Reducing Query Latencies in Web Search using Fine-Grained Parallelism

Eitan Frachtenberg, Microsoft
eitanf@microsoft.com

April 30, 2009

Abstract

Semantic Web search is a new application of recent advances in information retrieval (IR), natural language processing, artificial intelligence, and other fields. Our group (Powerset) develops a semantic search engine that aims to answer queries not only by matching keywords, but by actually matching meaning in queries to meaning in Web documents. Compared to typical keyword search, semantic search can pose additional engineering challenges for the back-end and infrastructure designs. Of these, the main challenge addressed in this paper is how to lower query latencies to acceptable, interactive levels.

Index-based semantic search can include numerous synonyms, hypernyms, multiple linguistic readings, and other semantic information, both on queries and in the index. In addition, some of the algorithms can be super-linear, such as matching co-references across a document. Consequently, many semantic queries can run significantly slower than the same keyword query. Users, however, have grown to expect Web search engines to provide near-instantaneous results, and a slow search engine could be deemed unusable even if it provides highly relevant results. It is therefore imperative for any search engine to meet its users' interactivity expectations, or risk losing them.

Our approach to tackle this challenge to exploit data parallelism in slow search queries to reduce their latency in multi-core systems. Although all search engines are designed to exploit parallelism, at the single-node level this usually translates to throughput-oriented task parallelism. This paper focuses on the engineering of two latency-oriented approaches (coarse- and fine-grained) and compares them to the task-parallel approach. We evaluate on Powerset's deployed search engine the various factors that affect parallel performance: workload, overhead, load balancing, and resource contention. We also discuss heuristics to selectively control the degree of parallelism and resulting overhead on a query-by-query level. Our experimental results show that using fine-grained parallelism with these dynamic heuristics can significantly reduce query latencies compared to fixed, coarse-granularity parallelization schemes. Although these results were obtained on, and optimized for, Powerset's semantic search, they can be readily generalized to a wide class of inverted-index search engines.

Keywords

Semantic Web; Search engines; Performance evaluation; Multi-core processors

1 Introduction

Web search engines are an essential tool for today's Web users. Semantic Web search engines such as Powerset's index-based search[1] carry the promise of increased utility by gleaning semantic meaning from indexed documents, later matching it to the semantic information in user queries [1]. But their usefulness hinges on good performance: low throughput will adversely affect the economic viability of the service, while high response times will drive users away. For traditional search engines and IR systems, responsiveness and interactivity are relatively well understood and managed. Semantic Web search, however, is required to handle much larger amounts of dynamic or static data per query, and some of the algorithms involved may be super-linear in the input data, as is the case for some natural-language algorithms. On the other side of search we have humans, whose perception and patience limits remain

[1] Powerset is a Microsoft company. The main interface to the search engine can be found at `http://www.powerset.com`

relatively unchanged. Consequently, these applications are at risk of losing users—unless they are engineered differently than traditional systems, with query latencies as a top priority.

Until recently, it was easy to reduce the latency of a given algorithm and associated inputs by just waiting: each processor generation was faster than the previous generation. But recent technological trends have not only stopped this passive gain, and even reversed it. Processors have stopped growing faster, because of energy and temperature constraints, and instead have grown more parallel and sometimes slower [2, 3, 4]. On the other hand, the capacities per dollar of primary and secondary storage keep increasing at exponential rates. Combined with the growth of data on the Web, a search engine's back-end processor is now required to process more data per cycle, not less, exacerbating the latency problem. Processing more data sequentially in a node is thus infeasible, because more data processed sequentially leads to unacceptable response times. Keeping the size of the data fixed per node defeats the economic gains of increased storage capacity as well as of chip parallelism. It also increases the economic cost of growing the index size, since it requires adding compute nodes, instead of fully exploiting the existing nodes. Barring any unforeseen breakthrough in IR algorithms or single-core performance, parallelization within the node is therefore the only cost-effective way to increase search capacity and throughput while keeping response times from growing.

If indeed intra-node parallelization is a technological imperative, what is the best layer in the algorithm to parallelize? In other words, what is the best parallelization granularity in the node's computation? The technological trends described above point to two dimensions of performance and capacity scaling: increased processor parallelism and increased storage. The introduction of many-core processors (such as Intel's Larrabee [4]) will further increase the gap between data processing capacity and sequential speed. Traditional Web search engines can exploit increased parallelism by using a coarse-grained task-parallel approach: each core processes a different query, and single-threaded latency of each query is controlled by limiting the data per node or per query. Although this works well to increase throughput, this approach may prove quite wasteful as the data capacity of each node increases beyond its utilization. But more importantly for semantic search, coarse-grained parallelism does not always suffice for interactive latencies, given its complex algorithms.

The main motivation for this paper is to explore the feasibility of fine-grained parallelization in this application domain. To this end, we evaluated several parallel models on Powerset's deployed search engine, comparing the performance of the new fine-grained approach to the more common coarse-grained parallel approaches. Our main contributions are:

1. A fine-grained parallel algorithm for index-based search in general, and semantic search in particular, that translates increased parallelism to reduced latencies.

2. Two novel flexible and dynamic parallelism heuristics, arising from the fine granularity of this algorithm. In sequential query execution, fixed-size index partitions impose a lower bound on response time and limit machine utilization in low-load conditions. Our dynamic approach is more flexible than the current execution model since it can dynamically control the balance between processing power and data size on each node. This allows improved resource utilization in unloaded conditions, leading to faster response times with fewer nodes. Moreover, it expedites the software's adaptation to changes in the underlying hardware balances. Choosing the right parallel granularity is key to attaining good scalability, and this choice itself is sensitive to changes in the underlying hardware [5].

3. A discussion of several performance factors and evaluation approaches with a rigorous experimental evaluation on a deployed Web search engine.

The rest of this paper is organized as follows. Section 2 overviews index-based search engine architecture, as well as the relevant details of Powerset's back-end architecture and the experimental methodology used throughout the rest of the paper. The next three sections describe three parallelization approaches for multi-core back-end servers, including our novel query-level parallelization and heuristics. These are followed by a discussion of their performance and scalability characteristics in Sec. 6. Finally, we survey the more general field of parallel and distributed IR systems in the related work (Sec. 7) and offer conclusions and future directions in Sec. 8.

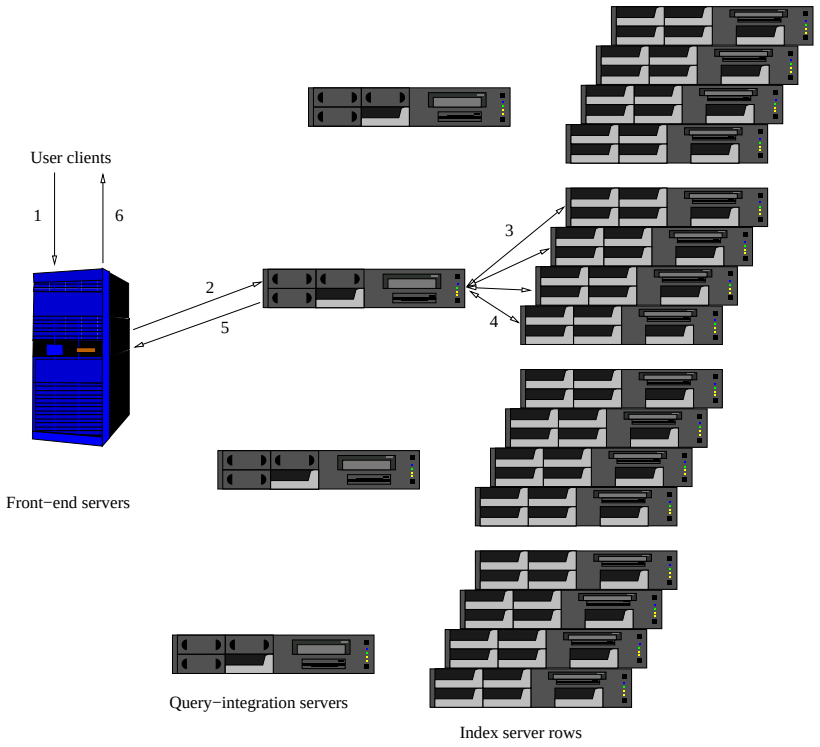


Figure 1: Generalized index-based search engine architecture. A query starts when a client sends a user query over the Web to a cluster of front-end servers (1). The query is then sent to one of several identical query-integrator (QI) servers, based on load-balancing factors (2). The QI sends the query (3) to a *row* of index server nodes (IS), so that the aggregate index shards in the row cover the entire index. Each IS computes the best hits for the query in its index shard and returns it to the QI (4), which combines all the results into the top-scoring results, and returns them to the front end (5). Lastly, the front-end servers format the results for display, possibly adding more data such as snippets and highlighting, and send it back to the user client (6).

2 Architecture and Methodology

A typical index-based search engine works by comparing terms in a query with an index of terms[2] from a corpus of documents. The index (or *inverted list*) resembles a book's index, in the sense that it contains a data structure of search terms (called *postings*) and a mapping from each term to the documents in which it appears (called a *posting list*). Every query is processed and reformulated (for example, to include stemmed forms of nouns and verbs), and compared against some or all of the posting lists of its reformulated terms. The document IDs that are found to match are called *hits*. All hits are then scored based on sophisticated algorithms, and sorted to find their relative rank [6, 7]. Finally, the front-end returns snippets and links for the top-scoring hits to the end user. This short description omits many implementation details that can be found in the literature [6, 8, 9, 10], but these are immaterial to the parallelization of the search back-end.

Powerset's search engine architecture resembles that of a typical clustered Web server [10, 11]. Fig. 1 shows the typical route of a user query in such a system. The front-end nodes pick a back-end cluster (or *row*) based on load-balancing criteria [8]. Each row has one or more query integration programs and multiple index server dæmons on compute nodes. These nodes contain a striped subset—a fixed partition, or shard—of the index, so that a query passed to the query integrator, distributed to index servers, and aggregated (sorted together) back at the query integrator, contains results from the entire index.

[2]Typically, but not always, these terms represent keywords: a subset of normalized tokens found in the indexed documents. Some services search for additional information, (such as syntactic and semantic roles and relationships, in Powerset's case), but the basic algorithm described here remains the same.

While there are several different approaches to semantic search [12, 13, 14], Powerset's search engine[3] generalizes the typical keyword search engine architecture described above and in past studies [6, 8, 9]. To match semantic meanings in the queries and documents, Powerset employs natural-language parsing technologies, developed at Xerox PARC over the last three decades. Specifically, Powerset uses a Lexical Functional Grammar (LFG) parser called XLE to parse both indexed documents and queries [15, 16, 17]. XLE creates a syntactic representation of the sentences, which is then transformed into a semantic representation. This representation adds lexical information, such as synonyms and hypernyms, and normalizes semantically equivalent structures.

There are two components in our system that use this technology: crawling documents offline and serving queries at runtime. On the offline side, each document crawled and indexed by our search engine is parsed for linguistic and semantic information. For example, if a document contains the sentences "John bought a cake. He ate it", we store the facts about buying and eating a cake (and their synonyms), as well co-reference information (meaning that "He" refers to John and "it" refers to the cake). The storing of this extra information in the index does come at a significant size increase compared to a keyword-only index for the same documents.

On the runtime side, there is an additional step (between steps 3 and 4 in Fig. 1), wherein the query integrator employs additional query processing prior to sending it to the index servers. This step involves parsing the query and building the semantic and linguistic information that is to be matches against the query. For example, the query "who had a cake and ate it too?" would match the above document because it does have someone buying (which by a hypernym reading is a form of having) a cake, and someone eating a cake. Since this paper is concerned with the parallelization of the index server multi-core node, this extra step does does not affect the performance or generality of our results. Unlike typical keyword search, however, we do search for the constraint that the same person having the cake is the person eating the cake. This involves a super-linear algorithm that can slow down some queries by an order of magnitude. This algorithm and others, combined with the additional semantic information stored in the index, mean that Powerset's semantic search can be more data-intensive than keyword search for the same inputs. Nevertheless, the results discussed here can be applicable to keyword search as well, since the architectures are similar and the technological trends are also driving keyword search to more data and more cores per node.

Hardware configuration and experimental setup We ran all experiments on the actual production code of Powerset's search engine, using an index of Wikipedia[4] documents and a random sampling of queries from our query logs. Each experiment consisted of clearing the operating system cache, restarting the search processes, and launching 12,000 non-unique queries from a client machine. We remove the first 2,000 queries from the resulting logs, to allow for postings and file cache warm-up. This methodology achieved repeatable latency statistics with less than 1% variability. We used the same hardware platform as our production machines (a Dell PowerEdge 2950 server with eight Xeon 2.4GHz cores and 16GB of RAM), which is representative of contemporary search engines. We varied parameters such as degree of parallelism, parallelization algorithm, and index partition size. For the latter, we started with a basic "unit" index of an approximate size of 31GB on disk, which was used as a performance baseline since it offers acceptable latencies in sequential execution. But because we are interested in scaling behavior on the storage axis as well,[5] we also evaluated index partitions with two and four times the number of documents, denoted respectively as 2X and 4X, the latter representing our target disk and RAM utilization for this hardware.

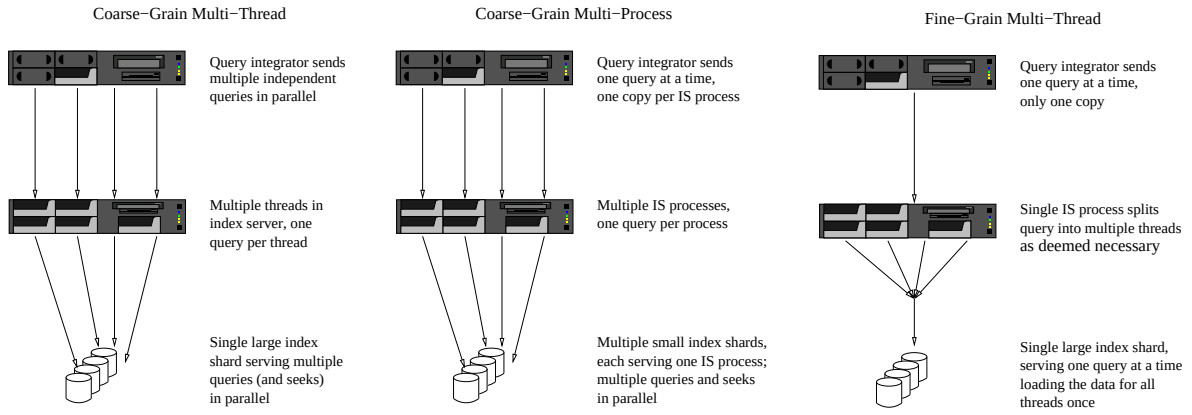
In terms of metrics, we are primarily concerned with query latencies, and to a lesser extent, throughput and CPU allocation efficiency. We additionally introduce a more subjective metric of responsiveness: the percentage of queries that complete under an *interactivity threshold*, set somewhat arbitrarily to 1*s*. To reduce the effect of saturation load on our measurements, we avoid oversubscribing CPU cores in our experiments. In other words, the number of threads concurrently serving queries never exceeds the number of cores. The topic of performance under load receives a short treatment in the discussion below but merits its own extensive study, to be followed up in a future publication.

This paper examines three software architectures to parallelize queries–the existing throughput-oriented approach and two novel latency-oriented approaches, which are the main focus of this paper. These architectures are briefly summarized in Fig. 2 and explained in detail in the next three sections. The rest of the paper discusses refinements and comparisons across the metrics of these approaches.

[3]Not to be confused with Microsoft's Live! search, which uses a different code base.

[4]`en.wikipedia.org`

[5]By storage we refer to both primary (RAM) and secondary (disk) storage. In practice, most of the program's RAM is dedicated to caching posting lists from disk, so the two are closely coupled.



(a) CGMT: simple to implement designed for high utilization under load

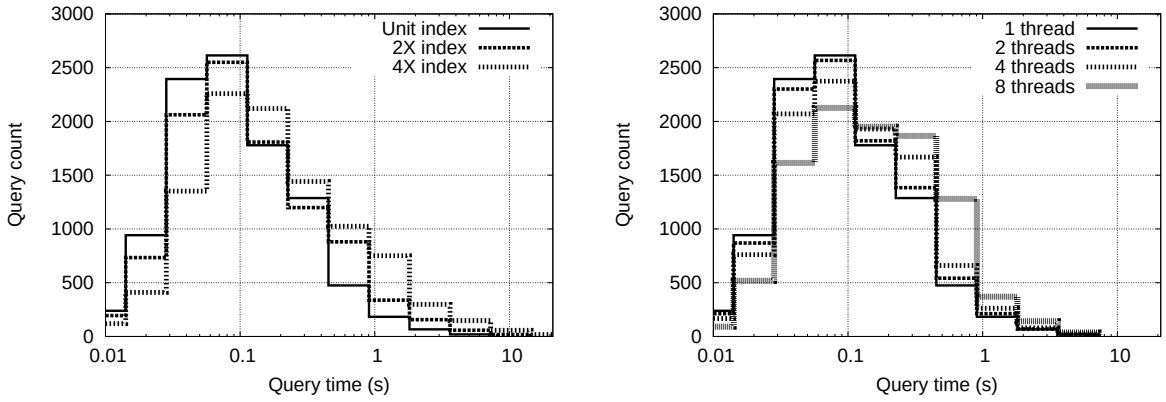
(b) CGMP: simple to implement designed for low query latencies

(c) FGMT: Adaptive to workload optimized for minimum seeks and latencies

Figure 2: Architectural overview for three different query parallelization techniques

3 Coarse-Grained Multi-Thread

As described in the introduction, to get performance improvements from newer generations of processors, applications must be parallelized for multi-core execution. And the back-end index serving application is no exception. This section introduces a simple and widely used scheme of multi-threading for index-based search engines. In this approach, which we refer to as coarse-grained multi-thread (CGMT), we run multiple independent queries on a single index partition per node, each in its own thread (Fig. 2(a)). Additional cores translate to more concurrent threads and queries running on the same index, thus increasing throughput. This approach is relatively easy to implement and offers good throughput scaling with increased cores, as shown later. But it does not offer good latency scaling on the parallelism and storage axes.



(a) Scalability with index size, single thread

(b) Scalability with threads, unit index

Figure 3: Run time distribution for queries with three index sizes

Figure 3 illustrates these scaling issues. On the storage axis (left), it depicts the run time distribution histogram of the 10,000 queries on three index sizes, run sequentially one query at a time. The histogram shows a power-law "long-tail" distribution (note the logarithmic x-axis). It also shows that for this architecture, this index partition size is quite reasonable for our desired interactivity level: 9,853 of the 10,000 queries complete in under one second, and only 3 queries take longer than 5s. If, however, we multiply the index size by 4, as shown in the overlaid histogram, only 9,289 of the queries complete in under a second, and 80 of the queries on the long tail take 5s or longer to run. To maintain acceptable latencies, we could bound the index partition to a predetermined size (e.g., unit index)—at an economical cost—or we could employ heuristics to limit the search space to a subset of the index (e.g., static ranking and timeouts [18]). Consequently, CGMT does not scale very well on the storage axis: either index size or coverage

must be limited.

On the parallelism axis, Fig. 3(b) shows similar latency histograms as we increase the number of independent queries that the index server handles concurrently, up to the hardware capacity of 8 threads. Obtaining good throughput requires a high sustained load, to keep all cores working on separate queries; but increasing the number of concurrent independent queries also increases competition over the posting lists in storage, since each query may require different posting lists. This competition can lead to contention over serializing resources, such as memory banks or hard drives, and slow down queries significantly. The figure shows that as the number of independent queries executing on the same node and contending for shared resources grows, the histograms shift their main weights right to higher latencies. Interactivity subsequently also suffers, reducing the sub-second queries to 9,743 when running 8 independent threads.

Combining the two axes by running parallel threads on the 4X index amplifies this performance degradation, as evident in Fig. 4 (four leftmost boxplots in each cluster). As we add more competing tasks, the query latency is pushed up across the spectrum, but most noticeably on the higher end: the number of non-interactive queries balloons from 711 with one thread to 1813 with all eight cores running queries, and the 95^{th} percentile latency grows from 1.34s to 2.16s.

Because of these scaling issues, we looked for different ways to exploit multiple cores. If we consider CGMT as task-parallel, since threads run different tasks, the next two approaches may be considered as data-parallel, where threads divvy up the data for a single task.

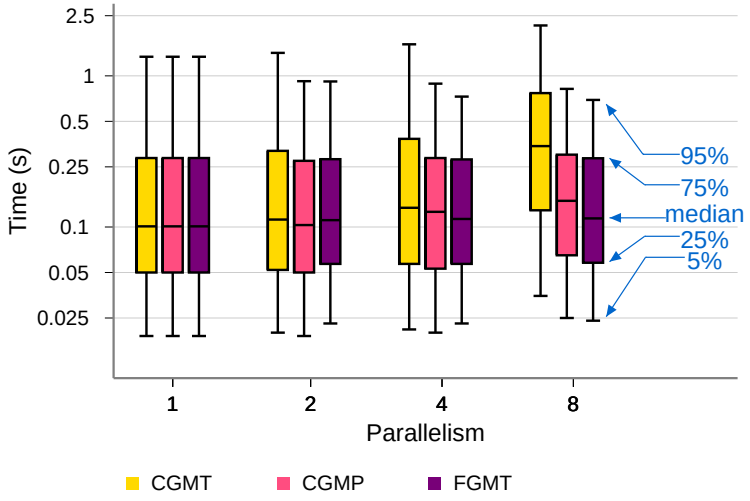


Figure 4: Latency distributions of 10,000 queries on the 4X index (logarithmic scales). To help with color distinction, the left-to-right ordering of the bars in each group corresponds to the ordering of the legend, as is the for all figures.

4 Coarse-Grained Multi-Process

Our first attempt represents an even simpler approach than the first: parallelize each query at the process level. Instead of writing threading support in the search application, we can just run multiple copies of the index server, each on its own core and using its own fixed index partition. In essence, each process behaves as if though it was running on its own node in the distributed environment.[6] This simple approach (referred to as coarse-grained multi process, or CGMP in Fig. 2(b)) takes advantage of the increase in both chip parallelism and storage capacity without having to rewrite the code or taking elaborate thread-safety measures. Instead of running a single index server with a 4X index, we could run two processes with a 2X index each, and half the RAM for cache; or 4 processes with a unit index each and a quarter of the RAM for cache; and so on, until the cache becomes too small. The query integrator later aggregates the results from the different processes as it normally does in the distributed environment.

[6]This model can be extended further to running each process in its own virtual machine, but even then, in reality the processes share some of the machine's resources.

We ran the same workload with CGMP and plotted its latency distributions with increasing parallelism in Fig. 4 (second boxplot in each set). The results show that halving the search space for each index server and running two parallel processes does reduce query latencies, most noticeably in the 95^{th} percentile, which is reduced by 31% on the first halving. Halving it again to 4 processes only yields marginal improvement in 95^{th} percentile latency (3.6% reduction) and even small *increases* in the lower latencies. By using 8 half-unit indexes, all latencies have now increased noticeably, except for the very slowest queries that show another modest improvement. Although this approach scales better than CGMT as we enlarge the index and add processors, only the slow queries truly see a benefit, and even that with diminishing returns.

There are several factors that could contribute to this lackluster scalability, including: I/O contention; system noise and index variability; query integrator aggregation; and various unidentified overhead and contention factors. To single out the most influential of these factors, we turn to Fig. 5, that shows the latencies in CGMP with several variations. The first set represents the unmodified 4-process CGMP with each process using its own different unit

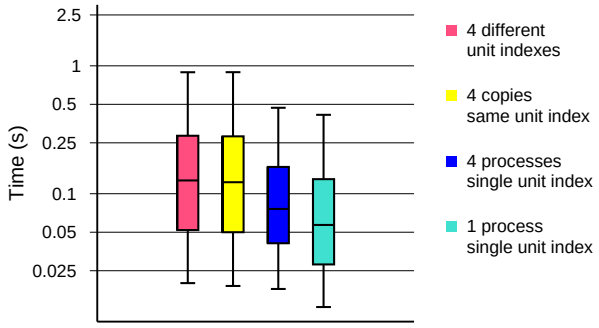


Figure 5: I/O effect on the CGMP approach

index (reproduced from Fig. 4). The next set uses four exact copies of the same unit index slice, thereby eliminating variations in the input data. The similarity of these two sets suggests that the variability in the data hardly affects query latency. The third experiment uses the exact same unit index for all four processes–they all point to the same physical index slice on disk, thereby avoiding I/O contention. And to eliminate further the overhead of multi-processing, query integration, and reduced cache, we also include the results for the single-process single-unit index from Fig. 3. Again, the relative closeness of the last two sets suggests that having four processes or one process makes relatively little difference if they both use a single unit index on disk.

The largest latency drop, between the second and third sets, occurs when changing from four different copies of the same index in the file system to a single index, even with four processes. This suggests that the dominant factor in the latency cost of CGMP is not the additional data from using a larger index, but rather I/O contention. Also supporting this hypothesis that I/O contention inhibits data scaling more than index size is the relatively small performance difference between the third and fourth sets. In other measurements, we found that the majority of the I/O cost stems from disk seeks rather than read size. As long as disk seeks have a significant limiting and serializing effect on query latency, this approach and its hybrids remain inherently nonscalable. This finding motivated us to find a scaling approach that maintains or reduces the number of disk seeks.

5 Fine-Grained Multi-Thread

To scale well on both the storage and parallelism axes, without sacrificing latency or index coverage, we developed another data-parallel approach (referred to as fine-grained multi-thread or FGMT in Fig. 2(c)) that introduces another layer of data parallelism: thread-level parallelism within a query. FGMT allows the index size to grow with storage and processing capacity, while still searching all of it; but to reduce latency, it first coalesces all disk reads in a single process, and then distributes the index data among the parallel threads. So when the index server receives a query, it loads from storage (disk or cache) the required posting lists, initializes the query execution, and then spawns lightweight threads, one per core. Each thread receives an equal-sized subset of document IDs to scan, together covering the entire index partition. All threads execute the same code on the same query, but with private data structures.

The only writable shared data structure is a heap of the top-scoring hits, protected by a lock. At the end of the threads' execution, the heap contains the highest-scoring hits in the entire index partition, which is then transmitted to the query integrator as usual. Since the index contains skip-lists that permit near-random-access to any document ID, and since hits are generally distributed evenly in the index, our measurements show that all threads complete their work with little variability, within 1–2% of each other.

5.1 Comparison to coarse-grained approaches

We refer to Fig. 4 once again to see the results of our FGMT tests. The following observations arise from comparing FGMT to the other approaches:

- The performance in the sequential case is identical for all approaches, as expected.
- For an equal number of threads, FGMT exhibits lower latencies than CGMP for the median and slower queries. At 8 threads FGMT shows significantly lower latencies, and about half the non-interactive queries as CGMP.
- FGMT *reduces* the latency of long-tail queries index with increased parallelism, like CGMP and unlike CGMT, and exhibits the lowest 95^{th}-percentile latency overall, ($0.693s$). Even with a smaller unit index and 8 threads (not shown), FGMT has 95^{th}-percentile latency of $0.418s$, compared to CGMT's $1.015s$. In fact, the overall latencies for 8-way FGMT with a 4X index are relatively close to those of 8-way CGMT with only a unit index (worse on the 5^{th}-percentile, similar on the median, and better at the long-tail), while serving four times the data.
- FGMT can reduce variability in latencies, leading to a more homogeneous and predictable workload.
- FGMT can hurt fast queries: going from sequential execution to parallel actually worsens 5^{th}-percentile latencies, going from $0.014s$ to $0.024s$.

It is this last observation that leads us to the next questions: why does fine-grained parallelization not always reduce query time? which queries does it accelerate, and which does it slow down? and can we tell the former from the latter in advance, and selectively turn parallelization off where it would only do harm?

5.2 Deciding when to parallelize

Parallelization has a cost, and speedups are rarely linear. This is a consequence of factors such as overhead, resource contention and certain serialized elements of the code. Some of these elements are an unavoidable part of the algorithm (e.g., reading the posting lists from storage) and limit the maximum attainable speedup, as per Amdahl's Law [19]. To get good speedups despite these factors, one must scale the problem to larger sizes as well [20]. Indeed, growing the index size to 2X and 4X does improve the parallel speedup and efficiency for many queries, especially on the long tail of the distribution. But many short queries do not have enough work to amortize the cost of parallelization, as evidenced by the growth of the 5^{th}-percentile latency with increased parallelism. This growth is shared by all three approaches, but is perhaps most easy to address in the FGMT approach, since the degree of parallelism is determined on query-by-query level, requiring no interaction between different queries (threads or processes).

We therefore developed the following heuristic to try to predict which queries will parallelize well and which will not, based on observations described later in Sec. 6. When a query comes in, it is first run sequentially for a small subset of the index partition (say, 1000 documents). At this point, we evaluate the ratio of hits to documents—if it is below a predetermined threshold, we assume the query will not parallelize well, and continue the sequential execution until the query is complete or the threshold is met. If and when the threshold is met, we distribute the remaining index documents among a number of threads, as described earlier for FGMT. We tried a number of different threshold values, and empirically determined that a threshold of 1% of the documents predicts parallelization well for most queries. Higher thresholds also work, obviously, but needlessly preclude some queries.

The results of the comparison of this heuristic to CGMT and CGMP are shown in Fig. 6. Compared to the other parallel approaches, this heuristic *successfully reduces the parallel short latencies to the same levels as the sequential ones*. Since many fast queries—and a few slow ones with few hits—now run sequentially, and subsequently faster, all percentiles are pushed down. The number of non-interactive queries, however, is only reduced by 30 queries, since non-interactive queries typically already have enough hits to be parallelized anyway. As for the overhead of the heuristic, the initial sequential execution adds negligibly to the latency of most queries, and offers an additional advantage of run time prediction, as we now turn to discuss.

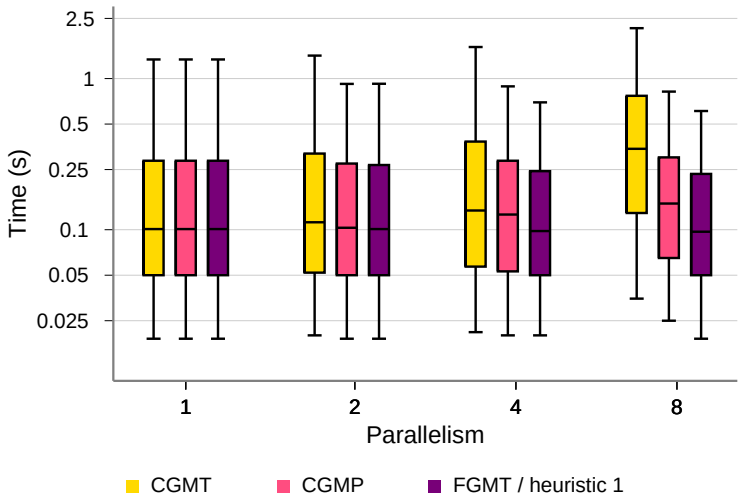


Figure 6: Run time distributions of 10,000 queries for the 4X index, using first FGMT heuristic

5.3 Limiting parallelism

One of the interesting observations we made is that parallel efficiency diminishes with more threads, even for queries with good parallelization potential, as seen in Table 1. Sources of parallel inefficiency include: serialized initializations; memory, cache, and lock contention; overhead; and miscellaneous code issues. Our data also show that not all queries gain equally from parallelization: some queries (most notably simple keyword queries) do not have enough work per hit to amortize the serializing elements of the query, and exhibit poor speedup scaling. Consequently, more parallelization can also mean more wasted resources. Throwing more cores at queries eventually hits diminishing returns. But even if everything was perfectly scalable, consistently using all cores can incur additional performance penalties, as evidenced in Fig. 4 for CGMT.[7] Ideally, we would like to be able to direct more resources only at those queries that not only can benefit from, but also require, higher levels of parallelism.

threads	1	2	4	8
speedup	1	1.82	3.51	6.13
efficiency	100%	91%	88%	77%

Table 1: Median parallel speedup of the 100 slowest queries and their efficiency, defined as speedup divided by the number of threads.

Most queries just do not run long enough to require full parallelization, even if they parallelize well. In terms of human perception, the difference between 100*ms* and 500*ms* is not as acute as that between 1*s* and 5*s*. We will therefore focus our attention on accelerating the perceptively slow queries, those slower than an interactivity threshold. Even for these slow queries we can limit their resource allocation to a more efficient thread count—just enough to change their responsiveness' perception. This would leave more cores available for other queries, whether for increasing throughput, for cooling down, or for reducing power consumption.

Our second heuristic attempts to predict the sequential run time of a query by running on a small subset of the index, and extrapolating the run time linearly to the entire index size. This extrapolation is justified by the observations that hits are distributed uniformly, and that search time grows approximately linearly with index size. Empirically, we found that a subset of 1,000 documents gives an accurate-enough extrapolation and adds less than 2*ms* to the average query's latency.

We can thus implement the heuristic as follows. We first execute each query sequentially for 1,000 documents. If the query has any parallelization potential, (over 10 hits, as determined by the previous heuristic in Sec. 5.2),

[7] One important factor in this performance degradation is that with all 8 cores running user processes, operating system activities add noise and variability that degrade overall performance [21].

	sequential time	parallel time	speedup	hits
parallel time	0.99			
speedup	0.61	0.44		
hits	0.69	0.84	0.69	
query words	0.03	0.01	−0.04	−0.10
index size	0.98	0.98	0.97	1.00

Table 2: Pearson correlation coefficients between pairs of sequential latency, 8-way FGMT parallel latency, parallel speedup, hit and word counts, and index size. Except for the last row, index size is set to the 4X index. In the last row, index sizes of 1X, 2X, and 4X are correlated with the mean of each factor.

we estimate the sequential run time of the query and divide it by the latency threshold we allow for interactivity. The resulting quotient is the number of threads we use to execute the query on the rest of the index (capped by the number of cores). As a margin of safety for inaccurate estimates, we use a latency threshold of $0.33s$ (one third of the interactivity threshold).

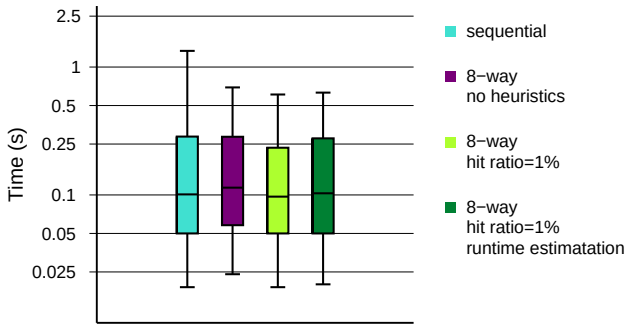


Figure 7: Effect of parallelization and heuristics on FGMT's performance

Figure 7 summarizes the performance of each version of FGMT. Starting from sequential execution with the 4X index, the initial version of FGMT successfully accelerates slow queries, but at a cost for fast queries. The first heuristic (Sec. 5.2) runs the fast queries sequentially, thus lowering the latencies across the entire distribution and even reducing 95^{th}-percentile latency by 12%. Lastly, the second heuristic gives up some of this latency reduction for improved resource allocation, with a rather minor effect on query latencies, increasing the 95^{th}-percentile latency by only 3.4%. Similarly, the number of non-interactive queries only goes up by 6 queries. But the overall CPU consumption (defined as the sum of products of each query's run time by its thread count) is reduced by 21.2% with this heuristic. Combined with intelligent scheduling, this "green" approach could potentially lead to significant energy savings in unloaded systems, or higher throughput in loaded systems.

6 Discussion

This section provides in-depth motivation for the FGMT heuristics, as well as a comparative analysis of the strengths and weaknesses of all the approaches described to this point.

To develop a heuristic that can reliably predict whether a query can parallelize well, and to what extent, we measured the pairwise correlation coefficient between multiple query factors (Table 2). In addition, we varied the index partition size from 1X to 4X and correlated it to the four performance factors (last row Table 2). Looking at the data both validates several performance assumptions and provides some interesting insights (in logical sequence):

- Run time, both sequential and parallel, is nearly linear in the data size, as evidenced by the tight correlation between run time and index size. This justifies the linear extrapolation of run time for a short sequential prefix.

- Speedup is also correlated with the index size, meaning that the more data (and run time) we have, the better the opportunity to parallelize. Index size, however, is fixed and not query-dependent, so it cannot be used as a speedup predictor.
- The strong link between sequential and parallel run times is again evident in their 0.99 correlation factor, and also by the positive correlations between speedup and run times (second row). This suggests that a property of a query that predicts high latency also predicts good parallelization. Sequential run time is obviously one such property, and our initial heuristic just looked at the time to run the first 1,000 documents sequentially as a speedup predictor. But this measure is not a static property of the query, since it depends on changing factors such as system load and hardware.
- Our search for a static property started with the number of words in the query (fourth row). This provided surprisingly little correlation with run time, and even a small negative correlation with speedup, concluding that at least in our semantic search engine, the "difficulty" of a query is not clearly related to its length.
- Looking at the number of matching documents, or search hits (third row), we found that it correlates very well with both run times and parallel speedup. It might even serve to explain the small negative correlation between query words and speedup, since generally speaking, longer queries yields fewer hits (evidenced by the -0.1 correlation between them).
- Furthermore, the perfect correlation between index size and hits confirms our assumption that hits are uniformly distributed across the index. this is an important assumption for using a prefix of the index for prediction, because a linear run time extrapolation will no longer work with a skewed distribution of hits.

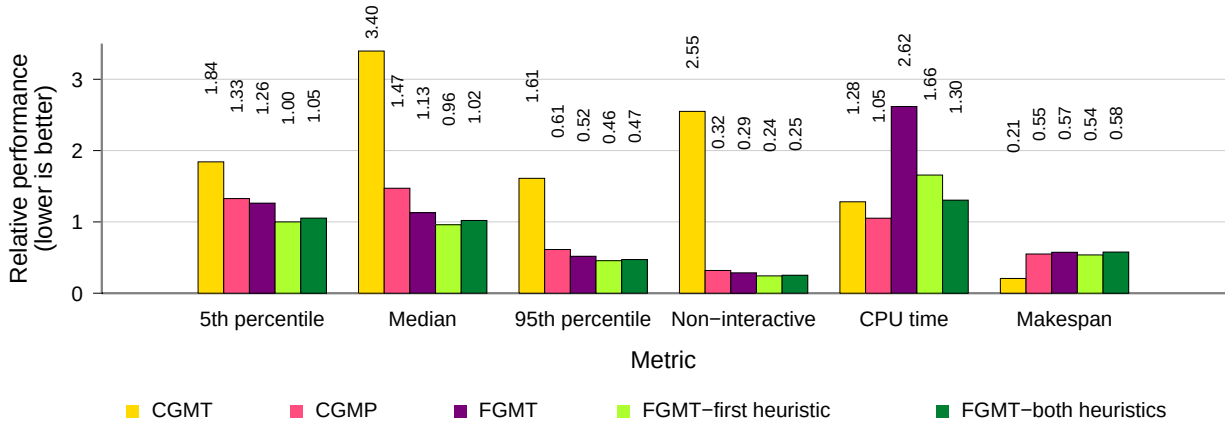


Figure 8: Overall performance comparison of parallel approaches ($y = 1$ corresponds to the sequential performance for each metric). All tests performed on a 4X index with 8-way parallelism. Metrics (clusters) are 5^{th}, 50^{th}, and 95^{th} percentile latencies, number of non-interactive queries, and total CPU seconds and wall-clock seconds to run the 10,000 queries.

To better understand some of the performance tradeoffs, we compared six metrics for CGMT, CGMP, FGMT, and the two heuristics (Fig. 8). To use a common scale, all performance results are normalized to a baseline, such that sequential performance on the 4X index always equals one. The following are a few noteworthy observations:

- For the faster half of the queries, parallelization hurts latency: any performance benefits it may have over sequential execution are countered by the additional overhead and contention. Only FGMT with the first and second heuristic can provide the low-overhead of sequential execution for fast queries (5^{th}, 50^{th} percentiles), because they do indeed run those sequentially.
- For slow queries (95^{th}-percentile and non-interactive queries), all parallelization schemes except for the throughput-oriented CGMT successfully reduce query latencies, with a small performance advantage to FGMT, especially with the heuristics.

- The second heuristic has in fact virtually no negative impact on slow queries, compared to the first heuristic. On the other hand, it uses 22% less CPU time. And although both coarse-grained approaches use less CPU time than FGMT, this does not translate to faster queries. One of the main reason for their higher latencies is that both suffer from disk seek contention: for CGMT, with different queries that suffer from cache misses concurrently, and for CGMP, from any cache misses (because all processes run the same query and have the same cache, they all miss together, creating 8 concomitant disk seeks).
- The makespan metric approximates the reciprocal of throughput in these experiments, because the system is underloaded (to obtain an accurate measurement of throughput we must measure at a load close to the system's saturation point). Not surprisingly, the throughput-oriented CGMT dominates this metric, with small differences among the other approaches.

In summary, we see that the choice of appropriate parallelization scheme is closely tied to the metric we are trying to optimize. If the foremost priority is reducing query latencies and non-interactive occurrences, as is the case in the comparatively slow semantic search, then FGMT with the first heuristic offers the lowest run times across the board. If minimizing CPU time is more important (from which higher utilization can sometimes be derived), then CGMP has the advantage, although one must be careful of its inherent scalability limitations. FGMT with the second heuristic might strike an even better latency/utilization tradeoff. And finally, if the most important consideration is throughput, as is the case for many keyword search engines, then CGMT offers the best performance.

7 Related Work

Because of its economic importance, improving the responsiveness of Web search engines and IR systems is a topic of extensive research. For example, numerous studies discuss improvements to latency through caching or reduced I/O operations [22, 23, 24]. Disk-seek latency often plays a pivotal role in search responsiveness, because it has not kept up with the increases in processor performance, so its relative part in the latency, compared to fast computations, has been steadily growing [6]. As discussed earlier in this paper, disk-seek latency can play an important part in parallelization choices as well, and can be the limiting factor to scalability in approaches such as CGMP.

Generally speaking, a Web search engine works by crawling the Web for documents, parsing the documents for search terms, and creating an index of inverted files, linking each term to the documents in which it appears [25]. Past work discussed the parallelization of most of these phases. On the indexing side, for example, servers can parallelize in the crawling, document parsing, and inverted-file-creation [26, 9, 27]. On the query side, most search engines exploit both task parallelism, in the sense that multiple queries are executed concurrently in a cluster, and coarse-grained data parallelism, in the sense that a single query execution typically spans multiple nodes [10]. Because of disk and memory constraints, virtually all Web search engines split up the index into multiple partitions on nodes, using distribution and replication [10, 25]. Each query is then processed in parallel by each constituent node in a row, and multiple queries are distributed across replicated rows. These parallel mechanisms are used for several important reasons:

- Higher throughput using task parallelism—serving multiple distinct queries concurrently.
- Lower latencies using data parallelism—distributing the index to different partitions to be searched in parallel.
- Fault-tolerance—having replicated copies of the index helps to ensure continuous service in the presence of component failures.
- Load-balancing—multiple rows can distribute query load and improve responsiveness.

Distribution and replication in IR have been studied extensively in the past. Dias *et al.*, for example, discuss the benefits of striping the index across multiple back-end nodes, with the front-end nodes detecting and working around failures in the back-end [8]. In another example, Schroeder et al. surveyed and classified different distributed load-balancing schemes in Web servers, based on the network level in which routing decisions are taken [11]. Another study that specifically focused on search engine parallelism used the MOSE search engine to analyze the tradeoffs between task and data parallelism, and concluded that a tuned combination of both offers the best throughput [10]. Data parallelism is still rigid in the sense described above: since the index is partitioned into fixed-size units, each then processed sequentially, the granularity of parallelism is statically defined. In MOSE, the computation on each

partition is stack-based, requiring significant effort to parallelize further. This fixed data-parallel scheme characterizes most Web search engines in the literature [6, 25].

In the more general IR field, there exist numerous works on creating distributed and parallel index and database partitions [24], and even on updating the partitions in parallel [28]. As in the other studies, each fixed-size partition is processed sequentially during search. Other studies discuss parallelization in the context of improving the I/O performance of large IR systems [29, 30]. In practice, Web search engines typically cache disk data to mitigate I/O costs [22, 23]. Still, most search engines remain sensitive to I/O seek time on cache misses, as demonstrated in Fig. 5.

This paper discusses a parallelization approach that has finer-granularity than the works cited here, which in turns allows for the heuristics described in Sec. 5. These optimizations can only be implemented when the basic unit of parallel computation on a node is finer than a single "atomic" query, the level that is typically handled by these previous studies.

8 Conclusions and Future Work

There are components in contemporary server nodes that are constantly growing—such as storage and processing capacities—and others that remain relatively constant—such as storage and processing speeds. The goal of this study was to explore different methods to parallelize index-based Web search that scale well with the former but do not increase the requirements from the latter. This paper discusses two simple coarse-grained approaches to parallelization and introduces a new fine-grained approach. This approach represents the most scalable way of the three to addressing increased storage and cores while maintaining a latency envelope, which is the primary performance concern for Powerset's semantic Web search engine. Other search applications that prioritize latency and have relatively costly computation can also benefit from this approach, or alternately use the coarse-grained multi-threaded approach for increased throughput, or the coarse-grained multi-process approach for lower CPU time.

We confirmed experimentally that the main obstacle to scalability in the coarse-grained approaches is that since independent search queries all require their own data, they contend for the slowest system components—in our case, disk seeks—which in turn force a degree of serialization on the queries. The primary advantage of the novel fine-grained approach is therefore that it allows search queries to share the serializing elements while parallelizing the independent elements. Compared to the coarse-grained approaches, the fine-grained approach is successful in significantly reducing the latency of slow ("long-tail") queries. For example, in our workload, this approach reduces the 95^{th}-percentile latency by 31%–71% compared to the coarse-grained approaches while staying on par or better with the faster queries.

The second important advantage of the fine-grained approach is its flexibility and adaptability for resource allocation at run time. Whereas the coarse-grained approaches rely on static allocation of data to tasks based on a fixed index partition size, the fine-grained approach can dynamically allocate threads on a query-by query basis. The heuristic to control the degree of parallelism can also limit it for fast or fast-enough queries, thereby increasing efficiency. Since the coarse-grained approaches cannot dynamically control parallelism, they consequently yield higher latencies for fast queries as well, showing slowdowns of 33%–84% compared to the fine-grained parallel approach. This flexibility can become an increasingly important advantage in the future as we consider factors such as transient load, varying latency thresholds, and even heterogeneous server architectures.

Future Directions There are numerous interesting questions following from this study. In the immediate term, we plan to study the performance of all these algorithms under load: What do their throughput and saturation-point look like? How effective can hybrid approaches combine throughput and responsiveness? Can the fine-grained approach perform better by adjusting the parallelism of queries in response to transient load, query complexity, latency and throughput balancing, or even power considerations? How will it affect scheduling of queries? Conceivably, this approach even opens the door for dynamic re-allocation of data to threads mid-flight through the execution of a query. Another interesting direction would be to quantify and model how much increase in node parallelism is required to exactly offset a given growth in storage capacity, which can be later used to plan hardware requisition to meet latency and throughput goals.

Acknowledgements I would like to thank my team at Powerset for their help and support, and in particular David Simpson for his work on the experimental framework, and Chad Walters for his numerous suggestions for improving this paper.

References

[1] B. T. Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American* , May 2001.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," in *27th International Symposium on Computer Architecture (ISCA)*, pp. 248–259, June 2000. Available from `citeseer.ist.psu.edu/agarwal00clock.html`.

[3] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "A multi-core approach to addressing the energy-complexity problem in microprocessors," in *Workshop on Complexity-Effective Design (WCED)*, June 2003. Available from `citeseer.ist.psu.edu/kumar03multicore.html`.

[4] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics* **27**, pp. 1–15, Aug. 2008. Available from `portal.acm.org/citation.cfm?doid=1360612.1360617`.

[5] Intel, "Developing multithreaded applications: A platform consistent approach," Feb. 2005. Version 2.0, Chapter 3. Available from `intel.com/cd/ids/developer/asmo-na/eng/53797.htm`.

[6] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks* **30**, pp. 107–117, Apr. 1998. Available from `citeseer.ist.psu.edu/brin98anatomy.html`.

[7] J. Zakos and B. Verma, "A novel context-based technique for Web information retrieval," *World Wide Web* **9**, pp. 485–503, Dec. 2006. Available from `www.springerlink.com/content/dm7r46l5g557h285/`.

[8] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A scalable and highly available Web server," in *Compcon'96 'Technologies for the Information Superhighway', Digest of Papers*, pp. 85–92, Feb. 1996. Available from `citeseer.ist.psu.edu/dias96scalable.html`.

[9] D. Harman and G. Candela, "Retrieving records from a gigabyte of text on a minicomputer using statistical ranking," *Journal of the American Society for Information Science* **41**, pp. 581–589, Dec. 1990. Available from `asis.org/Publications/JASIS/Best_Jasist/1991CandelaandHarman.pdf`.

[10] S. Orlando, R. Perego, and F. Silvestri, "Design of a parallel and distributed Web search engine," in *Proceedings of the Conference on Parallel Computing (ParCo'01)*, pp. 197–204, Imperial College Press, Sept. 2001. Available from `citeseer.ist.psu.edu/orlando01design.html`.

[11] T. Schroeder, S. Goddard, and B. Ramamurthy, "Scalable Web server clustering technologies," *IEEE Network* **14**, pp. 38–45, May-June 2000. Available from `www.cse.unl.edu/~goddard/papers.html`.

[12] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs, "Swoogle: A search and metadata engine for the semantic web," in *Proceedings of the 13th ACM International Conference on Information and Knowledge Management (CIKM'04)*, pp. 652–659, (New York, NY, USA), Nov. 2004. Available from `http://portal.acm.org/citation.cfm?doid=1031171.1031289#`.

[13] A. G. Maguitman, F. Menczer, F. Erdinc, H. Roinestad, and A. Vespignani, "Algorithmic computation and approximation of semantic similarity," *World Wide Web* **9**, pp. 431–456, Dec. 2006. Available from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.1007`.

[14] L. M. Stephens, A. K. Gangam, and M. N. Huhns, "Constructing consensus ontologies for the semantic web: A conceptual approach," *World Wide Web* **7**(4), pp. 421–442, 2004. Available from `http://portal.acm.org/citation.cfm?id=1017683.1017685#`.

[15] D. G. Bobrow, B. Cheslow, C. Condoravdi, L. Karttunen, T. H. King, R. Nairn, V. de Paiva, C. Price, and A. Zaenen, "PARC's bridge and question answering system," in *Proceedings of the Workshop on Grammar Engineering across Frameworks (GEAF'07)*, pp. 46–66, CSLI, (Stanford, CA), 2007.

[16] R. M. Kaplan and J. Bresnan, "Lexical-functional grammar: A formal system for grammatical representation," in *The Mental Representation of Grammatical Relations*, J. Bresnan, ed., 1982. Available from citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.3002.

[17] R. M. Kaplan, J. T. M. Iii, T. H. King, and R. Crouch, "Integrating finite-state technology with deep LFG grammars," in *Proceedings of the Workshop on Combining Shallow and Deep Processing for NLP (ESLLI'04)*, Aug. 2004. Available from citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.8628.

[18] M. Richardson, A. Prakash, and E. Brill, "Beyond PageRank: Machine learning for static ranking," in *Proceedings of the 15th International Conference on World Wide Web*, pp. 707–715, ACM, (Edinburgh, Scotland), May 2006. Available from research.microsoft.com/~mattri/papers/www2006/staticrank.pdf.

[19] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 483–485, Apr. 1967.

[20] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM* **31**, pp. 532–533, May 1988. Available from http://portal.acm.org/citation.cfm?id=42411.42415.

[21] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *19th ACM International Conference on Supercomputing*, pp. 302–312, (Boston, MA), June 2005. Available from www.cs.huji.ac.il/~feit/papers/Noise05ICS.pdf.

[22] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *Proceedings of the Annual International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 183–190, ACM, (New York, NY, USA), 2007. Available from portal.acm.org/citation.cfm?id=1277775.

[23] L. Cherkasova and G. Ciardo, "Role of aging, frequency, and size in Web cache replacement policies," in *Lecture Notes in Computer Science*, **2110**, pp. 114–123, Springer-Verlag, Jan. 2001. Available from citeseer.ist.psu.edu/cherkasova01role.html.

[24] A. MacFarlane, J. A. McCann, and S. E. Robertson, "Parallel search using partitioned inverted files," in *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE)*, pp. 209–220, IEEE Computer Society, 2000. Available from http://portal.acm.org/citation.cfm?id=829519.830829.

[25] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys* **38**, July 2006. Available from http://portal.acm.org/citation.cfm?id=1132959.

[26] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 137–150, USENIX Association, (Berkeley, CA, USA), Dec. 2004. Available from labs.google.com/papers/mapreduce.html.

[27] F. Silvestri, S. Orlando, and R. Perego, "WINGS: a parallel indexer for Web contents," in *Proceedings of the International Conference on Computational Science*, pp. 263–270, (Krakow, Poland), June 2004.

[28] A. MacFarlane, J. A. McCann, and S. E. Robertson, "Parallel methods for the update of partitioned inverted files," *ASLIB Proceedings: New Information Perspectives* **59**(4/5), pp. 367–396, 2007.

[29] F. J. Burkowski, "Retrieval performance of a distributed text database utilizing a parallel processor document server," in *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems (DPDS)*, pp. 71–79, ACM, 1990. Available from portal.acm.org/citation.cfm?id=319057.319065.

[30] Z. Lin and S. Zhou, "Parallelizing I/O intensive applications for a workstation cluster: a case study," in *Proceedings of the Workshop on Input/Output in Parallel Computer Systems*, pp. 17–36, (Newport Beach, CA), 1993. Available from citeseer.ist.psu.edu/37062.html.