Operational Requirements for Scalable Search Systems

Abdur Chowdhury, Greg Pass
America Online
cabdur@aol.com, gregpass1@aol.com

ABSTRACT

Prior research into search system scalability has primarily addressed query processing efficiency [1, 2, 3] or indexing efficiency [3], or has presented some arbitrary system architecture [4]. Little work has introduced any formal theoretical framework for evaluating architectures with regard to specific operational requirements, or for comparing architectures beyond simple timings [5] or basic simulations [6, 7]. In this paper, we present a framework based upon queuing network theory for analyzing search systems in terms of operational requirements. We use response time, throughput, and utilization as the key operational characteristics for evaluating performance. Within this framework, we present a scalability strategy that combines index partitioning and index replication to satisfy a given set of requirements

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software – *distributed systems, performance evaluation.*

General Terms

Performance, Design, Theory.

Keywords

Operational requirements, search scalability.

1. INTRODUCTION

Prior research has examined how to efficiently index text documents and resolve text queries: for example, with inverted-indices [3], signature files [8], or sparse matrices [9]. Further improvements to these index structures have been made for handling special query types [10, 11, 12] and reducing I/O overhead [13, 14, 15]. While much work addresses this index-level view of search performance, little work addresses performance at the architectural level of a complete search service. At this level, operational requirements, such as delivering sub-second response times to millions of simultaneous users, set the context for evaluating search performance.

In this paper, we consider three operational requirements when designing a search service: throughput, response time, and utilization. We provide a framework for analyzing and comparing architectures and achieving these operational goals. This type of evaluation is of great interest to search services like Yahoo, MSN, and AOL, whose products' success depends upon the quality of service. For example, such a service needs to understand the effect on response time if the number of users is doubled, and at what point the bottleneck of the system will begin to impact the system throughput. Using this framework, we also examine a scalability strategy that combines index partitioning and replication to meet operational requirements imposed on search systems.

2. PRIOR WORK

Stanfill first explored the use of a partial inverted index distributed among multiple computers to support large text collections [16]; the implementation used a connection machine. Frieder examined document clustering across CPUs to optimize search processing time, also with the use of a connection machine [17]. Macleod et al. presented some of the initial motivation behind distributed search, exploring data partitioning and replication strategies for meta-data [18]. Danzig et al. proposed a collection selection architecture in which partitions were selected via a title search [19]. Harman et al. [20] described a prototype distributed IR system in which collection data is stored centrally but maintained in separate indices organized by content; they also examined caching of popular data. Tomasic et al. [21] explored various index organization structures and provided a simulation of those strategies to examine which factors contributed most to efficiency. This study is closely related to ours in terms of examining bottlenecks and utilization of the system, but differs from ours in that the tradeoffs of throughput, index size, and response time are not examined.

Moffat et al. examined distributed search in the TREC conference [22], but only in terms of effectiveness. Gravano et al. and Callen et al. explored the collection selection problem as a means of improving distributed search and addressing effectiveness [23, 24, 25, 26]. Again, this work focused on the collection selection problem as an effectiveness problem, whereas we are providing a framework in which distributed search services are evaluated in terms of efficiency.

Couvreur et al. examined three different search architectures (document representations) via a simulation and explored response times for various loads [27]: inverted indices were not partitioned or replicated, although signature file and in-memory systems were used to examine the scalability of these options. Bestavros explored the use of collection replication based on usage; this work is comparable to document retrieval optimizations for web servers rather than for search [28]. Viles examined the effectiveness of several collection division strategies in which collections were partitioned randomly, and studied the effect on efficiency for maintaining effectiveness [29, 30]. Bailey et al. examined parallel search on the PADRE system, a distributed in-memory pattern matching system, and explored the issues of scaling it to one terabyte [31]. Additionally, their scaling approach partitioned data to many CPUs of a virtual machine -- an approach we show is not always reasonable.

More recently, Cahoon et al. studied the performance of various distributed architectures and provided a simulation comparing the approaches [32]. This is similar to our framework with several fundamental differences: first, they assumed that users are a closed set and thus they modeled their system a closed network; we show this is not representative of a search service. Also, Cahoon et al. did not present a methodology for data partitioning in terms of throughput and response time. Cahoon et al. followed up that study [33] by examining architectures under various workloads; again, they examined the system as a closed system. Ribeiro-Neto et al. examined partitioning data among machines based on terms, but found bad performance in terms of effectiveness; they also provided a simulation validating results, but did not provide any results on how a practitioner would partition their collection when considering throughput and response time [34]. De Kretser et al. examined the efficiency of distributed search over a local and distributed LAN, and provided timings comparing the various architectures [35]. Lu et al. examined scaling in terms of threads on an SMP machine [36]. Lu assumed that queries arrived at the system as a Poisson process, which we also consider a valid assumption.b Lu et al. also examined partial collection replication with the InQuery system as a means of improving distributed search time by reducing the amount of data actually searched, and provided a simulation validating their results [37, 38].

3. RESPONSE TIME SCALABILITY

In order to provide an attractive product to its customers, a search service must guarantee some acceptable average response time for the service. For example, it would not be acceptable to keep a user waiting ten minutes at a web browser for search results. While acceptable response times vary, most search services try to provide sub-second response times.

Given such an operational requirement, we must first characterize the average response time of the system, and second understand how to scale the system to satisfy the requirement.

3.1 Characterizing Response Time

The architecture of a search service typically consists of a frontend process, such as a web server, and a backend search process. The frontend accepts a query from a user, issues that query against the backend search process, and then constructs and returns a search results page to the user. The search process retrieves and ranks the search results from a search index. This architecture is depicted in Figure 2.

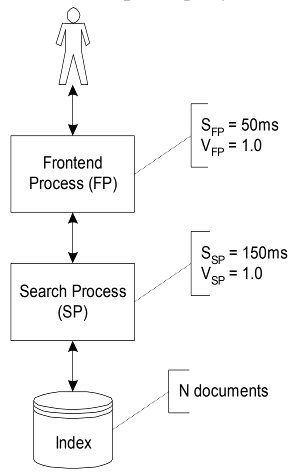

Figure 2: Simple Search Service Architecture

Consider a single query flowing through this system. The system response time will be the sum of the service times for each node in the architecture. The service time is the mean time for each node to process a single query. For example, if the search process takes 150ms to perform the search and the frontend takes 50ms to handle the query and results, the total response time will be 200ms. Therefore, to determine the response time of the overall system, we can analyze the service time characteristics of each node individually and sum them.

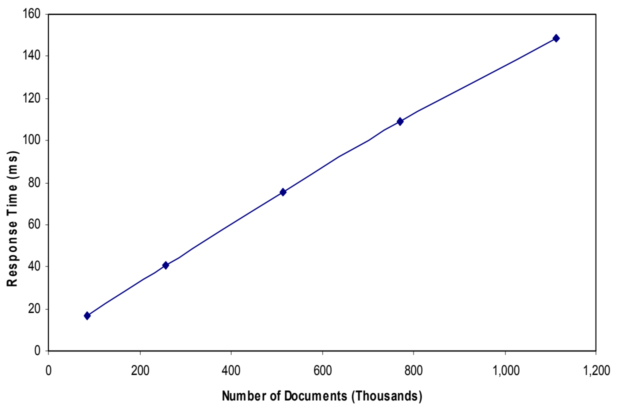

Figure 1: Average Response Time vs. Number of Documents

First, we characterize the search process. Prior work has characterized the performance of the search process in terms of the number of documents in the search index and the types of queries requested. These contributing factors have either been modeled or simulated [21, 6, 34]. Tomasic & Garcia-Molina [21] argue that sampling provides a more accurate result, but is not as general in nature as a model. For our purposes, the mean service time of a node is sufficient for characterizing the node; therefore, sampling may be an appropriate approach.

We sample queries from search service query logs to collect a representative set of queries. We calculate the mean service time using this set of queries against our search index. To determine the number of queries required to achieve a representative average service time, we calculate the sample size of queries, $ss = (z^2\sigma^2)/\beta^2$, where z is the confidence level value, σ is the sample standard deviation, and β is the error rate.

Figure 1 shows the results of this analysis, plotting the average service time versus the number of documents in the index. For our system, the relationship between index size and response time is linear. We can use these results to characterize the service time of the search process node.

Next, we characterize the frontend process. The average service time of the frontend process can be determined using this same methodology. Unlike the search process, however, the performance of the frontend will typically be a fixed cost independent of the number of documents in the search index.

Clearly, there are additional factors, such as network latency, that are not being included in this calculation of the overall average response time; however, prior work and the experimental results of this paper show that it is the processing nodes that typically characterize and are sufficient to predict the overall response time of a system.

3.2 Meeting Operational Requirements

Keeping with the example search service above, say we now add an operational response time requirement of 100ms. As the average response time of the example system is 200ms, we need to identify a strategy to reduce the response time. In this section, we examine two such strategies: first, speeding up the existing architecture; and second, altering the architecture via partitioning of the search index.

To achieve the first strategy, speeding up the existing architecture, we need to identify the slowest node in the architecture -- the bottleneck. The overall system can never be faster than the bottleneck node; therefore, improving the performance of this node will have the greatest impact on the overall response time.

Borrowing from operational analysis, we define the demand of a node i, D_i, to be V_i*S_i, where S_i is the mean service time of the node and V_i is the visit ratio. The visit ratio is the probability that the node is visited during the fulfillment of a single query. In our example, both the frontend and backend nodes have a visit ratio of 1.

The demand indicates the total amount of time each node requires to service a single query. The node that requires the greatest amount of time will naturally be the bottleneck of the entire system. In this example, the search process is the bottleneck, with a demand of 150ms versus the frontend process's demand of 50ms.

To improve the performance of the search process, we can look at software improvements, such as optimizing the organization of the underlying index [3, 1, 2]. This strategy has been the focus of most of the performance work for IR systems. We can also look at hardware improvements, such as upgrading the speed of the CPU. There are limits to both of these approaches: improving the software is not always possible due to third party software or lack of developer resources, for example; and improving the hardware is not always cost effective.

Let us assume that either of these limits is the case and the search process node cannot be improved. The second strategy we examine to reduce the response time of the system is partitioning the search index across parallel search process nodes. Data partitioning has been examined before [27]; here we use partitioning to achieve a required response time. This architecture, shown in Figure 3, is referred to as a fork-and-join architecture: a single search is forked to multiple, parallel search processes, and the results of each process are joined to obtain the complete search results.

In the example shown, each search index now contains one half of the total number of documents. Correspondingly, the response time of each search process will be reduced according to the relation derived in the previous section. In this example, the response time is reduced from 150ms to 75ms. As the two search processes execute in parallel, the total response time for both process will be the same as the response time for each process, 75ms.

This architecture must also include a merge process that combines and sorts the result lists from each search process. While the true cost of the merge process is dependent on the number of result lists being merged, modeling this process as a fixed, low cost is needed, but has been removed from other examples in this paper for the sake of clarity.

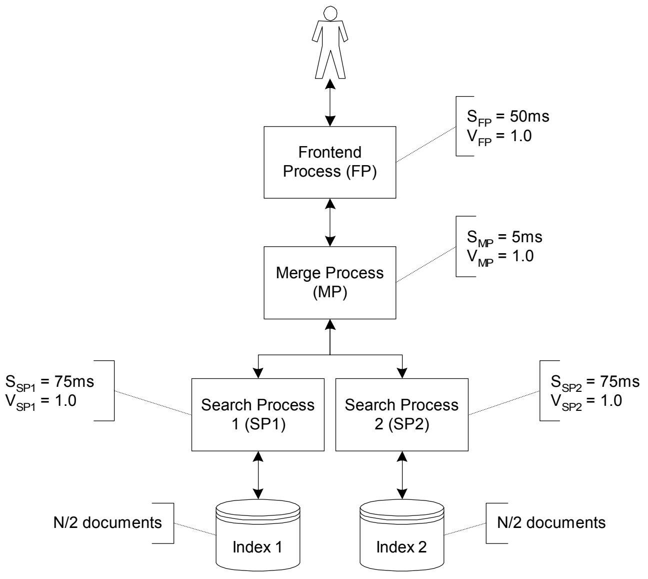

Figure 3: Search Service Architecture with Index Partitioning (Fork and Join)

For the given example, partitioning the search index into two equal-sized partitions reduces the total response time of the system from 200ms (50ms + 150ms) to 130ms (50ms + 5ms + 75ms).

More generally, we can calculate the number of partitions, *P*, required to satisfy a given response time requirement, $P = N / F(R - S_{NSP})$, where *N* is the total number of documents to be searched, *R* is the required response time, S_{NSP} is the sum of the response times of all non-search process nodes (i.e., the frontend and merge processes), and *F* is a function (or lookup) relating response time to index size. The factor (R - S_{NSP}) equals the required response time of the search process, S_{SP}, and the value of the function, *F*, therefore, is equivalent to the number of documents that can be searched in a single partition.

4. THROUGHPUT SCALABILITY

The prior section considered response time with respect to a single query flowing through the system. In reality, of course, there are multiple queries arriving and flowing through the system at any given time. In this section, we introduce a second operational requirement, throughput. Throughput is the number of queries serviced by the system in some unit of time. For example, we can say a given system's throughput is two queries per second (qps).

4.1 Calculating Throughput

Queries arrive at the system at some average rate. Most simulations and queuing models use the Poisson distribution to model this arrival process. As any given query arrives at the system, therefore, it is possible that at certain times that query may have to wait for a previous query (or queries) to complete. This query will wait in queue, and any time spent waiting will contribute to the response time. As the arrival rate fluctuates around its average, the size of these queues will grow and shrink.

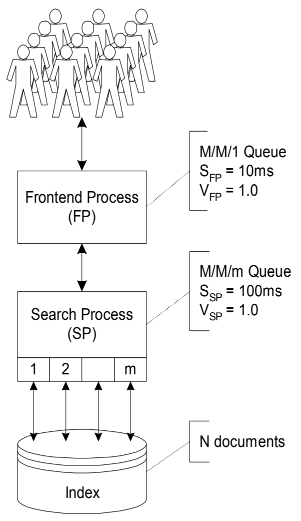

Figure 4: SMP Search Service Architecture

In queueing networks, we distinguish between closed systems and open systems. A closed system has a fixed number of jobs (or queries): each job can be either actively flowing through the system or waiting to enter the system. For example, consider an airline reservation system accessed through a fixed set of terminals. At any given time, some of the terminals are being used to make reservations and the rest are waiting for the operator to make the next request. Closed systems have been examined in the content of information retrieval by Cahoon et al. [33] in their simulation.

In an open system, at any given time, there are new jobs entering the system and completed jobs leaving the system; thus, the total number of jobs is not fixed, as in a closed system, but continuously varies with time. In other words, the arrival process for an open system is external to the system itself, while the arrival process for a closed system is a function of the same fixed set of jobs cycling through the system.

We believe an internet search service is best described as an open system: it accepts queries from an unknown population of users external to the system, and returns search results to this same external population. This choice impacts the queuing network model we use for performance evaluation.

As the throughput of the arrival process increases, the probability of queueing increases. Past some point, the queues will only continue to grow, and the system will not be able to keep up with the arrivals. This point, therefore, is the maximum throughput the system can handle.

In the previous section, we used operational analysis to identify the bottleneck node in a given system. The bottleneck, as the node with the greatest demand, will also be the node to incur the greatest queueing effects. Thus, the bottleneck node establishes an upper bound on the throughput of the system: $1/D_{\text{bottleneck}}$. At this throughput, the bottleneck node will be continuously servicing queries; thus if the arrival throughput increases, the system will be overtaxed.

The effects of queuing complicate the response time calculation presented in the previous section. For example, say we have a system with a mean service time of 500ms. If one query arrives at time zero, and a second query arrives 500ms later, the response time for both queries will be 500ms, and the average response time of the system will also be 500ms. However, if in a given second two queries arrive at the same time (still a 2qps arrival rate), the response time of the first query will be 500ms, and the response time of the second query will be 1000ms (due to queueing). The mean response time of the system, therefore, is 750ms.

To formally analyze this problem we can view it as a network of queues. As in the prior section, we analyze each node independently, now accounting for both service time and queuing. This analysis is referred to as product form analysis in queuing theory. Queuing theory models nodes differently depending on their operational characteristics. For this analysis, we selected the *M/M/m* queue to model our nodes. The first *M* represents a Poisson arrival distribution; the second *M* represents an exponential service time distribution, and *m* is the number of CPUs in a given node. This allows us to model both single CPU machines (when *m* = 1), and multiple CPU machines.

Now, to calculate the response time for any node, we can use Equation 1. Note that response time is equivalent to the service time, plus any time spent waiting in the queue.

$$rt = \frac{1}{\mu}\left(1 + \frac{\varsigma}{m(1-\rho)}\right)$$

Equation 1: M/M/m Queue Response Time

To calculate response time, we must determine the probability of queuing, ζ, and our traffic intensity, ρ. A node's traffic intensity is determined by the operational characteristics of the arrival rate at that node, λ, and the service rate, μ.

$$\varsigma = \frac{(m\rho)^m}{m!(1-\rho)} * \left[1 + \frac{(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1}\frac{(m\rho)^n}{n!}\right]^{-1}$$

Equation 2: M/M/m Queue Probability of Queuing

$$\rho = \frac{\lambda}{(m\mu)}$$

Equation 3: M/M/m Queue Traffic Intensity

Given the response time for each node and depending on the architecture, the overall system response time is the summation of individual nodes.

For a more thorough review of queuing theory and its assumptions see [39]. Queuing theory makes assumptions about the arrival process and service time distributions being modeled. It has been shown that even if these assumptions are not strictly satisfied, the overall analysis of the system when dealing with a network of queues is very resilient [40, 41, 42].

4.2 Meeting Operational Requirements

In this section we explore two strategies for meeting an operational throughput requirement: first, we use a SMP (symmetric multi-process) machine to service queries in parallel within a single search process node; and second, we replicate the complete search indexes across multiple search process nodes.

In order to highlight just the throughput requirement, let us consider a system that satisfies a response time requirement without index partitioning (as explored in the last section). Say the system has a mean service time of 110ms: 10ms for the frontend process and 100ms for the search process. The response time requirement is 200ms and the throughput requirement is 50 qps.

In the first strategy, we employ multiple CPUs to search a single search index. In this case, the search process node is modeled using an *M/M/m* queue. The single search process node queues queries and distributes them among the *m* CPUs. This architecture is depicted in Figure 4. While each CPU will have the exact same response time characteristics, by executing in parallel the throughput of the system increases.

Our second strategy is to replicate the entire search index on a number of machines. Each index will be searched by a single CPU and is represented as an *M/M/1* queue, as shown in Figure 6. The frontend process sends queries to the search processes in round robin fashion: if there are *P* search processes, each search processes will see just *1/P* of the query arrivals. This is reflected in the fractional visit ratio of each search process node.

Both of these strategies increase throughput by adding more processing power. In terms of queuing theory, the primary difference is that in the SMP solution all of the CPUs draw from the same queue, while in the replicated solution each search process has its own queue. Figure 5 compares the two strategies in terms of response time for a given throughput and varying numbers of CPUs or nodes. We see that the SMP strategy always delivers the lowest response time. Next, in Figure 7, we fix the number of CPUs in both strategies and compare response times for varying throughputs. Again, the SMP strategy is the clear winner. In terms of queuing theory, a single queue will always be more efficient than multiple queues for the same arrival process.

Let us now consider hardware cost: is it more effective to purchase and operate a large 24 CPU server or 24 single CPU servers? Based upon performance alone, using a large multi-CPU server is the best strategy; however Figure 8 tells a different story. Using the exact same CPUs for both strategies, the SMP configuration becomes vastly more expensive as the number of CPUs increases.

As a simple metric for comparing the cost benefits of competing architectures, we calculate the throughput per thousand dollars for a given response time requirement. For example, referring to Figure 7 and Figure 8, for a response time requirement of 400ms, the replication configuration can service about 1qps/$k, whereas the SMP configuration can service only .25qps/$k. The SMP solution is four times as costly.

5. RESPONSE TIME, THROUGHPUT, AND UTILIZATION SCALABILITY

In the last two sections, we addressed the response time requirement by partitioning the search index, and the throughput requirement by replicating the search index. In this section we introduce a third operational requirement, utilization, and present a mixture model strategy combining index partitioning and index replication to meet all three operational requirements.

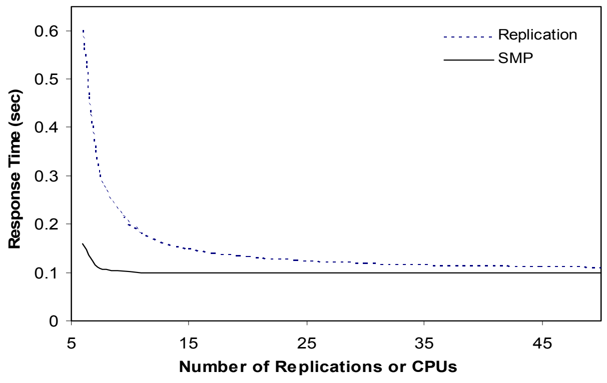

Figure 5: SMP vs. Replication - For a fixed throughput of 50 qps, varying CPU/Nodes

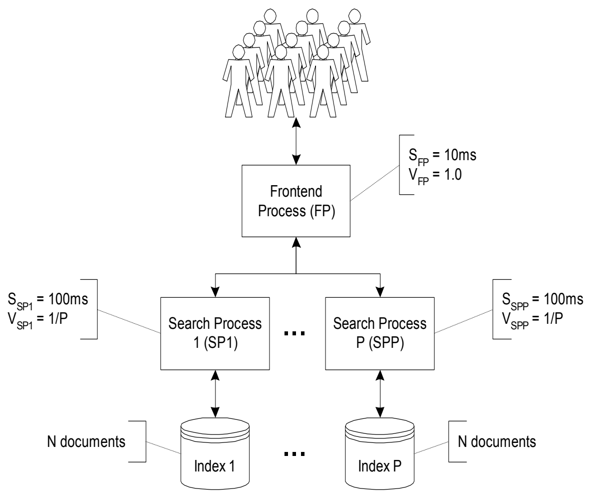

Figure 6: Search Service Architecture with Index Replication

5.1 Utilization

In the previous section, we identified the maximum throughput a given system is capable of servicing, and noted that at this maximum throughput the bottleneck node is servicing queries continuously. Continuous, or 100%, utilization of a CPU or other resource, however, can have adverse operational consequences. For example, consider a multi-server service that is running at 100% CPU utilization: if one server goes down, the throughput requirement for that system cannot be sustained. This same consequence could arise if the throughput of the arrival process increased unexpectedly (an open network cannot necessarily restrict this), or if, for a hardware upgrade, it was necessary to bring down a server but still keep the service operational.

Utilization is defined as the percentage of time a node is working, or busy, over some time-period, and can be derived from the throughput and service time characteristics of that node. For a given node, i:

- Throughput, X_i = (number of completions)/time = C_i/T
- Service Time, S_I = (busy time)/(number served) = B_i/ C_i
- Utilization, U_i = (busy time)/time = $B_i/T = (C_i/T)(B_i/ C_i) = X_i S_i$

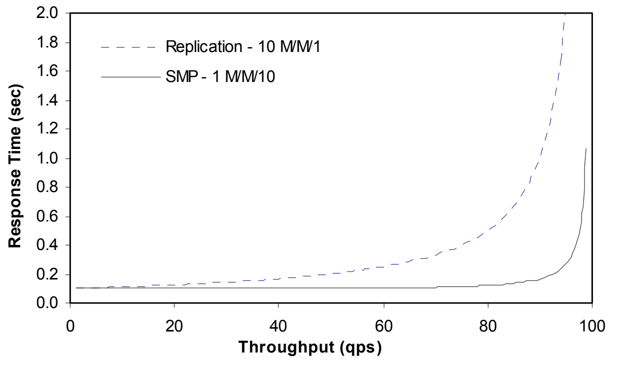

Figure 7: SMP vs. Replication - For a varying throughput

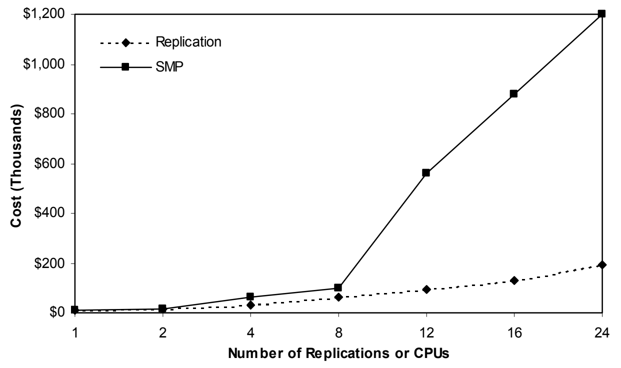

Figure 8: Hardware Cost of SMP vs. Replication

For example, if a node's service time is 100ms and the throughput of the system is 5qps, our utilization would be 50%.In practice, operational departments establish utilization constraints to avoid overtaxing a system. While no hard and fast rules determine what these acceptable constraints are, we cannot ignore utilization in an operational strategy for a search system. Thus in the following sections, utilization is a parameter we use in determining if a given architecture can comfortably meet our throughput and response time requirements.

5.2 Mixture Model

Given operational requirements for response time, throughput, and utilization, we now want to design a strategy for scaling a service to meet all three. Our first strategy used index partitioning to satisfy a response time requirement for a single query. Our second strategy used index replication to satisfy a throughput requirement. In the last section, we pointed out that, while an architecture may meet either of these requirements, it may still have problems if its resources are being overutilized. Thus, we have a need to combine all of these strategies into a single, mixture model framework for building an operational search system.

First, let us examine a strategy of just doing each of the two prior strategies in order:

1. Partition the index such that our response time goals are satisfied for a single query.
2. Replicate each set of partitions to satisfy the throughput requirement.

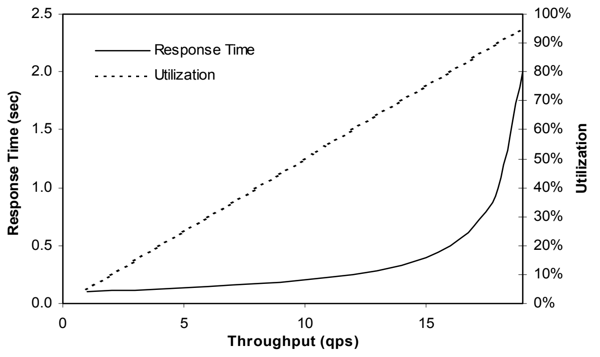

Figure 9: Operational Characteristics for Simple Mixture Model Strategy

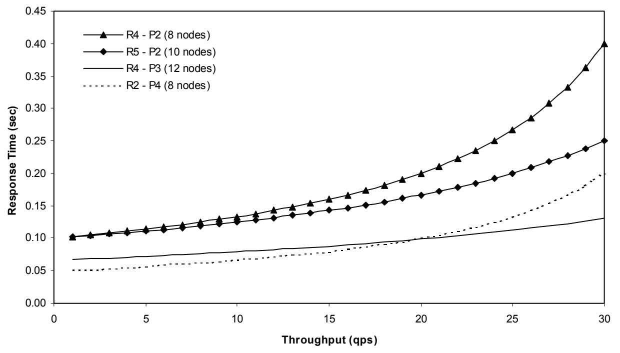

Figure 10: Comparing Mixture Model Strategies

For example, say our response time requirement is .1s and that we have partitioned the index to produce a system with a response time of .1s, satisfying the requirement. Say the throughput requirement is 20qps. At .1s, a single set of partitioned indices can maximally support a throughput about 10qps (1/.1s), therefore we need two replications to match the 10qps requirement.

We now use queuing theory to examine the strategy. As before, each replication is represented as an M/M/1 node with a visit ratio of .5. Figure 9 shows the results of the strategy. Note that the response time is the expected .1s when the throughput is low; however, while the architecture can service 20qps, the response time at that throughput has increased to two seconds, and the utilization of the bottleneck node is 100%. This simple strategy did not produce a viable architecture.

Unfortunately, there is no straightforward strategy to determine an ideal architecture. Any strategy is a balance of operational requirements and practical constraints such as hardware costs and maintainability. The best strategy is simply to employ the operational characteristics as a framework for evaluating architectures in combination with practical constraints.

As an illustration of this process, consider again the previous example with operational requirements of .1s response time and 20qps throughput. We will also set a maximum utilization of 50% for any node in the system. Finally, as a practical matter we want to minimize the hardware costs of the architecture. As a simple metric, we equate hardware cost with the number of nodes (CPUs) in the architecture. Therefore, our goal is to satisfy the three operational requirements with the fewest number of nodes possible.

Figure 10 compares the results of four potential architectures. In the first architecture, (R4, P2), the index is partitioned in half, as in the prior example, and then replicated four times. At 20qps the system has a utilization of 50%, which satisfies the requirements, and a response time of ~196ms, which exceeds the requirements. This architecture has 8 nodes (4*2).

In the second architecture, (R5, P2), the index is again partitioned into half, but replicated five times (versus four times). At 20qps the system has a utilization of 40%, which meets our requirement and is an improvement over the first architecture. However, the response time, at ~166ms, still exceeds the requirements. This architecture has 10 nodes (5*2).

In the third architecture, (R4, P3), the index is partitioned into 3 divisions and replicated 4 times. At 20 qps the utilization is down to 33% and the response time is now .99ms, satisfying all of the requirements. This architecture has 12 nodes (4*3). In the fourth architecture, (R2, P4), the index is partitioned into 4 divisions and replicated only twice. At 20qps the utilization is 50% and the response time is ~100ms, both of which just meet the requirements.

Of the four architectures, only the third and forth satisfy all three operational requirements. The fourth architecture is the cheapest, at 8 nodes versus 12, and is therefore the system of choice for this example. As an additional practical constraint, we can also ask which of the two systems would best scale if the arrival throughput were to grow. According to Figure 10, up until 20qps the 8-node system outperforms the 12-node system in terms of response time; at 20qps the two systems are roughly equivalent; but past 20qps the 12-node system begins to outperform the 8-node system. So, if the potential for increased traffic is significant, the 12-node system might be worth the higher cost.

5.3 Cost-Based Partitioning & Replication

In prior sections we presented operational considerations and tradeoffs that are evaluated when developing a search system. Given that multiple factors must always be examined when finding an optimal architecture, we now present a cost-based approach to finding a set of ideal architectures. While no closed form solutions exist for finding the optimal architecture, this cost-based solution allows us to weigh the various tradeoffs and produce a set of "good" solutions.

Given operational requirements of throughput, response time, and utilization our goal is to minimize the necessary hardware. This goal is formalized by minimizing the cost function

$$c = p * r$$

and meeting the operational constraints of

$$t = f_t(p, r, X) < T$$
$$u = f_u(p, r, X) < U$$

where:

- p = number of partitions
- r = number of replications
- t = response time of the proposed architecture
- T = operational response time requirement
- u = utilization of the proposed architecture
- U = operational utilization requirement
- X = operational throughput requirement

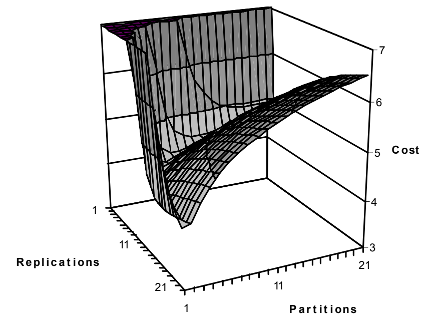

Figure 11: Architectural Tradeoffs Cost Topology

- f_t and f_u calculate the response time and utilization for a given architecture and throughput, as described above.

The cost function represents the hardware cost, which we wish to minimize, by the product of the number of replications and partitions (i.e., the number of needed CPUs). The two operational constraints confirm that the response time and utilization calculated for the given architecture meet the operational requirements.

In order to conveniently combine the operational constraints with the cost function, we can represent the inequalities with exponential functions having soft thresholds. This transformation is shown in Equation 4, where Δ = the width of the transitional window of the exponential function.

$$c' = \log((p * r) * f(t, T, \Delta) * f(u, U, \Delta))$$
$$f(x, x_o, \Delta) = 1 + e^{((x - x_0)/\Delta)}$$

Equation 4: Architecture Cost

This functional representation of the response time has a low value when the response time is below the operational requirement, and a high value when the response time exceeds that requirement (and the same holds for utilization).

If we evaluate this cost function over all possible solutions (i.e., all combinations of replications and partitions), we can determine an optimal set of solutions that minimize the needed hardware and meet all operational requirements. For example, Figure 11 examines a system servicing 12M searches per day, plotting the cost function across possible solutions. The set of optimal solutions corresponds to the trench.

In practice, we can bound the space of all possible solutions: as the number of partitions or replications increases along either axis, once the cost at a given point exceeds its minima in that direction, the search in that direction can be bounded.

Lastly, since the cost-based approach provides a set of good solutions that are functionally equivalent in terms of response time, utilization, and hardware, architects of search systems need to examine which solution most naturally fits in their environment. For example, in some applications the search index cannot be easily partitioned; therefore a solution that favors replication is desirable.

6. SUMMARY AND CONCLUSIONS

Prior research into search system scalability has primarily dealt with either query processing efficiency or presented some arbitrary system architecture. In this paper, we presented a product form queuing analysis technique allowing us to compare search architectures in terms of throughput, response time, and utilization, key metrics for understanding how a system will respond under use. Additionally, we presented a new mixture model strategy for meeting operational requirements by both partitioning and replicating data. Lastly, we introduced a new cost-based analysis model that finds an optimal set of solutions to consider when architecting a search system. Without this formal framework for examining search architectures, we are not able to provide adequate guidance for architectural analysis in terms of operational requirements.

7. REFERENCES

[1] D. Bahle, H. Williams, J. Zobel, "Efficient phrase querying with an auxiliary index", ACM SIGIR, 2002, pp215—221.

[2] F. Scholer, H. Williams, J. Yiannis, J. Zobel, "Compression of inverted indexes For fast query evaluation", ACM SIGIR, 2002, pp. 222—229.

[3] Witten, Moffat, Bell, "Managing Gigabytes: Compressing and Indexing Documents and Images", Second Edition, Morgan Kaufmann, San Francisco, 1999.

[4] S. Brin, L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine". WWW7 / Computer Networks 30(1-7): 107-117 (1998).

[5] A. Arvind, C. Junghoo, G. Hector, P. Andreas, R. Sriram, "Searching the Web", Stanford Technical Report, Dec. 2000, http://dbpubs.stanford.edu/pub/2000-37.

[6] B. Cahoon, K. McKinley, "Evaluating the Performance of Distributed Architectures for Information Retrieval using a Variety of Workloads", ACM Transactions on Information Systems, (1997).

[7] Z. Lu and K. S. McKinley, "Partial Collection Replication for Information Retrieval", UMASS Technical Report, UM-CS-1999-051, August, 1999.

[8] L. Lee, L. Ren, "Document Ranking on Weight-Partitioned Signature Files", ACM Transactions on Information Systems, Vol 14#2, pp 109-137, 1996.

[9] N. Goharian, T. El-Ghazawi, D. Grossman, A. Chowdhury, "On the Enhancements of a Sparse Matrix Information Retrieval Approach", PDPTA, Las Vegas, Nevada, 2000.

[10] H. Williams, J. Zobel, P. Anderson, "What's Next? Index Structures for Efficient Phrase Querying", Australasian Database Conference 1999: 141-152.

[11] D. Bahle, H. Williams, J. Zobel, "Compaction Techniques for Nextword Indexes", SPIRE 2001: 33-45.

[12] D. Bahle, H. Williams, J. Zobel, "Efficient phrase querying with an auxiliary index", SIGIR 2002: 215-221.

[13] J. Zobel, H. Williams, "Compact In-Memory Models for Compression of Large Text Databases". SPIRE/CRIWG 1999: 224-231.

[14] H. Williams, J. Zobel, "Compressing Integers for Fast File Access". The Computer Journal 42(3): 193-201 (1999).

[15] F. Scholer, H. Williams, J. Yiannis, J. Zobel, "Compression of inverted indexes for fast query evaluation". SIGIR 2002: 222-229.

[16] C. Stanfill, "Partitioned posting files: A parallel inverted file structure for information retrieval", SIGIR 1989.

[17] O. Frieder, H. Siegelmann, "On the Allocation of Documents in Multiprocessor Information Retrieval Systems". SIGIR 1991: 230-239.

[18] I. Macleod, T. Martin, B. Nordin, J. Phillips, "Strategies for Building Distributed Information Retrieval Systems", INFORMATION PROCESSING & MANAGEMENT, Vol. 23, No. 6, pp. 511-528, 1987.

[19] P. Danzig, J. Ahn, J. Noll, K. Obraczka, "Distributed indexing: a scalable mechanism for distributed information retrieval", SIGIR 1991, 220-229.

[20] D. Harman, W. McCoy, R. Toense, G. Candela. "Prototyping a distributed information retrieval system using statistical ranking". Information Processing & Management, 27(5):449--460, 1991.

[21] A. Tomasic, H. Garcia-Molina, "Performance of Inverted Indices in Distributed Text Document Retrieval Systems", Parallel and Distributed Information Systems, pp 8-17, 1993.

[22] A. Moffat, J. Zobel: "Information Retrieval Systems for Large Document Collections". TREC 1994.

[23] L. Gravano, H. Garcia-Molina, A. Tomasic, "The Effectiveness of GlOSS for the Text Database Discovery Problem", SIGMOD, pp 126-137, 1994.

[24] L. Gravano, H. Garcia-Molina, "Generalizing {GlOSS} To Vector-Space Databases and Broker Hierarchies", VLDB, pp 78-89, 1995.

[25] L. Gravano, H. Garcia-Molina, a. Tomasic, "GlOSS: text-source discovery over the Internet", ACM Transactions on Database Systems, V24#2, pp 229-264, 1999.

[26] James P. Callan, Zhihong Lu, W. Bruce Croft, "Searching Distributed Collections with Inference Networks". SIGIR 1995: 21-28

[27] T. Couvreur, R. Benzel, S. Miller, D. Zeitler, D. Lee, M. Singhai, N. Shivaratri, W. Wong, "An analysis of performance and cost factors in searching large text databases using parallel search systems". Journal of the American Society for Information Science, 7(45):443--464. (1994).

[28] A. Bestavros, "Demand-based document dissemination to reduce traffic and balance load in distributed information systems". SPDP'95, pages 338-345, San Anotonio, Texas, October 1995.

[29] C. Viles, J. French, "Dissemination of Collection Wide Information in a Distributed Information Retrieval System", SIGIR 18, pp 12-22, 1995.

[30] C. Viles. "Maintaining retrieval effectiveness in distributed, dynamic information retrieval systems". PhD thesis, University of Virginia, May 1996.

[31] Peter Bailey and David Hawking, A Parallel Architecture for Query Processing Over a Terabyte of Text, Department of Computer Science, Technical Report TR-CS-96-04, (June 1996).

[32] B. Cahoon, K. McKinley: "Performance Evaluation of a Distributed Architecture for Information Retrieval". SIGIR 1996: 110-118

[33] B. Cahoon K. McKinley, "Evaluating the Performance of Distributed Architectures for Information Retrieval using a Variety of Workloads", ACM Transactions on Information Systems, (1997).

[34] B. Ribeiro-Neto, R. Barbosa, "Query performance for tightly coupled distributed digital libraries". ACM Conference on Digital Libraries, pages 182-190, June 1998.

[35] O. De Kretser, Moffat, Shimmin, Zobel, "Methodologies for Distributed Information Retrieval". International Conference on Distributed Computing Systems, Amsterdam, May 1998, pp 66-73.

[36] Z. Lu, K. McKinley, B. Cahoon, "The Hardware/Software Balancing Act for Information Retrieval on Symmetric Multiprocessors". Euro-Par 1998: 521-527.

[37] Z. Lu, K. McKinley, "Partial Replica Selection Based on Relevance for Information Retrieval". SIGIR 1999: 97-104

[38] Z. Lu, K. McKinley, "Partial collection replication versus caching for information retrieval systems", Research and Development in Information Retrieval, pages 248-255, 2000.

[39] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling," Wiley- Interscience, New York, NY, April 1991.

[40] R. Suri, "Robustness of Queuing Network Formulas.", JACM 30(3): 564-594 (1983).

[41] Y. Tay, R. Suri, "Error Bounds for Performance Prediction in Queuing Networks.", TOCS 3(3): 227-254 (1985).

[42] Z. Liu, P. Nain, "Sensitivity results in open, closed and mixed product form queueing networks", Performance Evaluation, V13, #4 (1991).