A Global Search Architecture

F. J. Burkowski G. V. Cormack C. L. A. Clarke R. C. Good

Department of Computer Science
University of Waterloo, Waterloo, Canada
email: mt@plg.uwaterloo.ca
WWW: ftp://plg.uwaterloo.ca/pub/mt

Technical Report CS-95-12
March 15, 1995

Abstract

Recent advances in communication and storage technology make available vast quantities of on-line information. But this information is of limited use unless it can be searched effectively. Huge scale and heterogeneity of data raise a unique combination of architectural issues that must be addressed to support effective search. These issues occasion the use of multi-user distributed search databases with the following capabilities: efficient structured searching of the contents of files having various schema; continuous availability in spite of failures and maintenance; high-throughput incorporation of a continuous stream of updates, especially the arrival new data and removal of obsolete data. We present an architecture that embodies solutions to specific technical problems that arise in the provision of these capabilities.

The global data abstraction

Until recently, the availability of useful information was constrained by the capacity of physical storage and communication media. While this capacity has ceased to be a constraint the availability of *useful* information is limited by our ability to find it [8]. A number of architectural principles arise in addressing the problem of finding useful information when we assume that everyone has the capacity to access all the world's data at all times.

Multiple users. It is not reasonable to assume that each user has a copy of all useful information. No matter how vast the capacity of a single user to collect and store information, it will be exceeded by the world's capacity. It may be argued that the user could elect to store only information useful to him or her, but effecting this selection begs the question raised here: how to find useful information.

Multiple servers. It is not reasonable to assume that any server has the capacity to handle all the data. To afford scalability, the data must reside on several servers that collaborate to solve search problems posed by users.

Heterogeneous data. It is not reasonable to expect that the world's data be converted to a common format or organization for the purpose of searching. Data exist in a variety of formats and a variety of repositories. It must be possible to integrate these data and repositories into a common architecture, while harnessing their unique characteristics for search and retrieval.

Navigational and non-navigational structured search. Navigational access such as browsing is a useful technique for exploring data according to a pre-defined organizational structure. Creating, maintaining and navigating this structure imposes a burden on the user and archivist alike. It is not reasonable to assume that it is feasible to create a navigational structure amenable to all users' search requirements. Therefore, it must be possible to find data without resorting to navigation. Nevertheless, it must also be possible to harness whatever structure is available to guide the search.

Continuous availability. The universe constantly acquires new data. It is unlikely that growth in on-line storage and communication capacity will accommodate the addition of all new data unless a comparable quantity of data is deleted. It is necessary that the architecture accommodate this flux without massive reorganization. It is reasonable to assume that platform components undergo a similar flux: at any given time, some part of the system will be subject to failure or maintenance; new components will continuously be added and old components removed. The abstraction of a continuously available universe of data must be preserved — it is not reasonable to require the user to circumnavigate failures or to accommodate to system reorganization.

The abstraction of a global searchable data repository will never be fully realized. However, it suggests design decisions that yield an architecture with wide applicability. The obvious application is to improve our ability to search data on the World-Wide Web. The current Web architecture is heavily navigational, exhibiting a number of the problems cited above. On the other hand, specialized search databases (even those reachable via the Web) do not well support the browsing and heterogeneity of the Web. Other applications include more specialized large applications like bulletin boards, newspapers and archival publications.

In support of the abstraction we have designed an architecture and built the key components of a multi-user multi-server search database system. This paper reports on the overall architecture and three critical components: a new query language supporting non-navigational search of data with heterogeneous organization; a search engine embodying new efficient algorithms for searching text and for on-line update; a coded redundancy strategy that maintains availability in the face of failures and maintenance.

The global search architecture

Data model

We assume that the data to be searched is decomposed into files of various sizes and schema. Each file must be uniquely identified within the universe; a hierarchical naming scheme like the Uniform Resource Locator, while not ideal, is sufficient for this discussion. We project each file onto a linear stream of tokens that represent the searchable content and structure of the file. While each token has a textual mnemonic representation, we do not require that the contents of each file be text. Nor do we require that the tokens in the stream represent elements of the file in any straightforward correspondence.

In the simplest case, a file may be ASCII text and the stream of tokens a list of all the words or word stems in order of occurrence. The token stream may instead be an abstract or list of keywords in the file. A file may be text with internal markup like SGML, the internal format of a word processor, or a display language like Postscript. A file may be compressed, or it may not represent text at all — many kinds of data may be displayed sensibly as a linear sequence of tokens. The token stream need not be derived entirely from the contents of a file. The file's name, type, update time, and other environmental information might be used to determine, alter or augment the token stream. For example, annotations or markup may be stored in a separate file or derived using a program. A machine-language program might be disassembled to a stream of opcodes and operands with mnemonic representations derived an using external symbol table. A table in a relational database might be represented in row-major order with annotations to indicate the row and column boundaries. Entire file systems can be represented by traversing them in some canonical order, with sufficient annotation to recover the structure of the file system.

Our global data model treats the word's data as belonging to a huge global file system, projected onto a stream of tokens as described above. A *search* takes the form of a query that is satisfied by zero or more subsequences in the token stream. A *retrieval* is a request to fetch the file (or file fragment or set of files) containing a particular subsequence.

Search model

A search must identify subsequences of tokens that meet specific criteria specified by the user. It is commonly assumed that a logical predicate or objective function is sufficient to specify a search. This is not the case. A predicate or objective function can be used only to determine which members of some *a priori* set (universe) meet or optimize specific criteria. For the data model used here, there is no suitable universe.

One might treat each file as a member of the universe. An elementary predicate might involve testing the presence of a particular token in the file, and expressions might be built using the boolean connectives *and*, *or* and *not*. This *boolean model* for information retrieval is inadequate to take advantage of the intra- and inter-file markup that represents the structural relationships among the data.

Instead of a predicate, one might state an objective function and recast the search as an optimization problem. This is the *relevance ranking model* [5]. Commonly, the universe and the elementary terms in the objective function are the same as those for the boolean model (the presence or number of occurrences of tokens in a file), and so this approach is no more adept than the boolean model in harnessing structural information.

Using the same universe of files, one might use a regular expression or other formal language as a predicate. The result of a search would then be the set of files whose token sequence was a member of the language. This approach can harness intra-file but not inter-file markup. Retrievals are restricted exactly to members of the universe; more fine-grained selection is impossible, as is the selection as a unit of a group of files. Also, common formal language notations, such as ordinary regular expressions, are not amenable as search notations.

To harness the structural relations among files and components of files, it is necessary to abandon the assumption that the universe for search is the set of files: The division of data into files is arbitrary, and any particular choice will necessarily be too coarse for some purposes and too fine for others; hierarchical and other structural relationships cannot be captured by any simple set. Instead, we allow any search to specify any subsequence of the global data. Care must be taken, however, to limit the number of subsequences that are deemed to meet the user's search criteria:

The number of solutions to a query could be quadratic in the size of data. A simple constraint imposed on all queries guarantees linearity; namely that a sequence satisfies a query only if it does not properly contain a subsequence that also satisfies the query. We call this approach the *shortest substring model.*

Query language

The query language GCL is used to specify search criteria under the shortest substring model. A GCL query specifies a set of valid sequences. The validity of a sequence depends on the contents of the sequence (like a formal language) and on the context in which it occurs (unlike a formal language). An elementary term in GCL is simply a token, specifying all single-element subsequences consisting of exactly that token. GCL has three types of operators: *combination*, *ordering* and *containment.*

The combination operators are a generalization of boolean *and* and *or* under the shortest substring model. A query of the form

`all of` (*list of queries*)

specifies every subsequence that contains a solution to every element in the list of queries. A query of the form

`one of` (*list of queries*)

specifies every subsequence that is a solution to one of the queries in the list. (We say "is a solution" rather than "contains a solution" because the shortest substring model precludes a solution from properly containing a solution to the same query.) A query of the form

n `of` (*list of queries*)

is a generalization of the two; it specifies every subsequence that contains solutions to *n* of the queries in the list. It is important to note that none of the combination operators assume any *a priori* structure: They may be satisfied by by the sequence of tokens representing a file, a line, a row, a file fragment, a collection of files, or any other substructure represented by markup.

The ordering operator

query1 `...` *query2*

specifies every subsequence that begins with a solution to one query and ends with a non-overlapping solution to another. It is used to identify regions of text bracketed by words, markup or patterns specified by more complex queries. While some retrieval systems based on the boolean model include queries based on word position, the ordering operator is more general because the bracketing queries can be non-elementary terms, and because the ordering operator can be used in combination with the containment operators.

There are four containment operators:

query1 `containing` *query2*
query1 `not containing` *query2*
query1 `contained in` *query2*
query1 `not contained in` *query2*

Each containment operator specifies every subsequence satisfying the first query such that the subsequence contains, does not contain, is contained in, or is not contained in a subsequence satisfying the second query. Thus queries like

find the titles of documents authored by Burkowski and not referencing Cormack

are easily expressed, provided that documents, titles, authors and references are identified by suitable markup.

Further details of our data and search model, the formal query algebra on which GCL is based, and a complete description of GCL itself is available elsewhere [4, 3].

Implementation framework

GCL has a straightforward implementation for two common data representations: linear and inverted. The implementation based on a linear token stream yields a utility like `grep`, which has running time proportional to the length of the token sequence [1]. The implementation based on inverted lists is like that used in many text-retrieval engines, and has running time no greater than the length of the inverted lists for each of the elementary terms in the query [3]. In addition, the implementation can avoid examining infeasible solutions to subterms in the query if the capability exists to skip forward in the index lists. Both implementations are amenable to use in a distributed system: it is possible to divide-and-conquer the token stream, especially if an additional constraint is imposed restricting inter-file solutions to the ordering operator.

Retrieval is a facility separate from and effected after search. Given a subsequence of tokens yielded by search, the retrieval process recovers from the sequence the names of files, records, or other necessary information to fetch the original data that yielded the subsequence. Because each file in the original data is uniquely named, it is a simple matter to maintain reverse-directory information mapping each token to the file from whence it came. It is also a simple matter to maintain with the original data additional sequencing information that allows the particular record or fragment containing the token to be selected. In the global architecture, the search and retrieval engines are quite separate.

The overall architecture consists of a number of search engines (or *index engines*) and retrieval engines (*text servers*) connected by a network (figure 1). Each file in the global data abstraction is handled by at least one search engine and at least one retrieval engine. A search, expressed as a GCL query, may involve several retrieval engines: the query must be dispatched to a search engine for each file that may contain a solution to the query; the results from the various search engines must be marshalled to yield the overall set of search results. To preserve the global data abstraction, the burden of identifying search engines, dispatching queries and marshalling results must not fall on the user. The architecture therefore includes several *marshaller/dispatcher* engines to automate this process.

From the user's point of view, the marshaller/dispatcher appears to be a single search engine. It accepts a GCL query and yields subsequences in the global information space that satisfy the query. The marshaller/dispatcher maintains a list of available search engines, selects the set necessary to solve the query, and dispatches the query to each of them. It receives the various results, combines them and returns them to the user as if they were the result of a single search. The

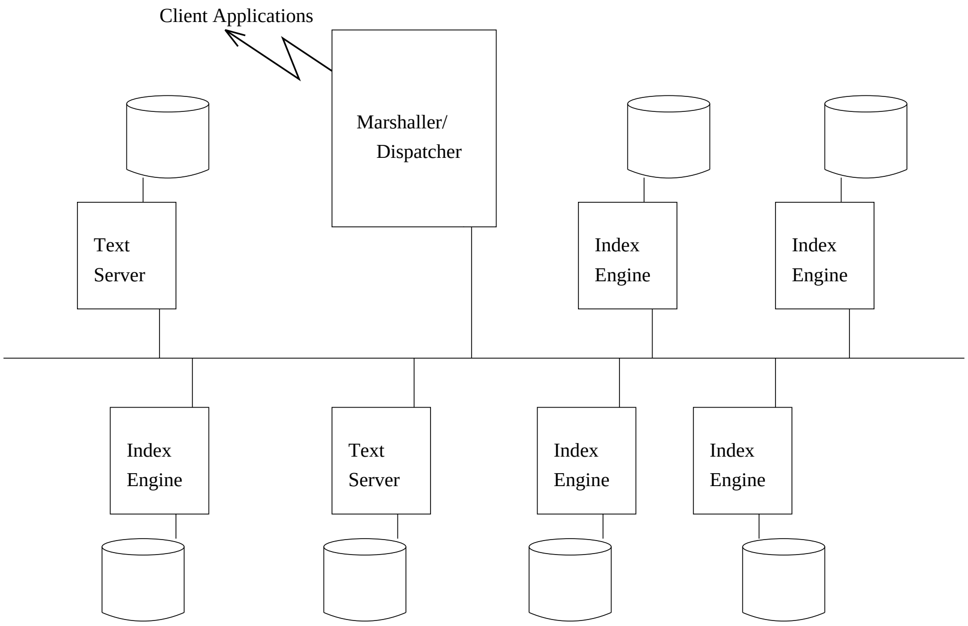

Figure 1: Architecture of the retrieval system.

marshaller/dispatcher may be replicated as necessary to provide capacity adequate to meet demand. Also, marshaller/dispatcher engines may be arranged hierarchically or in layers.

Continuous availability

We wish to maintain continuous availability, uncompromised by component failures, hardware maintenance, software maintenance, data update or data reorganization. Ideally, all data should be continuously available in spite of these occurrences. At least, the global abstraction must be continuously available, even if some data are not. The global search architecture is designed to approach the ideal as closely as available resources permit, while also maintaining the weaker requirement. In this section we outline how the architecture supports a variety of availability mechanisms.

Availability in spite of failures is achieved using redundant data, which may be organized in a number of ways. Data may be replicated on various devices and if one device fails, its data may be found elsewhere. *Mirroring* is the use of redundant search engines with data that duplicate others. If one fails, it is a simple matter to use the mirror instead. Full mirroring involves 100% space overhead, and the issue of how to coordinate access and updates to the various mirrors must be addressed. In the event that a database and its mirror both fail, its contents are lost. *Scattering* involves distributing pieces of the data from one database to many others. When a failure occurs, each of the others assumes responsibility for part of the data. In the event that a database and one of the replicants fails, only part of its contents are lost. The problem of coordinating access and updates to scattered data is somewhat more complex than with mirroring. *Coded redundancy* [6] is an alternative to replication. Using coded redundancy, n units of data may be encoded in m units of storage ($n < m$). Provided n units of storage remain operational, the data can be recovered. Of the m storage units, n store the original data, and so search and retrieval speed are unimpaired so long as these units are operational. However, if one or more of these units fails, recovering its data from the others involves considerable overhead. Also, the problem of coordinating access and updates is more complex than with either of the replication schemes. The architecture supports both replication and coded redundancy — a hybrid of scattering and coded redundancy is particularly attractive because it achieves greater availability than replication alone while incurring only a fraction of the overhead of coded redundancy.

Maintenance can be viewed as the addition and deletion of hardware and software components. Ideally, the system as a whole would automatically adjust to the addition or deletion of these components, resulting in the most efficient and reliable organization. At least, the system must allow components to be added and deleted without necessitating a global shutdown or reorganization of the system. Within this model, data update and reorganization can be viewed as a maintenance activity subject to the same criteria: it must be possible to add or delete and reorganize data while maintaining continuous availability and an efficient and reliable organization. Central to maintenance is the ability to copy or move data from one place to another, while keeping invariant the ability to search it as a unified and consistent whole. The marshaller/dispatcher affords this ability. It must be aware of the location of data, and it must ensure that queries are dispatched that yield all solutions. The marshaller/dispatcher must also eliminate replicated answers, and must ensure that inconsistent or incomplete results, such as those that might arise from querying a partially updated database, are consolidated before being returned to the user.

The problem of maintenance can largely be reduced to that of on-line update. One approach to on-line update is to devise data structures and algorithms that can be updated in real-time. This approach is unlikely to yield high-performance searching, and is likely to mandate periodic data reorganization either because of overflow or because of fragmentation or other gradual degradation of the data structures. Another approach is to apply the updates to a *delta* database and to have the marshaller/dispatcher merge the delta with the master database. Because the delta is small relative to the master, it can be implemented (possibly in RAM) so as to optimize update without significantly compromising overall search performance. From time to time, the delta may be frozen (and a new delta created) and merged with the master to form a new master. This new master is then adopted by the marshaller/dispatcher in place of the old master and old delta. A hybrid of on-line update and old-master/new-master techniques yields a high-performance self-reorganizing search engine [2].

Implementation experience

The MultiText project at the University of Waterloo is working to realize the architecture described in this paper. Each of the components described has been prototyped. Work is proceeding to make some of the components available as a software release and toward integrating the components into an Internet-accessible Network News server as a demonstration of the system as a whole.

`cgrep`

The `cgrep` regular expression search utility makes significant improvements on existing search utilities. Existing search tools limit searches to single lines. The `cgrep` utility allows the search universe itself to be defined using a regular expression. The utility has been used for search and retrieval from repositories of personal email, network news articles, and from text structured with SGML tagging — none of which would be possible with current tools. The results of a search may be tagged to aid further searches.

GCL

An implementation of GCL that is designed to front-end existing search engines has been developed for Internet release. This front end may be used a query language for systems that provide basic indexing of document terms and markup. The back-end interface is extremely simple and supports either sequential indexing or skip-forward indexing. The implementation provides a simple command-line interface and an API to permit the development of a graphical interface.

Text server and index engine

Our TCP/IP-based text server supports the functions of a retrieval engine and our index engine supports the functions of a search server. The servers support multiple search or retrieval clients and permit simultaneous update by an single maintenance client.

RSS

The Robust Storage System (RSS) provides a layer of fault tolerance on which storage for both the search and retrieval servers may be provided. Using the system, a group of disks distributed about a LAN appears either as a smaller group of equally sized disks or as an equally sized group of smaller disks. In either case the storage provided by the disks is more robust. A predetermined number of the disks may fail without apparent effect from the point-of-view of the client, apart from a slight decrease in storage speed.

References

[1] Charles L. A. Clarke and Gordon V. Cormack. On the use of regular expressions for searching text. Technical Report CS-95-07, University of Waterloo Computer Science Department, February 1994.

[2] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Fast inverted indexes with on-line update. Technical Report CS-94-40, University of Waterloo Computer Science Department, November 1994.

[3] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 1995. To appear. An early version of this paper was distributed as University of Waterloo Computer Science Department Technical Report number CS-94-30 [7].

[4] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Schema-independent retrieval from heterogeneous structured text. In *Fourth Annual Symposium on Document Analysis and Information Retrieval*, Las Vegas, Nevada, April 1995. An early version of this paper was distributed as University of Waterloo Computer Science Department Technical Report number CS-94-39 [7].

[5] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval — Data Structures and Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[6] R. C. Good, G. V. Cormack, C. L. A. Clarke, and D. Taylor. A robust storage system architecture. Technical Report CS-95-10, University of Waterloo Computer Science Department, March 1995.

[7] The MultiText Project. Project repository: `ftp://plg.uwaterloo.ca/pub/mt`.

[8] Ian H. Whitten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.