query information for better resource allocation and adapt parallelism decisions to varying system load. They are specifically designed to reduce the tail latency. $AP$-$Pred$ exploits the characteristics of the search server workload. $Timeline$ is a more general solution to reduce tail latency of various workloads.

#### 6.3.1 AP-$Pred$

We initially designed Adaptive Parallelism ($AP$) approach to take into account the average parallelism speedup of all queries as well as the waiting queue length [5]. $AP$-$Pred$ extends $AP$ by accounting for parallelism efficiency. This section first describes the key idea of $AP$ as a background, and then presents $AP$-$Pred$.

**Background.** $AP$ considers waiting queue length and average parallelism speedup of requests to decide request parallelism degree [5]. Suppose $T^\text{AVE}_i$ is the average request execution time with parallelism degree $i$, $K$ is the waiting queue length, and
$P$ is the number of cores. There are two parts we consider for AP: (1) \textit{self-increase}, $T_{i}^{AVE}$, which is the estimate of the execution time of the request being parallelized, and (2) \textit{peer-increase}, $T_{i}^{AVE} \cdot i \cdot K$, which estimates the additional time that other requests in the system have to wait because of the execution of the query being parallelized.

The following examples illustrate how these two parts work in response to system loads. First, at very light load with only query $Q$ in the system, the self-increase of $Q$ dominates and the peer-increase becomes zero, so small $T_{i}$ is better. As larger parallelism degree often reduces execution time, AP chooses a high parallelism degree at light load. Second, at heavy load, the peer-increase dominates due to large $K$. We need to minimize the value of $T_{i} \cdot i$, which is the total amount of work query $Q$ incurs. Sequential execution often gives the minimum because of parallelization overhead. Therefore, AP chooses a low parallelism degree at heavy load.

To illustrate the impact of parallelization efficiency, consider two extreme cases. (1) For a query with perfect linear speedup, the values of $T_{i} \cdot i$ are the same for any parallelism degree $i \leq P$. In this case, the peer-increase does not change with $i$ value, thus AP chooses the $i$ value minimizing the self-increase, \textit{i.e.}, a fully parallelized execution with $i = P$. (2) For a query without any speedup, the values of the execution time $T_{i}$ are the same for any $i \leq P$. In this case, self-increase does not change with $i$ value, thus AP chooses the $i$ value minimizing the peer-increase, \textit{i.e.}, a sequential execution with $i = 1$.

AP has a measure defined to estimate impacts related to response times by the query being parallelized. AP is designed to operate in a greedy fashion by minimizing the impacts per query. Before each query execution, AP decides query parallelism
degree $d$ by summing up the self-increase and peer-increase as follows:

$$d = \arg \min_{i=1}^{P} \min_{i=1}^{P} (T_{i}^{AVE} + \frac{T_{i}^{AVE} \cdot i}{P} \cdot K).$$

(6.1)

The arg min operator searches for the parallelism degree from 1 to $P$ that minimizes the estimated increase on the total response time and returns that parallelism degree. This formula is a heuristic that has been shown to work well in practice, as will be presented in experimental results in Section 7.

More generally, when a request has high parallelization efficiency and low overhead, AP is more aggressive at using higher degrees of parallelism: higher degrees reduce the self-increase significantly while increasing the peer-increase slightly. On the other hand, when a request has low parallelization efficiency, its decision is more conservative: higher degrees significantly increase the peer-increase as the total work of the parallel execution is much larger than sequential.

**AP-Pred algorithm.** AP-Pred extends AP by using predictive per-query information to refine query execution time profile. Instead of using the average execution time profile $\{T_{i}^{AVE}\}$ from all queries (Equation 6.1), which does not differentiate parallelism efficiency among queries, our workload characterization indicates a better approach. Queries can be classified into groups based on their sequential execution time, and long-running queries exhibit better speedup than the short-running ones (Figure 4.4 in Section 4.5.2). Since sequential execution time of a query can be predicted, AP-Pred uses it to find the right group of speedup profiles for the query. For each group $c$ of queries, we measure their speedup profiles $\{S_{i}^{c}\}$, where $S_{i}^{c}$, the speedup with parallelism $i$ is computed as the average speedup of all queries within the group. AP-Pred uses the group speedup profiles and the predicted query sequential execution time to estimate query’s parallel execution time. For a query $q$, its
execution time with parallelism $i$ is $T_i^q = T_1^q / S_{c_i}^c(q)$, where $c(q)$ is a function that maps the query to its group of queries sharing similar speedup profiles.

Combining the refined parallelism efficiency with Equation 6.1 of AP, AP-Pred chooses the parallelism degree $d$ as follows:

$$d = \arg \min_{i=1,...,P} \left( \frac{T_i^q}{S_i^c} + \frac{T_1^q}{S_1^c} \cdot \frac{i}{P} \cdot K \right),$$

where $T_i^q$ is the predicted execution time of query $q$, and $S_i^c$ is the measured speedup profile for the group $c$ that query $q$ belongs to.

**Merits.** AP-Pred differentiates queries with different parallelism efficiency: it grants higher parallelism to queries with larger speedup and higher efficiency while limits the parallelism degree of queries with smaller speedup and lower efficiency. This can be seen from Equation 6.3. When a query $q$ exhibits higher speedup, given a larger parallelism degree $i$, $S_i^q$ is larger and the corresponding increase on total response time is smaller. Therefore, under the same load, AP-Pred chooses a larger parallelism degree for the query. By allocating resources to queries with higher parallelism efficiency, AP-Pred reduces waste, improves system efficiency and thus reduces total and average response time.

**Tail versus average latency.** Although AP-Pred seems to optimize for reduced average latency, our experimental results show that AP-Pred significantly reduces the tail latency of web search workload because of the correlation of execution time and speedup profiles that the workload exhibits. Recall that short-running queries exhibit almost no speedup with parallelism, whereas high parallel efficiency is obtained for long-running queries. While AP-Pred favors queries with higher efficiency, for
the search engine workload, they are also long-running queries; while short-running
queries, due to their poor parallelism efficiency, are hardly parallelized, effectively
reducing tail latency.

6.3.2 Timeline

The study of AP-Pred motivated us to develop an algorithm, which does not rely
on the correlation between execution time and parallelism efficiency and can effec-
tively reduce tail latency of other workloads. To achieve that, we propose Timeline
approach. Timeline allocates available resources to long-running requests which im-
pact the tail. The challenge is to decide which queries impact the tail and by how
much. Timeline addresses the challenge in two steps: (1) Decide a target comple-
tion time $E$ for all requests based on system load and parallelization efficiency. All
requests try to complete within the target, and the requests with predicted sequen-
tial execution time longer than $E$ can potentially violate the target, which shall be
parallelized. (2) Decide the request parallelism degree to meet $E$ by using minimum
resource: short-running queries can complete within $E$ using sequential execution and
only long-running queries are parallelized. Putting extra resources for a request to
complete much earlier than $E$ is not beneficial to the tail while other requests are
trying to meet $E$. Instead, the extra resources may be better used by other requests,
reducing their waiting and/or execution time. We start with step (2) as it is more
intuitive.

Using target completion time $E$ to decide parallelism degree. This step
finds the request parallelism degree to meet $E$ by using minimum resources, which
translates to finding the minimum parallelism degree to complete the query within
for our workload. In particular, once the ISN predicts the execution time $L_q$ of a request $q$, we first get the execution time profile of the query by using the profile of the query group $c(L)$ as that in AP-Pred. The profile shows speedup $S_i$ for the request with parallelism degree $i$. Consequently, the remaining work is to find the smallest parallelism degree $d$ such that $d$ gives request completion time less than $E$:

$$d = \min_{i=1 \ldots P} \{ L/S_i \leq E \}. \quad (6.4)$$

Using minimum resources to meet $E$ saves resources from short-running queries and use them to parallelize long-running queries as they are more important for tail reduction. In the remainder of this section, we discuss why the selection of $E$ is important, how its value affects tail latency and how we decide it.

**Impact of $E$ on tail latency.** To demonstrate the impact of $E$, we consider two extreme cases. (1) $E$ is very large, i.e., larger than the maximum sequential completion time of all queries. In this case, all queries execute sequentially. System does not benefit from parallelization, and the tail latency is long. (2) $E$ is very small, i.e., all queries are executed using maximum parallelism degree. Even under a moderate query arrival rate, the system will be overloaded with large parallelism overhead, causing long-running query waiting time, high average and tail latency. Therefore, when $E$ is too small or too large, tail latency suffers. In particular, with too small $E$, high tail latency is dominated by long waiting time, while with too large $E$, high tail latency is dominated by long execution time.

**Factors affecting the selection of $E$.** Two factors affect the selection of $E$: system load and parallelism efficiency. When load increases, we shall choose a larger $E$ because the total spare resources in the system decreases; we shall parallelize queries
less aggressively and use spare resources for really long-running queries only.

When parallelization is more efficient with better speed up and less overhead, we shall choose a smaller E because, under the same load, the same amount of total spare resources can afford to parallelize more queries when overhead is lower, supporting smaller E and parallelizing queries more aggressively.

**Deciding $E$ empirically.** The appropriate value of $E$ is related to both system load and parallelization efficiency of queries. Ideally, we want to produce a table that gives an appropriate $E$ value for each pair of system load and parallelization efficiency. However, in practice, parallelization efficiency of queries is fairly stable. They only change with deployment of new parallelization techniques or major change of query distributions. Therefore, we build a table that relates $E$ with varying loads. For example, one can consider query arrival rate as load metric in the range of 0 - 900 QPS with an increment of 50QPS per entry. Then, for each load (e.g. 0, 50, 100, ...,900QPS), we try different $E$ values experimentally by running a test set of queries offline and find the best $E$ that leads to the smallest desired percentile latency. The exploration of $E$ is done using binary search.

This experimental approach is simple and effective. Any fine-grained load indicator presented in Section 6.2.2 can be used as load metric in the table to capture temporal load fluctuation. At runtime, we decide parallelism degree at per query basis. Each query uses the table to look up the appropriate $E$ value given the load at query arrival. The parallelism degree of the query is decided using $E$, its predicted execution time, and its speedup profile as shown in Equation 6.4.

**Deciding $E$ analytically.** Intuitively, we may compute the optimal value of $E$ numerically using a queueing model [70] and a binary search. For a given load and par-
allelism efficiency, given a value of $E$, use queueing model to compute the tail latency, i.e., 99-percentile latency. Then, use binary search to find $E$ value that minimizes the 99-percentile latency. In the queueing model, both system load and parallelism efficiency are modeled. System load is represented by query arrival rate and arrival distribution, while parallelism efficiency affects service demand distribution. With both arrival and service distribution, we may apply an appropriate queueing system model to compute the tail latency.

Applying the above analytical model, however, has some challenges and limitations. (1) The well-known queueing models consider sequential requests only. Little prior work on queueing theory addresses parallel requests, such as split-merge [71] and fork-join queueing model [72]. However, they require all requests having a fixed parallelism degree. (2) Queueing model quantifies system behavior on steady states and uses QPS (queries per second) as load indicator. It does not capture instantaneous variation of system load and job mixes. The experimental approach, however, can use more fine-grained load indicators to provide more prompt indication of load variations. It allows the algorithm to exploit instantaneous spare resources and prevent transient overloading.

**Summary.** The Timeline approach coordinates job completion time using a common target, which is used to identify long-running queries that are likely to impact the tail and thus should be parallelized. The key idea is to allow a request to run fast enough without prolonging the tail while using minimum resources — additional resources can be used to complete other jobs within the target. The key challenge is to decide the target completion time $E$, where we identify key factors affecting $E$ and propose an experimental approach to determine its value. This approach effectively
reduces tail latency without relying on the execution time and parallelism efficiency correlation as AP-Pred does.

6.4 Summary

This chapter developed the factors that impact how query processing should be parallelized. This chapter presented three factors available in a search server that represent both query and system load information. If the goal is to lower tail latency, it is essential to parallelize long-running queries and reduce their latencies. The accurate predictor for query execution time plays a key role in identifying these queries at the earliest possible time even before a query is processed. Moreover, their speedup efficiency is estimated through system calibration, and better efficiency allows higher parallelism degree. Finally, the spare resources available to parallelize queries vary over time, and they are captured as instantaneous load.

This chapter presented two novel resource management algorithms, AP-Pred and Timeline, specifically designed to reduce the tail latency. They exploit per-query information for better resource allocation and adapt parallelism decisions to varying system load. AP-Pred exploits the characteristics of the search server workload. On the other hand, Timeline is a more general solution to reduce tail latencies of various workloads.

The next chapter compares the proposed algorithms to existing work experimentally. It shows that no existing work achieves similar gains as they do not exploit both query and load information.
Chapter 7

Evaluation

While the primary goal of this chapter is to show the superiority of AP-Pred and Timeline, this chapter evaluates various points in the parallelization design space as described by Table 7.1. There are largely three dimensions for the design space. First, parallelism change illustrates whether the desired degree of parallelism for a request is reached in the middle of execution or before execution. Second, information use describes three factors presented in Section 6.2 that can be used collectively for the parallelism decision. Third, target specifies the performance goal, for which a parallelization strategy exploits information in the first two dimensions. Note that the same information can be used by several strategies differently — such examples are AP-Pred and Timeline.

Most of the strategies in Table 7.1 were already explained. RampUp increases the degree over time at each predefined interval, starting from sequential execution (Section 6.1), Fixed is fixed parallelism that parallelizes each query always with a certain fixed number of threads (Section 6.1), and Pred predicts query execution time to parallelize long-running queries using a fixed degree of parallelism and runs remaining queries sequentially (Section 5.5). AP is Adaptive Parallelism that appears in Section 6.3. Those policies that were not explained are summarized as follows.

- **Linear.** Raman et al. proposed the Work Queue Linear (Linear) that considers only instantaneous system load, which is the number of queries waiting in the
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Parallelism change</th>
<th>Information use</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Exec. time</td>
<td>System load</td>
</tr>
<tr>
<td>RampUp</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Fixed</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Linear</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Binary</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>AP</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pred</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>AP-Pred</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Timeline</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 7.1: Decomposition of all the query parallelism strategies according to parallelism change and information use, and their performance targets. Note that Pred, AP-Pred, and Timeline use prediction to estimate query execution time.
queue [2]. For the system load below a threshold, queries are parallelized with a degree inversely proportional to the system load.

- **Binary.** Raman et al. proposed the Work Queue Threshold (Binary) that also considers the same instantaneous system load only [2]. It selects maximum parallelism when staying the system load below a threshold for a while, and minimum degree (sequential execution) otherwise.

One may wonder if RampUp can be used combined with other strategies and what the impact is. In fact, RampUp can be used combined with AP-Pred or Timeline for reducing the far tail of mispredicted queries. Section 7.2.4 evaluates this approach.

All the policies are implemented in the index serving nodes (ISNs) of Microsoft Bing and evaluated experimentally using production workloads. Experimental results are presented in two steps. First, AP is comprehensively compared to Fixed, Linear, and Binary to show the importance of considering both system load and parallelization efficiency (Section 7.1). Then, strategies exploiting either predicted query execution time or parallelism change are evaluated in order to contrast predictive approaches to others (Section 7.2). Note that compared to Section 7.2, because the experimental setup in Section 7.1 has no timeout set to the ISN and has only early termination factored in, it operates in much lower query arrival rates.

7.1 The Importance of System Load and Parallelization Efficiency

The following is key performance metrics in the evaluation:

- **Response time.** The query latency 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 (Mean) and 95th-percentile response time (95th-Percentile) are presented. Our measurements show that the improvement in 99th-percentile response time is similar to 95th-Percentile.

- Cost of parallelism. Average CPU utilization of the ISN and the I/O and network bandwidth overheads are reported.

7.1.1 Response Time

Figure 7.1 shows the Mean and 95th-Percentile response time for all competing policies. For fixed parallelism, we evaluate 5 different degrees of parallelism (from Degree-2 to Degree-6). We limit parallelism degree to 6 (number of cores on 1 CPU) because higher degrees do not perform better than degree 6 — we have already observed diminishing returns from degree 5 to 6 in Figure 4.4. Note that sequential execution (Sequential) is based on unmodified production code.

Figure 7.1 shows comparisons in two dimensions. First, it presents the performance of fixed parallelism over a variety of query arrival rates. The figure clearly shows that there is no fixed degree of parallelism that performs best across all the arrival rates. Second, the figure shows that our adaptive algorithm (Adaptive) performs the same or better than all other policies under all the arrival rates. In particular, for the arrival rates less than 90 QPS, adaptive parallelism outperforms all fixed parallelism solutions. For high arrival rate, 90 QPS and above, adaptive parallelism does not achieve any benefits, but it also does not incur any penalty.

As the figure shows, our adaptive approach dynamically selects the best degree of parallelism given the current arrival rate. As the arrival rate increases, the degree of parallelism for each query is reduced appropriately until the system is saturated at 90 QPS. This allows the adaptive strategy to achieve lower mean and 95th-percentile
Figure 7.1: Response time of sequential execution, fixed parallelism, and adaptive parallelism. The adaptive parallelism performs better than any of fixed parallelism configurations. It also achieves significant reductions in both mean and 95th-percentile latencies than sequential execution.
latencies under all arrival rates. For example, at 60 QPS, sequential execution has mean and 95th-percentile latencies of 86 ms and 316 ms, respectively. The best fixed parallelization strategy (Degree-3) lowers the mean and 95th-percentile latencies to 55 ms and 186 ms, whereas adaptive parallelism further reduces these latencies to 49 ms and 160 ms.

Note that Adaptive even outperforms the best fixed strategy at each arrival rate. That is because even then, there are queries being satisfied with different degrees of parallelism, more effectively utilizing the system. Figure 7.2 shows the distribution of the selected degree in the adaptive strategy at 60 QPS. The figure illustrates that 1) unlike fixed parallelism, Adaptive is able to select any degree among all possible options, and 2) these degrees are utilized unevenly to produce better performance. Adaptive parallelizes queries using a degree of 3 or 4 in most cases, with an average degree of 3.44. This behavior cannot be achieved by any fixed parallelism degree, and
Table 7.2: Comparison of CPU utilization. A policy & QPS with a **bold** value indicates worse response times than Sequential.

<table>
<thead>
<tr>
<th>Policy</th>
<th>10 QPS</th>
<th>30 QPS</th>
<th>50 QPS</th>
<th>70 QPS</th>
<th>90 QPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>7%</td>
<td>22%</td>
<td>33%</td>
<td>50%</td>
<td>63%</td>
</tr>
<tr>
<td>Degree-2</td>
<td>8%</td>
<td>23%</td>
<td>38%</td>
<td>57%</td>
<td>77%</td>
</tr>
<tr>
<td>Degree-3</td>
<td>9%</td>
<td>26%</td>
<td>44%</td>
<td>69%</td>
<td>87%</td>
</tr>
<tr>
<td>Degree-4</td>
<td>9%</td>
<td>29%</td>
<td>50%</td>
<td><strong>76%</strong></td>
<td>95%</td>
</tr>
<tr>
<td>Degree-5</td>
<td>9%</td>
<td>30%</td>
<td>55%</td>
<td><strong>85%</strong></td>
<td>≈100%</td>
</tr>
<tr>
<td>Degree-6</td>
<td>10%</td>
<td>36%</td>
<td>62%</td>
<td><strong>91%</strong></td>
<td>≈100%</td>
</tr>
<tr>
<td>Adaptive</td>
<td>10%</td>
<td>31%</td>
<td>51%</td>
<td>65%</td>
<td>62%</td>
</tr>
</tbody>
</table>

7.1.2 Cost of Parallelism.

Table 7.2 compares the CPU utilization of the different policies. CPU utilization is periodically sampled using the performance counters in Windows. The fixed parallelization strategies incur increasing CPU utilization as more threads per query are used. This is expected, due to increasing parallelization overheads and additional speculative work. On the test system, response times increase at high rate when CPU utilization goes above 70%. The significant CPU contention that results under high arrival rates for these fixed parallelization strategies therefore increases both queue waiting time and query response time.

While the adaptive strategy (the last row) also consumes additional CPU cycles when it parallelizes queries, CPU utilization is always below 70% across a wide range
Figure 7.3: Mean response time with changing arrival rates between adaptive parallelism and sequential execution. Moving average of 30 sec is used to compute mean.

of arrival rates. Therefore, the adaptive strategy is able to balance CPU utilization and query response time effectively.

The adaptive strategy also incurs minimal overheads in terms of I/O and network bandwidth. While using the adaptive strategy, I/O bandwidth increases from 9 MB/s to 14 MB/s at 60 QPS. The increased I/O bandwidth is due to the increase in speculation placing additional pressure on the cache of the index. This increase, however, is marginal considering the bandwidth supported by SSDs and thus is not a limiting factor in the ISN. There is no observable network overhead; this is expected, as the network is primarily used to simply transfer the top matching results.

7.1.3 Dynamic Changes in Query Arrival Rates

We so far have presented results when the arrival rate is fixed. Here, we vary the arrival rate over time to show that the adaptive strategy can react to such changes. For this experiment, we begin with a query arrival rate of 30 QPS, increase it to
60 QPS, and then reduce it back to 30 QPS. Figure 7.3 shows the moving average of the response time against time in this experiment.

Figure 7.3 shows that the adaptive strategy is responsive to runtime query arrival rate changes with response times consistent with what we observed in Figure 5.3. Specifically, at 30 QPS, the adaptive strategy executes queries mainly using 5 or 6 threads per query to aggressively make use of idle cores. When the arrival rate goes up to 60 QPS, however, it more conservatively uses 3 or 4 threads per query. As a result, the response times observed under these arrival rates are consistent to those in Figure 5.3. The sequential response times remain relatively constant because the machine is not saturated at these arrival rates.

7.1.4 Comparison to Other Algorithms

*Linear* and *Binary* [2] are other possible adaptive parallelization strategies, but they consider only instantaneous system load, which is the number of queries waiting in the queue.

The binary scheme selects either no parallelism (sequential execution) or maximum parallelism (degree 6). It begins with sequential execution. If the system load remains below a threshold $T$ for $N$ consecutive queries, it starts parallelizing queries with degree 6. If the system load then remains above the threshold $T$ for $N$ consecutive queries, it returns to sequential execution. In all experiments, we set $N$ to 50 and $T$ to be the average load observed in Sequential at 50 QPS.

The linear scheme utilizes all degrees of parallelization between 1 and 6. If the system load goes above some maximum, $T$, then queries are not parallelized. Below $T$, queries are parallelized inversely proportional to the system load. The degree of parallelism is computed using the formula $max(1, 6 \times (T - K) \div T)$, where $K$ is
Figure 7.4: Response time comparison with other schemes, *Binary* and *Linear*, from prior work [2]. Adaptive performs better than both of them.

the instantaneous system load. The result is rounded to yield an integral degree of parallelism. Small values of $T$ result in a more conservative approach where lower
degrees of parallelism are used for even moderate system loads. Higher values of $T$ result in a more aggressive approach that selects higher degrees of parallelism.

Figure 7.4 compares these schemes with various values of $T$. For comparison, the results of Sequential (sequential execution), Degree-6 (the highest degree of fixed parallelism), and Adaptive (adaptive algorithm) are repeated in the figure.

Given the limited choices available to the binary strategy, one would expect it to vary in performance between sequential and Degree-6. As the figure shows, this is exactly what happens. As the threshold $T$ is reached (around 50 QPS given the parameters we used), it transitions from being close to the Degree-6 performance to the sequential performance. While the transition point can be tuned, this strategy fundamentally follows these two curves.

The linear strategy provides more freedom to utilize the system and balance performance. If the value of $T$ is set to be too low (i.e., $T = 1$), then the strategy will be too conservative and will largely track sequential performance. As Figure 7.4 shows, however, Linear-1 is still able to achieve gains over Sequential at low arrival rates. If the value of $T$ is set to be too high (i.e., $T = 20$), then the strategy will be too aggressive and will blow up. As the figure shows, it still outperforms a fixed degree of 6, but has simply moved the saturation point where response times increase dramatically from around 50 QPS to 70 QPS.

With moderate values of $T$, such as 5 or 10, performance is more balanced. Figure 7.4 shows that Linear-5 and Linear-10 perform well across the spectrum of arrival rates. However, even Linear-10 is too aggressive under high arrival rate, which results in significantly increased response times over sequential for 100 QPS. Our application has non-negligible parallelization overheads due to a large number of short-running queries. Consequently, the more conservative linear approach ($T = 5$) is better than
linear with other $T$ parameters.

Adaptive performs better than linear because Adaptive exploits both parallelization efficiency and system load when selecting the degree of parallelism. For example, in 95th-percentile latencies, Adaptive performs 8.5%, 17.5%, 11.5%, and 14.6% better than Linear-5 under 50, 60, 70, and 80 QPS, respectively. The performance gap is wider for higher-percentile values; in 99th-percentile latencies, Adaptive performs 21%, 27.5%, 27.2%, and 23% better than Linear-5 under the same arrival rates.

The parallelization efficiency is a good indicator on how aggressively the parallelism degree should decrease with increased load. In contrast, linear uses only system load, and therefore it may be either too conservative or too aggressive. Also, as its performance is sensitive to the selection of the threshold value, if workload changes, one should make it clear how to change the threshold value, which is not a robust way.

**Summary.** Adaptive parallelization strategy dynamically selects the degree of parallelism on a query-by-query basis. The selection is made based upon the current load and the estimated cost of parallelizing the query. The cost amounts to a combination of the estimated benefits to the parallelized query and the estimated delays on subsequent waiting queries. The adaptive strategy outperforms the baseline sequential execution that has been performed on production servers. The delay is decreased or at least the same.

When system load is light, using fixed degrees of parallelism for all queries results in latency reductions of up to 50%, when compared to sequential execution. However, as load increases, the system becomes saturated and latency increases rapidly. This leads to both the need for parallelism to reduce latency and the need for adaptation to
prevent system saturation, depending on the system load. However, existing adaptive
techniques [2] decide the request parallelism degree using only system load without
considering request parallelization efficiency. This results in improvements over using
a fixed degree of parallelism. However, it is hard to decide how to decrease the degree
of parallelism with increased load without considering the efficiency with which an
individual request can be parallelized. Either the scheme will be too conservative
and not reduce latency as much as possible, or the scheme will be too aggressive and
latency will increase beyond sequential at higher loads. Our adaptive parallelization
strategy achieves the best of both worlds.

7.2 The Power of Predictive Parallelization Strategies

This section evaluates various points in predictive parallelization strategies. Specifically, five sets of experiments are conducted:

1. Compare AP-Pred with adaptive parallelism (AP) and Pred to show the importance of using both query and system load information.

2. Compare AP-Pred and Timeline to illustrate how they perform over different workloads.

3. Evaluate an alternative way (RampUp) to identify the effectiveness of speeding up long-running requests early enough using prediction.

4. Investigate how to reduce extremely high tail latency.

5. Study sensitivity on load indicators and efficiency grouping.
Policies for comparison. Our evaluation compares four parallelization algorithms in Table 7.1: AP, Pred, AP-Pred, and Timeline. The evaluation includes sequential execution (Sequential) and fixed parallelization (Fixed) for reference. Implementation of the algorithms follows their detailed description in Section 5.5 and Chapter 6 and 7. Two default algorithm configurations for the experiments are specified here:

1. To model parallelism efficiency at AP-Pred and Timeline, queries are classified into three groups, Short (<30 ms), Mid (30–80 ms), and Long (>80 ms) queries, which are identical to Figure 4.4. Other groupings are discussed in the sensitivity studies in Section 7.2.5.

2. Timeline uses the number of active threads used by long-running queries to model system load. Several other load metrics are compared quantitatively in Section 7.2.5.

The grouping is directed from our workload characterization. Since short-running queries are predominant and their parallelization speedup is close to no speedup, they are one group. Queries running longer than 80 ms exhibit high parallelization efficiency, so they become another group — the threshold of 80 ms with our predictor guarantees the 99th-percentile latency reduction, as explained in Section 5.2.

Performance metrics. To compare tail latency among policies, the 99th-percentile of query response time is presented. Section 7.2.4 discusses an extremely high-percentile response time, using 99.9th-percentile.

As parallelization consumes additional CPU cycles, CPU utilization of the ISN is also an important performance metric. It is not reported here, as Section 5.5.3 already discussed that using prediction for long-running query parallelization incurs a marginal increase in CPU utilization.
7.2.1 Power of Information

This experiment compares AP-Pred with AP and Pred to show the importance of using both query and system load information. Figure 7.5 presents the 99th-percentile response time and the CPU utilization of the policies over different loads. The load varies in a wide range from very low to very high values. Pred is configured with 80 ms for threshold and degree 3 as designed in [61], so the queries predicted to execute longer than 80 ms run with 3-way parallelism. For reference, Figure 7.5 also plots the results of using 3-degree fixed parallelism (Fixed) and sequential execution (Sequential).

**Importance of using query execution time.** Figure 7.5 shows that using per-query information, both Pred and AP-Pred significantly reduce tail latency even at moderate and heavy load. For example, in the range of 500–700 QPS, Pred and AP-Pred achieve around 100 ms 99th-percentile, while all other approaches treat-
ing short- and long-running queries equally have 200 ms latency or above. This is because predicted query information allows Pred and AP-Pred to parallelize long-running queries selectively, avoid parallelizing short-running queries, and dedicate more resources to long-running queries. In comparison, AP does not differentiate long-running and short-running queries, which have different parallelization efficiency and different impact to tail latency. Thus, as load increases, it reduces parallelism degree for all queries: short-running queries may still get parallelized unnecessarily, wasting resources, while long-running queries are not parallelized with a sufficiently large degree, causing high tail latency.

**Importance of considering system load.** When Pred and AP-Pred are compared, they have similar tail latency under heavy load, but AP-Pred reduces tail latency further from 100 ms of Pred to 60 ms under low to moderate load. The reason is that Pred is not adaptive to varying system load and it always parallelizes the same set of long-running queries with the same parallelism degree. In comparison, AP-Pred decides whom to parallelize and by how much according to system load and exploits the available resources to the fullest. Specifically, up to 400 QPS, AP-Pred aggressively parallelize both the medium-running and the long-running queries with high degrees, 5 or 6. At 400–700 QPS, AP-Pred parallelizes mainly the long-running queries with high degrees. This clearly shows that predictive parallelism is more effective if combined with adaptation to system load.

In summary, the comparison in this section shows the importance and effectiveness of factoring in both per-query information and system load information to (1) favorably parallelize long-running queries on reducing tail latency, and (2) adapt parallelism degrees at runtime. Ignoring either type of information will result in
suboptimal performance.

7.2.2 AP-Pred and Timeline

This section compares AP-Pred and Timeline, discussing when and why they perform similarly/differently over different query workloads.

Search engine workload. Figure 7.6(a) compares the performance between AP-Pred and Timeline on our search engine workload. Two algorithms perform similarly, both achieving good reduction on tail latency.

Synthetic workload. As discussed in Section 6.3, AP-Pred depends on correlation between execution time and parallelism efficiency, whereas Timeline does not. So, both strategies are evaluated using a synthetic workload to show how they perform for a workload with less correlation. In the synthetic workload, long-running queries have smaller speedups compared with the search engine workload. In particular, we inject additional sequential computation is injected to long-running queries (>80 ms) of our original workload, so that they exhibit the same efficiency to mid-running queries (30–80 ms). This synthetic workload is designed considering a workload where there is a bottleneck in either synchronization or memory bandwidth [24]. In such case, there is no (or decreasing) speedup if parallelism degree of a request is over a certain value.

Figure 7.6(b) shows the results of using the synthetic workload. The figure shows that Timeline significantly outperforms AP-Pred across loads. If query execution time and parallelism speedup are less correlated, AP-Pred is less effective on reducing the tail latency. AP-Pred gives higher parallelism to queries with higher parallelism efficiency. At search engine workload, long-running queries have high parallelism effi-
Figure 7.6: AP-Pred and Timeline with two workloads where long-running queries have distinct parallelization speedups.

In (a) Long-running queries with high efficiency (original) and (b) Long-running queries with low efficiency (synthetic), AP-Pred and Timeline graphs illustrate how the efficiency and parallelization degrees are assigned to mid-running and long-running queries.

At synthetic workload, AP-Pred considers mid-running and long-running queries with the same parallelism efficiency, so it assigns them the same parallelism degree under a given load. However, long-running queries more critically contribute to tail latency and they should...
be parallelized with higher degrees. In contrast, because Timeline is directed by target completion times, all queries running longer than a target are all parallelized and coordinated to meet the same target latency. This experiment illustrates when AP-Pred is effective/ineffective and why. It also demonstrates the power of Timeline approach on reducing tail latency over different workloads. Furthermore, it shows that speeding up long-running queries is critical on reducing tail latency, even when they may not have high parallelism efficiency.

7.2.3 Alternative Way to Selectively Parallelize Long-Running Queries

This experiment presents an alternative way, named RampUp, to identify long-running queries, and compares it with Ap-Pred and Timeline. RampUp increases parallelism degree of a request along its execution, starting from sequential execution. As a result, this approach completes short-running requests sequentially, whereas it allocates more cores to parallelize long-running requests without knowing query execution time. Different thread ramp-up intervals (5 ms, 10 ms, and 20 ms) are used to evaluate Ramp-up. For example, an interval of 5 ms means that an additional thread is added to the query with each 5 ms of query execution. The maximum number of threads per query are 6. The smaller the interval is, the sooner a query is parallelized. This section evaluates the pros and cons of the RampUp approach compared with the predictive solutions.

Figure 7.7(a) compares Timeline with RampUp for the 99th-percentile and shows that Timeline achieves lower tail latency than RampUp for a large range of loads up to 700 QPS. Timeline outperforms RampUp because it can fairly accurately predict query execution time. It makes the parallelism decision before executing a long-running query, whereas RampUp intrinsically has delay in adding up parallelism
Figure 7.7: Timeline, RampUp, and Timeline/w correction for reducing tail latencies.

while executing the query. If RampUp adds up parallelism too fast (e.g., at every 5 ms interval), it may effectively reduce tail latency at light load but it will incur high parallelism overhead at high load, similar as using fixed parallelism with high degrees.

Although RampUp is less effective on reducing 99th-percentile latency than Timeline, its new way of deciding long-running queries creates opportunities to tackle
extremely high tail, e.g., 99.9th-percentile, as shown next.

### 7.2.4 Reducing Extreme Tail Latency

For our workload, Timeline cannot reduce execution time of few long-running queries due to the limitation of prediction accuracy. To illustrate, there are 4% of long-running queries (>80ms) in our workload and our predictor has a recall value of about 86%. Therefore, our predictor correctly identifies most of the long-running queries, but there are $4\% \times (1 - 86\%) = 0.56\%$ of queries that are long-running but misidentified as short-running. Timeline will not parallelize those queries. Therefore, it guarantees a reduction of response times up to about 99.4th-percentile ($(100\% - 0.56\%) = 99.44\%$). For even higher percentile latency, e.g., 99.9th, a predictive approach may not be effective.

To deal with extremely high tail, RampUp can be used to correct Timeline’s possible misprediction, which we call this new approach **Timeline+Correction**. Under this approach, a query whose execution time reaches to a target completion time gets more threads (up to the maximum degree 6 in total) to speed up the execution. As shown in Figure 7.7(b), this simple correction reduces 35–64 ms for the 99.9th-percentile latency from Timline. Combining the two approaches makes a synergistic impact on reducing extremely high tail.

### 7.2.5 Sensitivity Studies

**The number of parallelism efficiency groups.** With more groups, predictive parallelization can potentially apply more accurate speedup efficiency information given query execution time. As shown in Figure 7.8, with the increase in the number of groups from 1 to 3, higher latency reduction is obtained. However, using from 3
Figure 7.8: AP-Pred with different groups for speedup curves.

groups (in Figure 7.8) to 6 groups (6 groups is obtained by evenly dividing each of
the 3 groups into 2 groups) does not lead to further improvement. This is because
the speedup profiles of queries among neighboring groups have become similar when
3 groups are even further divided. This supports why our algorithms use 3 groups of
parallelism efficiency profiles.

**System load metric.** Figure 7.9 compares various metrics for identifying instan-
taneous system load. The figure shows the results of our baseline (using the number
of active threads for long-running queries (LongT)), and two alternatives (using CPU
utilization (CpuUtil) and the total number of active threads (AllT)). The server re-
trieves CpuUtil by sending a query to Windows performance counters periodically,
for which we use Performance Data Helper (PDH) APIs [73]. The intervals are set to
25 ms in our system considering a tradeoff between more accurate information and
lower overhead.

Because CpuUtil is weighted for the past as illustrated in Section 6.2, Figure 7.9
shows that it performs worse than other two metrics in capturing instantaneous system load. The performance of CpuUtil becomes worse with increased load, as more queries are aggregated in a sampling interval. For example, under 100 QPS, the utilization for 30 past queries is aggregated. However, under 500 QPS, that for 150 past queries is aggregated, more likely to misrepresent the instantaneous load.

In contrast, accounting for the number of active threads is a good proxy to measure instantaneous system load. In particular, LongT is the best because threads running long-running requests are more likely to stay longer in the system, affecting the resource availability of the newly scheduled request, while short-running queries are transient and could have completed right after the new requests start.
Chapter 8

Conclusions

Web search has become an easy and popular way to access information on the web. Over the last decade, new devices have shown up, allowing people to access information at any moment using web search. Mobile devices, such as tablets and mobile phones, are such examples, and they are now a major source of search traffic. Even emerging wearable devices, such as Google Glass, also need associated web search services for seamless interactivity [6]. Therefore, people today have become dependent on the performance of web search in daily life.

A key factor in the success of web search engines has been their ability to rapidly find high-quality, highly relevant results to queries. Today, this is a challenging task for two reasons. First, the amount of data is growing very fast, and nowadays there are 15 billion web documents indexed in a major web search engine (e.g., Google or Microsoft Bing) [74]. At this scale, finding high-quality search results is time-consuming, and it is along the critical path for query processing. Second, modern processors have more cores rather than faster clock speeds. While in the past the processing time of a single search query had been improved for free with faster clock speeds over processor technology generations, this is no longer the case these days. In response to data and core counts scaling in the server, this thesis presents a new way of improving document index search without decreasing the quality of results. While web search engines operate under an SLA (e.g., 300 ms), this saved time in the index search can be used to better find the best matching documents, serve larger
indices, and host additional stages that improve user experience. In short, with these optimizations, users could see search results of higher quality while experiencing the same level of responsiveness.

Our approach was to reduce tail latencies within a single server without reducing quality. The first part of this thesis focused on parallelizing the query execution in web search servers to reduce their response time while providing the same response quality as in sequential execution. The proposed technique was dynamic fine-grain sharing that mimicked the sequential order of execution to parallelize each request with small speculative execution and good load balancing. Specifically, it partitioned the index data into small chunks, and when a thread became idle, it grabbed the most important unprocessed data chunk. While processing data chunks, the threads of a query communicated with each other to merge the top results they had found so far.

The proposed parallelization technique was implemented and evaluated experimentally on production servers and workloads. In the evaluation, it achieved good speedup for long-running queries, while short-running queries had little benefit from parallelization. This motivated the need for selectively parallelizing long-running queries, because parallelizing every query could result in much higher resource utilization than sequential execution and fail to reduce the response time.

This thesis employed pre-retrieval prediction of query execution time (i.e., before executing the query), and applied it to long-running query parallelization. By prediction, long-running queries could be identified and optimized at the earliest possible time — this is desirable for interactive workloads, like web search, where a small delay (e.g., 10 ms) is critical. Our predictor was accurate enough to effectively reduce tail latencies and efficient enough that it could be integrated in real server systems. To
provide high accuracy and efficiency, our predictor used cheap features and mainly considered query rewriting.

Next, this thesis presented new heuristic algorithms for tail reduction that exploited both query and system load information to decide parallelism degree of each individual query. It developed three factors (i.e., query execution time, parallelization efficiency, and system load) that were available in a search server and represented both query and system load information. Our predictor played a key role in accurately estimating query execution time and parallelization efficiency. Instantaneous load was captured for system load. As a result, at runtime, our algorithms selectively parallelized long-running queries with high parallelism efficiency and adapted the parallelism degree to system load.

Both proposed parallelization strategies and prior work were extensively evaluated on the production servers and workloads of the Microsoft Bing search engine. The results showed that, compared to executing queries sequentially, the proposed strategies reduce the 99th-percentile response times for queries by 70% under low-to-moderate load and by 45-70% under moderate-to-high load. No prior work has achieved such gains, as they have not exploited both query and load information. Moreover, an efficient fallback mechanism was used combined with the proposed strategies to recover from prediction errors and effectively reduced the extreme latency tail, e.g., 99.9th-percentile.

8.1 Future Directions

There are several possible research opportunities as follow-up to this thesis. First, predictive parallelization is a promising approach to reduce response time of various interactive workloads besides web search. These workloads include data analytics [75],
graph processing [76, 77], and MapReduce data processing [78–80]. Future work will be able to apply it to different workloads and resource bottlenecks. Second, evaluating our framework in a distributed system is needed to see bigger impacts of using predictive parallelization. In a distributed system, a server responding to a query slowly (i.e., straggler) comes to dominate overall performance of query processing even when all other servers are much responsive for the query. Moreover, the larger the scale is, the greater the impact of the straggler is. This problem can arise from several sources, including misprediction (of long-running queries as short-running ones) and transient load spike in a server. We believe that coordinating the parallelism decision among servers in a cluster to remove such stragglers will be of great benefit, effectively optimizing the latency that end users experience.

There are several trends in the server design that drive future research related to predictive parallelization. First, as processors are expected to continue having more cores than faster clock speeds, the maximum parallelism degree used in a server is likely to be larger. The task queue is a single point of contention in the current design of work sharing for parallelizing a single query. In such case, work stealing can be an alternative, and comparing the two design models with respect to the number of cores will be a meaningful study. Second, low tail latencies and popular use of larger SSD allow a search server to host larger indices. A server will then experience a larger memory footprint, and I/O latency can take a large portion in query processing time. In the future, index servers would need judicious memory management policies so that each index server spends its time for evaluating relevant documents, not waiting for I/O operations. Third, although this work assumes index servers that are comprised of homogeneous cores, future servers are expected to be more heterogeneous to add processing power under power constraints. Exploiting speedup information in the
heterogeneity for parallelizing queries is not trivial and opens up interesting research opportunities.


