

Processing Search Queries in a Distributed Environment

Frederick Knabe Daniel Tunkelang

Endeca Technologies

Cambridge, MA

+1 617 577 7999

{knabe,dt}@endeca.com

ABSTRACT

Endeca's approach to processing search queries in a distributed computing environment is predicated on concerns of correctness, scalability, and flexibility in deployment. Using a master-slave architecture, we are able to support classic search as well as more advanced features. We avoid bottlenecking the master with excessive computation or communication by limiting the information from the slaves in the expected case.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design.

Keywords

Information Retrieval, Search, Distributed Systems.

1. INTRODUCTION

To successfully address the needs of information retrieval applications, search engines today combine a variety of capabilities, ranging from sophisticated query transformation to composite relevance ranking to results summarization and analysis. In addition to providing such capabilities, search engines are also expected to scale readily, and domains with tens or hundreds of millions of documents are not uncommon.

In large search applications, a natural scaling strategy is to partition the corpus across multiple servers and to distribute the runtime computation of search results accordingly. As data sizes grow, processing and storage demands quickly outstrip the capabilities of a single server, and only a distributed architecture can provide the requisite performance.

Unsurprisingly, distribution also makes implementation more challenging. With the corpus divided between multiple servers, new algorithmic approaches are necessary for features that interpret queries in the context of the entire corpus or that require a unified view of results. For example, relevance ranking must apply across the entire result set, regardless of how that set is

fragmented across servers. Similarly, if a query is subject to data-driven spell correction, the correction must be uniformly applied to all query processors.

Significant constraints must be observed in solving these problems. First, the user should not be able to detect any artifact of how the corpus and computation are distributed. The results returned by any distributed arrangement must be identical to the results that would be returned if the entire corpus could reside on a single server. In particular, the results must be correct and consistent with respect to sorting, relevance ranking, and other advanced search features such as data-driven spelling correction and scripting.

Second, to properly address scalability, the demands placed on any single server must not grow as a function of the corpus size. For example, a single server generally cannot store the entire result set for a query, since the size of the result set may be proportional to the size of the corpus and thus exceed the server's capacity.

Finally, the system should still permit flexible administration choices. For example, incremental growth can lead to mixed-hardware deployments, where new machines with greater processing capabilities are combined with older machines. Fault tolerance can be structured with different trade-offs between cost, fault robustness, and recovery time. As far as possible, our solutions to problems such as distributed relevance ranking should be independent of such administrative decisions.

In this paper we describe Endeca's approach to processing search queries in distributed environments. We first outline our general architecture for distribution. We then explain how we divide the corpus among multiple servers and perform queries against them. We also consider several other search features that must be specially addressed in a distributed setting.

2. ARCHITECTURE

The constraints presented above motivate a distributed architecture in which a central, master server communicates efficiently with a collection of slave servers. The slave servers, among which we partition the corpus, are responsible for computing partial results, while the master server is responsible for combining them.

Both the slave servers and the master server may be replicated for fault tolerance and performance. All servers are sessionless, and any two masters or equivalent slaves (i.e., containing the same segment of the corpus) can be interchanged at any point. These design points offer considerable flexibility to administrators.

We assume that a user never expects to see more than a single page of results at a time—or that, if needed, results can be streamed to a user one page at a time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'04, November 8-13, 2004, Washington, DC, USA.

Copyright 2004 ACM 1-58113-874-1/04/0011...\$5.00.

Ideally, we process a user's search query as follows: the master server receives the query, passes it on to the slave servers, retrieves their partial results, and then combines them to produce the result it returns to the user [1]. However, the master server may learn from the slave servers' responses that it needs to resolve conflicts or compensate for the distribution of the results among the slave servers. Computing the results for a single query may therefore require multiple rounds.

In general, query processing proceeds as follows: the master server translates the user's query Q_0 into a more conservative query Q_1 that retrieves additional information to compensate for the anticipated non-uniformity of results distribution across the servers. The master server passes Q_1 on to the slave servers and aggregates the responses they compute in parallel. The master server then determines whether it has enough information to provide a correct response to the original query Q_0 . If not, it iteratively creates increasingly more conservative queries Q_2, Q_3, \dots until it is able to correctly respond to Q_0 .

The translation of Q_0 into Q_1, Q_2, \dots depends on the corpus distribution and the particular search features being used. We discuss some of these details in the next sections.

3. CORPUS DISTRIBUTION

In partitioning the corpus across multiple servers, we can take one of two approaches. The first is to partition the corpus systematically based on some knowledge of corpus structure or query distribution. The second is to partition the corpus randomly. We have found that uniformly random partitioning is not only more generally applicable, but also has desirable consequences.

There can be advantages to a systematic partitioning approach. For example, it may be the case that 80% of the queries access only the most recent 20% of the data. In such a case, we could partition the servers into "hot" servers for the recent data and "cold" ones for the older data. Throughput scaling of the system can then be performed more cost effectively, by adding proportionally more hot servers than cold ones.

Without assuming any knowledge about the corpus structure or the query distribution, we can nonetheless obtain efficient performance by partitioning the corpus using a uniform random distribution (if the servers vary in capacity, the distribution can be weighted accordingly). A uniform random distribution of the corpus ensures that, for any query, the expected number of results from each server will be the same.

The algorithm for aggregating partial results does, however, have to account for the possibility that the results for a particular query may be skewed.

For example, consider a query to retrieve a page containing results 50,001 to 50,100 out of 100,000 (sorted by a user-specified criterion) from a corpus partitioned across two slave servers. If the results were partitioned perfectly uniformly, there would be exactly 50,000 results in each slave server's partial results, and for each we would retrieve results 25,001 to 25,050. The probability of such a perfect split is, of course, practically nil. Instead, our Q_1 might retrieve a total of 200 results from the slave servers—if the results were split evenly between them, we would retrieve results 24,076 to 25,075 from each server. We might still not be able to construct our desired page from these partial re-

sults, in which case we could try a Q_2 that might retrieve a total of 400 results from the slave servers, and so forth.

In a pathologically skewed distribution, where all of the results from the first server preceded all of the results from the second server, many iterations would be necessary to retrieve the middle page. The probability of such an event, however, is infinitesimal if we distribute the corpus according to a uniform random distribution.

4. QUERY RESULT SETS

The precise tuning of the Q_i s in the query iteration strategy described above depends on the relative costs of computation, network latency, and network bandwidth. For example, the higher the cost of a network roundtrip, the more conservative we make Q_1 to avoid incurring it more than once.

More specifically, assume that a corpus consisting of n documents is uniformly distributed over k servers. While a query to this corpus may match many documents, only a single page will be returned with a small window on those results. If the user's query specifies a window of results *start* through *end*, then each slave server is directed to retrieve results $start/k - R$ through $end/k + R$, where R (the "radius") is a constant.

Upon receiving the results from all slaves,¹ the master determines their overlap. The overlap consists of the results that fall in the interval $max(min_1, min_2, \dots, min_k)$ through $min(max_1, max_2, \dots, max_k)$, where min_i corresponds to the result numbered $start/k - R$ returned from slave i and max_i corresponds to the result numbered $end/k + R$.

Given this information, it is straightforward to compute the global starting and ending indices of the overlap. If there is no overlap or if it does not span *start* through *end*, then the radius must be increased and Q_2 must be issued to the slave servers.

Given the uniform random distribution of documents, a probabilities analysis argues that R need only be proportional to the square root of the window size $end - start$ and also dependent on k . In practice, the number of results returned on a single page is small (no more than 50 in most search applications). Because the roundtrip time of a second query to the slaves so dominates the marginal cost of returning a few extra results, it is suitable to use default values for R comparable to the requested window size; a value of 100 works well. Furthermore, rather than increasing the radius in linear multiples ($2R, 3R, \dots$), we increase it exponentially by powers of 2. These adjustments make query iterations very rare.

In a sessionless architecture, caching of query results on each slave can further reduce the cost of iterated queries: the repeated query can reuse the intermediate results from the prior one, and simply return additional information or a different window of results. This win is partially negated if administrators choose to implement redundancy and load scaling by placing a load balancer between the master and the slaves, since a different slave may service the iterated query. However, given the low incidence of iterated queries, this is not a significant problem.

¹ In some applications, an acceptable fault tolerance strategy is for the master to proceed even if only a subset of slaves are able to respond with results.

5. ADVANCED SEARCH FEATURES

Various advanced search features rely on global information about the corpus [2]. These include data-driven spelling correction, relevance ranking, and scripted search. We consider how these are affected by distribution next.

5.1 Spelling Correction

Our data-driven algorithm for scoring and ranking spelling suggestions takes into account both the term frequencies and the number of results matching all of the terms in the suggestion. It also excludes corrections to terms that do not occur within the subset of documents the user is authorized to search—these subsets are defined by arbitrary Boolean expressions. Because of these dynamic data dependencies, the master is unable to generate spelling suggestions on its own. Instead, the slave servers perform this task. However, because the slaves contain different segments of the corpus, their top-ranked suggestions may differ.

The master receives the suggestions from each slave along with the criteria used to rank them. Because of the uniform data distribution, the suggestions normally agree. If they do not, the master combines the ranking criteria to make an overall top choice, then requeries the slave servers with the choice enforced. Requerying is necessary because the master returns global information about the scoring of spelling suggestions, thus every slave must return information about the same suggestions. Furthermore, our search system can be configured to automatically apply spelling corrections, rather than simply returning “did you mean” suggestions to the user. In this configuration, the slaves must return results that are consistent, with each slave using the same modified query.

In a highly skewed data distribution, it would be possible for the master to incorrectly choose the best spelling suggestions overall because it only considers the top choice from each slave, rather than factoring in information on lower-ranked choices. This scenario has very low probability, however.

5.2 Relevance Ranking

In general, relevance ranking in a distributed environment uses the same machinery as any sorted result set, and the query iteration strategy described previously applies unchanged. When combining results from slaves, the master uses the rank score of each result to order the results and determine the final result page. The only requirement is that ranks be absolutely comparable. Many ranking algorithms, such as those measuring term frequency, aggressiveness of query expansion, phrase matching, and proximity, can meet this requirement.

Some measures of relevance do depend on global information about the query term distribution. For example, measuring the relative information content of query terms can be approximated

based on a partial view of the corpus, but only computed exactly by aggregating information at the master server.

5.3 Scripted Search

Scripted search (e.g., match all query terms if possible, or else match any query term) can generally be implemented in two ways: either the master server can execute the script, or the slave servers can execute the script with the master resolving their results. The former approach, while conceptually simple, is generally less efficient because it may require multiple queries to the slaves.

In our approach, each slave server applies the script individually and reports to the master which matching criterion it actually applied. In the case of “match all terms if possible, or else any,” the master can toss out the “match any” results if any slave returns enough “match all” results to populate a page. Similarly, for the strategy “match all n query terms if possible, or else $n-1$ if possible, or else $n-2$ if possible, etc.,” the master simply retains the results with the highest number of matches, returning them according to their ranks.

6. SUMMARY

Endeca’s approach to distributed processing of search queries is based on requirements of correctness, scalability, and administrative flexibility. Correctness ensures that scale and performance considerations are not allowed to affect the system’s functional behavior. The scalability constraint ensures that we are not creating a scale bottleneck at the master server. Meanwhile, administrative flexibility demands that the solution be practical to deploy, with a range of possibilities for handling machine faults and incremental growth in load and corpus size. Our strategy of uniformly distributing the corpus allows us to pursue a general query-processing algorithm that balances the various costs of computation and communication.

7. ACKNOWLEDGMENTS

Our thanks to all our colleagues at Endeca for their contributions to the design and implementation of the approach described here.

8. REFERENCES

- [1] De Krester, O., Moffat, A., Shimmin, T., and Zobel, J. Methodologies for distributed information retrieval. *Proceedings of the 18th International Conference on Distributed Computing Systems* (May 1998), 26–29.
- [2] Viles, C. and French, J. Dissemination of collection wide information in a distributed information retrieval system. *Proceedings of the 18th ACM Conference on Information Retrieval* (July 1995), 12–20.