Implementation of a Modern Web Search Engine Cluster

Maxim Lifantsev Tzi-cker Chiueh
Department of Computer Science
Stony Brook University
maxim@cs.sunysb.edu chiueh@cs.sunysb.edu

Abstract

Yuntis is a fully-functional prototype of a complete web search engine with features comparable to those available in commercial-grade search engines. In particular, Yuntis supports page quality scoring based on global web linkage graph, extensively exploits text associated with links, computes pages' keywords and lists of similar pages of good quality, and provides a very flexible query language. This paper reports our experiences in the three-year development process of Yuntis, by presenting its design issues, software architecture, implementation details, and performance measurements.

1 Introduction

Internet-scale web search engines represent crucial web information access tools as well as pose software system design and implementation challenges that involve processing unprecedented volumes of data. To equip these search engines with sophisticated features compounds the overall architectural scale and complexity because this requires integration of non-trivial algorithms that can work efficiently with huge amounts of real-world data.

Yuntis is a prototype implementation of a scalable cluster-based web search engine that provides many modern search engine functionalities such as global linkage-based page scoring and relevance weighting [20], phrase extraction and indexing, and generation of keywords and lists of similar pages for all web pages. The entire Yuntis prototype consists of 167,000 lines of code, and represents a 3-man-year effort. In this paper, we discuss the design and implementation issues involved in the prototyping process of Yuntis. We intend this paper to shed some light into the internal workings of a feature-reach modern web search engine, and serve as a blueprint for future development of Yuntis.

The next two sections provide background and motivation for development of Yuntis, respectively. Section 4 describes its architecture. Implementation of main processing activities of Yuntis is covered in Section 5. Section 6 quantifies the performance of Yuntis. We conclude the paper by discussing future work in Section 7.

2 Background

The basic service offered by all web search engines is returning a set of web page URLs in response to a user query composed of words, thus providing a fast and hopefully accurate navigation service over the unstructured set of (interlinked) pages. To achieve this, a search engine must at least acquire and examine a set of URLs it wishes to provide searching capabilities for. This is usually done by fetching pages from individual web servers starting with some seed set, and then following the encountered links, while obeying some policy that limits and orders the set of examined pages. All fetched pages are preprocessed to later allow efficient answering of queries. This usually involves inverted word indexing, where for each encountered word the engine maintains the set of URLs the word occurs in (is relevant to), possibly along with other (positional) information regarding the individual occurrences of words. These indexes must be kept in a format that allows their fast intersection and merging during querying time, for example, they can be sorted in the same order by the contained URLs.

The contents of the examined pages can be kept so that relevant page fragments or whole pages can be also presented to users quickly. Frequently, some linkage-related indexes are also constructed, for instance, to answer queries about backlinks to a given page. Modern search engines following Google's example [5] can also associate with a page and index some text that is related to or contained in the links pointing to the page. With appropriate selection and weighing of such text fragments, the engine can leverage the page descriptions embedded into its incoming links.

Developing on the ideas of Page, et al [28] and Kleinberg [17], search engines now include some non-trivial methods of estimating the relevance or the "quality" of a web page for a given query using the linkage graph of the web. These methods can significantly improve the quality of search results, as evidenced by the search engine improvements pioneered by Google [11]. Here we consider only the methods for computing query-independent page quality or importance scores based on an iterative computation on the whole known web linkage graph [5, 19, 20, 28].

There are also other less widespread but useful search engine functions, such as the following:

- Query spelling correction utilizing collected word frequencies.
- Understanding that certain words form phrases that can serve as more descriptive items than individual words.
- Determining most descriptive keywords for pages, which can be used for page clustering, classification, or advertisement targeting.
- Automatically clustering search results into different subgroups and appropriately naming them.
- Building high-quality lists of pages similar to a given one, thus allowing users to find out about alternatives to or analogs of a known site.

Several researchers have described their design and implementation experiences building different operations of large-scale web search engines, for example: The architecture of the Mercator web crawler is reported by Heydon and Najork [12]. Brin and Page [5] document many design details of the early Google search engine prototype. Design possibilities and tradeoffs for a repository of web pages are covered by Hirai, et al [13]. Bharat, et al [4] describe their experiences in building a fast web page linkage connectivity server. Different architectures for distributed inverted indexing schemes are discussed by Melnik, et al [24] and Ribeiro-Neto, et al [31].

In contrast, this paper primarily focuses on design and implementation details and considerations of a comprehensive and extensible search engine prototype that implements analogs or derivatives of many individual functions discussed in the mentioned papers, as well as several other features.

3 Motivation

Initially we wanted to experiment with our new model, the voting model [19, 20], for computing various "quality" scores of web pages based on overall linkage structure among web pages in the context of implementing web searching functions. We also planned to implement various extensions of the model that could utilize additional metadata for rating and categorizing web pages, for example, metadata parsed or mined from crawled pages or metadata from external sources such as the directory structure dumps for the Open Directory Project [27].

To do any of this, one needs the whole underlying system for crawling web pages, indexing their contents, doing other manipulations with the derived data, and finally presenting query results in a form appropriate for easy evaluation. The system must also be sufficiently scalable to support experiments with real datasets of considerable size, because page scoring algorithms based on overall linkage in general produce better results when working with more data.

Since there was no reasonably scalable and complete web search engine implementation openly available that one could easily modify, extend, and experiment with, we needed to consider available subcomponents, and then design and build the whole prototype. Existing web search engine implementations were either trade secrets of the company that developed them, systems that were meant to handle small datasets on one workstation, or (non-open) research prototypes designed to experiment with some specific search engine technique.

4 Design of Yuntis

The main design goals of Yuntis were as follows:

- Scalability of data preparation at least to tens of millions of pages processed in a few days.
- Utilization of clusters of workstations for improving scalability.
- Faster development via simple architecture.
- Good extensibility for trying out new information retrieval algorithms and features.
- Query performance and flexibility adequate for quickly evaluating the quality of search results and investigating possible ways for improvement.

We chose C++ as the implementation language for almost all the functionality in Yuntis because it facilitates development without compromising efficiency. To attain decent manageability of the relatively large code-base we adopted the practice of introducing needed abstraction layers to enable aggressive code reuse. Templates, inline functions, multiple inheritance, and virtual functions all provide ways to do this, while still generating efficient code and getting as close to low-level bit manipulation as C when needed. We use abstract classes and inheritance to define interfaces and provide changeable implementations. Template classes are employed to reuse complex tasks and concepts. Although additional abstraction layers sometimes introduce run-time overheads, the reuse benefits were more important for building the prototype.

4.1 High-Level Yuntis Architecture

To maximize utilization of a cluster of PC workstations connected by a LAN, the Yuntis prototype is composed of several interacting processes running on the cluster nodes (see Figure 1). When an instance of the prototype is operational, each cluster node runs one database worker process that is responsible for storing, processing, and retrieving of all data assigned to the disks of that node. When needed, each node can also run one fetcher and one parser process that respectively retrieve and parse web pages that are stored on the corresponding

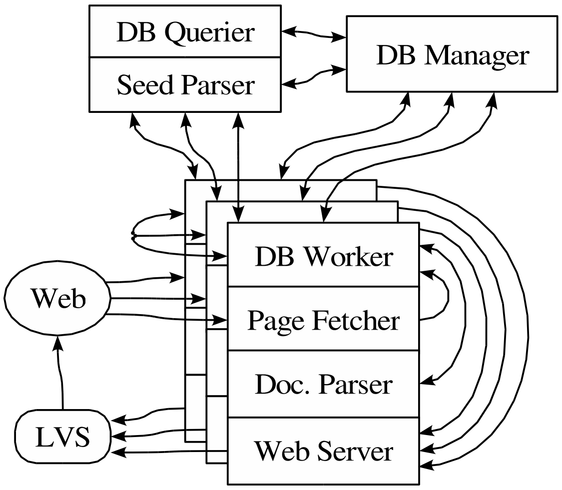

Figure 1: Yuntis cluster processes architecture.

node. There is one database manager process running at all times on one particular node. This process serves as the central control point, keeps track of all other Yuntis processes in the cluster, and helps them to connect to each other directly. Web servers answering user queries are run on several cluster nodes and are joined by the Linux Virtual Server load-balancer [23] into a single service.

There are also a few other auxiliary processes. The database querier helps with low-level manual examination and inspection of all the data managed by the database worker processes. Database rebuilders can initiate rebuilding of all data tables by feeding into the system the essential data from a set of existing data files. A seed data parsing and data dumping process can introduce initial data into the systems and extract some interesting data out of it.

A typical operation scenario of Yuntis involves starting up database manager and workers, importing an initial URL seed set or directory metadata, crawling from the seed URLs using fetchers and parsers, and complete preprocessing of the crawled dataset; then finally we start the web server process(es) answering user search queries. We discuss these stages in more detail in Section 5.

4.2 Library Building Blocks

We made major design decisions early in the development, that would later affect many aspects of the system. These decisions were about choosing the architecture for data storage, manipulation, and querying, as well as the approach to node-to-node cluster data communication. We also decided on the approach to interaction with web servers providing the web pages and with web clients querying the system. These activities capture all main processing in a search engine cluster. In addition, a process model was chosen to integrate all these activities in one distributed system of interacting processes.

The choices about whether to employ or reuse code from an existing library or an application, or rather to implement the needed functionality afresh were made after assessing the suitability of existing code-bases and comparing the expected costs of both choices. Many of these choices were made without a comprehensive performance or architecture compatibility and suitability testing. Our informal evaluation deemed such costly testing not justified by the low expectation of its payoff to reveal a substantially more efficient design choice. For example, existing text or web page indexing libraries such as Isearch [15], ht://Dig [14], Swish [35], or Glimpse [37] were not designed to be a part of a distributed large-scale web search engine, hence the cost of redesigning and reusing them was comparable with writing own code.

4.2.1 Process Model

We needed an architecture that in one process space could simultaneously and efficiently support several of the following: high volumes of communication with other cluster nodes, large amounts of disk I/O, network communication with HTTP servers and clients, as well as significant mixing and exchange of data communicated in these ways. We also wanted to support multiple activities of each kind that individually need to wait for completion of some network, interprocess, or disk I/O.

To achieve this we chose an event-driven programming model that uses one primary thread of control that handles incoming events via a `select`-like data polling loop. We used this model for all processes in our prototype. The model avoids multi-threading overheads of task switching, stack allocation, and synchronization and locking complexities. But it also requires to introduce call/callback interfaces to all potentially blocking operations at all abstraction levels, from file and socket operations to exchanging data with a (remote) database table. Moreover, non-preemptiveness in this model requires us to ensure that processing of large data items can be split into smaller chunks so that the whole process can react to other events during such processing.

The event polling loop can be generalized to support interfaces with the operating system that are more efficient than, but similar to `select`, such as Kqueue [18]. We also later added support for fully asynchronous disk I/O operations via a pool of worker threads communicating through a pipe with the main thread.

Another reason for choosing the essentially unithreaded event-driven architecture were the web server performance studies [16, 29] showing that web servers with such architecture under heavy loads significantly outperform web servers (such as Apache [2]) that al-locate a process or a thread per each request. Hence Apache's code-base was not used as it has different process architecture and is targeted to support highly configurable web servers. Smaller `select`-based web servers such as thttpd [36] were designed to be just fast light-weight web servers without providing a more modular and extensible architecture. In our architecture, communication with HTTP servers and clients is handled by an extensible hierarchy of classes that in the end react to network socket and disk I/O events.

4.2.2 Intra-Cluster Communication

We needed high efficiency of the communication for a specific application architecture instead of overall generality, flexibility, and interoperability with other applications and architectures. Thus we did not use existing network communication frameworks such as CORBA [8], SOAP [33], or DCOM [9] for communication among cluster workstations.

We did not employ network message-passing libraries such as MPI [25] or PVM [30] because they appear to be designed for scientific computing: they are oriented to support many tasks (frequently with multiprocessors in mind) that do not actively use local disks on the cluster workstations and do not communicate actively with many other network hosts. Because of inadequate communication calls, MPI and PVM require to use a lot of threads if one needs intensive communication. They do not have scalable primitives to simultaneously wait for many messages arriving from different points, as well as for readiness of disk I/O and other network I/O, for instance, over HTTP connections.

Consecutively, we developed our own cluster communication primitives. Information Service (IS) is a call/callback interface for a set of possibly remote procedures that can consume and produce small data items or long data streams. The data to be exchanged is untyped byte sequences and procedures are identified by integers. There is also an easy way to wrap this into a standard typed interface. We have implemented support for several IS clients and implementations to set up and communicate over a common TCP socket.

4.2.3 Disk Data Storage

We did not use full-featured database systems mainly because the expected data and processing load required us to employ a distributed system running on a cluster of workstations and use light-weight data management primitives. We needed a data storage system with minimal processing and storage overheads oriented for optimizing the throughput of data-manipulation operations, not latency and atomicity of individual updates. Even high-end commercial databases appeared to not satisfy these requirements completely at the time (May 2000). An indirect support of our choice is the fact that large-scale web search engines also use their own data management libraries for the page indexing data. On the other hand, our current design is quite modular, hence one could easily add database table implementations that could interface with a database management library such as Berkeley DB [3] or a database management system, provided these can be configured to achieve adequate performance.

A set of database manipulation primitives were developed to handle large-scale on-disk data efficiently. At the lowest abstraction level are virtual files that are large continuous growable byte arrays and are used as data containers for database tables. We have several implementations of the virtual file interface based on one or multiple physical files, memory-mapped file(s), or several memory regions. This unified interface allows the same database access code to run over physical files or memory regions.

The database table call/callback interface is at the next abstraction level, and defines a uniform interface to different kinds of database tables that share the same common set of operations: add, delete, read, or update (a part of) a record identified by a key. A database table implementation composed of disjoint subtables together with an interface to an Information Services instance allows a database table to be distributed across multiple cluster nodes while keeping data table's physical placement completely transparent to the code of its clients. To support safe concurrent accesses to a database table, we provide optional exclusive and shared locking at both the database record and database table levels.

At the highest abstraction level are classes and templates to define typed objects that are to be stored in database tables (or exchanged with Information Services), as well as to concisely write procedures that exchange information with database tables or IS'es via their call/callback interfaces. This abstraction level enables us to hide almost all the implementation details of the database tables behind a clean typed interface, at the cost of small additional run-time overheads. For example, we frequently read or write a whole data table record, when we are actually interested in just a few of its fields.

4.3 External Libraries and Tools

We have heavily relied on existing more basic and more compatible libraries and tools than the ones discussed earlier.

The Standard Template Library (STL) [34] of C++ proved to be very useful, but we had to modify it to enhance its memory management functionality by adding real memory deallocation, and eliminate a hash table implementation inefficiency of erasing elements from a large, very sparse table.

GNU Nana library [26] is very convenient for logging and assertion checking during debugging, especially when the GNU debugger (GDB) [10] due to its own bugs often crashes while working with the core dumps generated by our processes. Consequently we had to rely more on logging and on attaching GDB to a running process, which consumes a fair amount of processing resources. Selective execution logging and extensive run-time assertion checking greatly helped in debugging our parallel distributed system.

The eXternalization Template Library [38] approach provides a clean, efficient, and extensible way to convert any typed C++ object into and from a byte sequence for compact transmission among processes on the cluster of workstations, or for long-term storage on disk.

Parallel compilation via the GNU make utility and simple scripts and makefiles, together with right granularity of individual object files, allowed us to reduce build times substantially by utilizing all our cluster nodes for compilation. For example, a full Yuntis build taking 38.9min for compilation and 2min for linking on one workstation takes 3.7+2min on 13 workstations.

4.4 Data Organization

We store information about the following kinds of objects: web hosts, URLs, web sites (which are sets of URLs most probably authored by the same entity), encountered words or phrases, and directory categories. All persistently stored data about these objects is presently organized into 121 different logical data tables. Each data table is split into partitions that are evenly distributed among the cluster nodes. The data tables are split into 60, 1020, 120, 2040, and 60 partitions for the data respectively related to one of the above five kinds of objects. These numbers are chosen as to ensure a manageable size of each partition for all data tables at the targeted size of a manipulated dataset.

All data tables (that is, their partitions) have one of the following structures: indexed array of fixed-sized records, array of fixed-sized records sorted by a field in each record, heap-like addressed set of variable-sized records, or queues of fixed- or variable-sized records. These structures cover all our present needs, but new data table structures can be introduced if needed. Records in all these structures except queues are randomly accessible by small fixed-sized keys. The system-wide keys for whole data tables contain a portion used to choose the partition and the rest of the key is used within the partition to locate a specific record (or a small set of matching records in the case of the sorted array structure).

For each of the above five kinds of web-world objects there are data tables to map between object names and internal identifiers which index fixed-sized information records, which in turn contain pointers into other tables with variable-sized information related to each object. This organization is both easy to work with and allows for a reasonably compact and efficient data representation.

The partition to store a data record is chosen by the hash values derived from the name of the object to which the record is most related. For example, if the hash value of a URL maps it to the ith partition out of 1020, then such items as the URL's name, the URL's information record, lists of back and forward links for the URL are all to be stored in the ith partition of the corresponding data table. One result of such data organization is that a database key or textual name of an object readily determines the database partition and cluster node the object belongs to. Hence, for all data accesses a database client can choose and communicate directly with the right worker without consulting any central lookup service.

4.5 Data Manipulation

The basic form of manipulation over data stored in the data tables is when individual data records or their parts are read or written by a local or remote request and the accessing client activity waits for completion of its request. There are two kinds of inefficiencies we would like to eliminate here: the network latency delay for remote accesses and local data access delays and overheads. The latter occur when the data needed to complete a data access has to be brought into memory from disk and into the CPU cache from memory. This can also involve the substantial processing overheads of working with data via file operations instead of accessing memory regions.

To avoid all these inefficiencies we rely on batched delayed execution of data manipulation operations –see Lifantsev and Chiueh [21] for full details. All large volume data reading (and update when possible) is organized around sequential reading of the data table partition files concurrently on all cluster nodes. In most other cases, when we need to perform a sequence of data accesses that work with remote or out-of-core data, we do not execute the sequence immediately. Instead we batch the needed initiation information into a queue associated with the group of related data table partitions this sequence of data accesses needs to work with. When such batching is done to a remote node, in most cases we do not need an immediate confirmation that the batching has completed in order to continue with our work. Thus most network communication delays are masked. After a large number of such initiation records are batched to a given queue to justify the I/O costs (or when no other processing can proceed), we execute such a batch by loading or mapping into memory the needed data partitions and then working with the data in memory.

For many data tables, we can guarantee that each of their partitions will fit into the available memory, thus they are actually sequentially read from disk. For other data tables, the utilization of file mapping cache in the OS is significantly improved. With this approach, even for limited hardware resources, we can guarantee for a large spectrum of dataset sizes that in most cases all data manipulation happens with data already in local memory (or even CPU cache) via low-overhead memory access primitives. This model of processing utilizes such primitives as the following: support for database tables composed of disjoint partitions, buffered queues over several physical files for fast operation batching, classes to start and arbiter execution of operation batches and individual batched operations, and transparent memory-loading or mapping of selected database table partitions for the time of execution of an operations' batch.

In the end, execution of a batched operation consists of manipulating some database data already in memory and scheduling of other operations by batching their input data to an appropriate queue possibly on other cluster nodes. We wait for completion of this inter-node queueing only at batch boundaries. Hence, inter-node communication delays do not block execution of individual operations. High-level data processing tasks are organized by a controlling algorithm at the database manager process that initiates execution of appropriate operation batches and initial generation of operations. Both of these proceed on cluster nodes in parallel.

4.5.1 Flow Control

During execution of operation batches (and operation generation by database table scanning) we need to have some flow control: On one hand, to increase CPU utilization, many operations should be allowed to execute in parallel in case some of them block on I/O. On the other hand, batch execution (sometimes even execution of a single operation) should be paused and resumed so that inter-cluster communication buffers are not needlessly large when they are being processed. Our adopted solution is to initiate a certain large number of operations in parallel and pause/resume their execution via appropriate checks/callbacks depending on the number of pending inter-cluster requests at this node. Allowing on the order of 100,000 pending inter-cluster requests appears to work fine for all Yuntis workloads. The exact number of operations potentially started in parallel is tuned depending on the nature of processing done by each class of operations and ranges from 20 to 20,000.

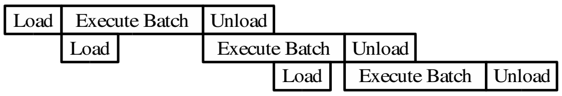

Figure 2: Operation batches execution pipeline.

4.5.2 CPU and I/O Pipeline

Since most data processing is organized into execution of operation batches, we optimize it by scheduling it as a pipeline (see Figure 2). Each batch goes through three consecutive stages: reading/mapping of database partitions from disk, execution of its operations, and writing out of modified database data to disk. The middle stage is more CPU-intensive, while the other two are more I/O-intensive. We use two shared/exclusive locks and an associated sequence of operations with them to achieve pipeline-style exclusive/overlapped execution of CPU and I/O-intensive sections. This requires us to double the number of data partitions so that the data manipulated by two adjacent batches all fits into the available memory of a node.

5 Implementation of Yuntis

In the following sections we describe the implementation details and associated issues for the major processing activities of Yuntis mostly in the order of their execution. Table 1 provides a coarse breakdown for code sizes of major Yuntis subsystems.

5.1 Starting Components Up

First, the database manager process is started up on some cluster node and begins listening on a designated TCP port. After that, the database worker processes are started on all nodes and start listening on another designated TCP port for potential clients, as well as advertise their presence to the manager by connecting to it. As soon as the manager knows that all workers are up, it sends the information about the host and port numbers of all workers to each worker. At this point each worker establishes direct TCP connections with all other workers and reports complete readiness to the manager.

Other processes are connected to the system in a similar fashion. A process first connects to the manager and once the workers are ready is given information about the host and port numbers of all workers. Then the process connects and communicates with each worker directly. Control connections are still maintained between the manager and most other processes. They are in particular used for a clean disconnection and shutdown of the whole system.

5.2 Crawling and Indexing Web Pages

The initial step is to get a set of the web pages and organize all the data into a form ready for later usage.

Subsystem	Code Lines	Code Bytes	Logical Modules
Basic Libraries	51,790	1,635,356	49
Web Libraries	15,286	476,084	24
Info. Services	3,950	107,260	13
Data Storage	16,924	566,721	22
Search Engine	79,322	2,855,390	49
Total	167,272	5,640,811	157

Table 1: Yuntis subsystem code size breakdown.

5.2.1 Acquiring Initial Data

The data parsing process can read a file with a list of seed URLs or the files that contain the XML dump of the directory structure for the Open Directory Project [27] (publicly available online). These files are parsed while read and appropriate actions are initiated on the database workers to inject this data into the system.

Another way of data acquisition is to parse data from the files for a few essential data tables available from another run of the prototype and rebuild all other data in the system. These essential tables are the log of domain name resolution results for all encountered host names, the log of all URL fetching errors, and the data tables containing compressed raw web pages and `robots.txt` files [32]. The rebuilding for each of these tables is done by one parser on each cluster workstation that reads and injects into the workers the portion of the data table stored on its cluster node. We use this rebuilding, for example, to avoid refetching a large set of web pages after we have modified the structure of some data tables in the system.

5.2.2 Fetching Documents from the Web

The first component of crawling is actual fetching of web pages from web servers. This is done by fetcher processes at each cluster workstation. There is a fetch queue data table that can be constructed incrementally to include all newly encountered URLs. Each fetcher reads from a portion of this queue located on its node and attempts to keep retrieving at most 80 documents in parallel while obeying the robots exclusion conventions [32]. The latter involves retrieving, update, and consulting contents of the `robots.txt` files for appropriate hosts. To mask response delays of individual web servers, we wish to fetch many documents in parallel, but too many simultaneous TCP connections from our cluster might muscle out other university traffic. If a document is retrieved successfully, it is compressed and stored in the document database partition local to the cluster node, then an appropriate record is added to the document parsing queue. That is, the URL space and the responsibility for fetching and storing it is split among the cluster nodes, also the split is performed in such a way that all URLs from the same host are assigned to the same cluster node. As a result, in particular to be polite to web servers, the fetcher processes do not need to perform any negotiation with each other and have to communicate solely with local worker processes. The potential downside of this approach is that URLs might get distributed among cluster nodes unevenly. In practice, we saw only 12% deviation from the average of the number of URLs in a node.

5.2.3 Parsing Documents

Document parsing was factored into a separate activity because fetching documents is not the only way of obtaining them. Parsing is performed by the parser processes on the cluster nodes. Parsers dequeue information records from the parse queue, retrieve and decompress the documents, and then parse them and inject the results of parsing into appropriate database workers. Parsers start with the portion of the parse queue (and documents) local to their cluster node, but switch to documents from other nodes when the local queue gets empty. Most of the activities in a parser actually happen in a streaming mode: a parser can communicate to the workers some results of parsing the beginning of a long document, while still reading in the end of the document. We also attempt to initiate parsing of at most 35 documents in parallel on each node so that parsers do not have to wait on document data and remote responses from other workers. A planned optimization is to eliminate the cost of page decompression when parsing recently-fetched pages. Another optimization is to have a small dictionary of the most frequent words in each parser so that for a substantial portion of word indexing we can map word strings to internal identifiers directly in the parsers. Having a full dictionary is not feasible as, for instance, we have collected over 60M words for a 10M pages crawl.

5.2.4 Word and Link Indexing

We currently parse HTML and text documents. Full text of the documents is indexed along with information about prominence of and distances between words that is derived from the HTML and sentence structure. All links are indexed along with the text that is contained in the link anchor, surrounds the link anchor within a small distance but does not belong to other anchors, and the text of two structurally preceding HTML headers. All links from a web page are also weighted by an estimate of their prominence on the page.

As a result of parsing, we batch the data needed to perform actual indexing into appropriate queues on appropriate worker nodes. After all parsing is done (and during parsing when parsing many documents) these indexing batches are executed according to our general data manipulation approach. As a result the following gets constructed: unsorted inverted indexes for all words, forward and backward linkage information, and information records about all newly encountered hosts, URLs, sites, and words.

5.2.5 Quality-Focused Crawling

Breadth-first crawling can be performed using already described components because the link-indexing process already includes an efficient way of collecting all newly encountered URLs for later fetching. The problem with breadth-first crawling is that it is very likely to fall into (unintended) crawler traps, that is, fetch a lot of URLs few people are going to be interested in. For example, even when crawling the `sunysb.edu` domain of our university a crawler can encounter huge documentation or code revision archive mirrors, many pages of which are quickly reachable with breadth-first search.

We thus employ the approach of crawling focused by page quality scores [7]. In our implementation, after fetching a significant new portion of URLs from an existing fetch queue (for instance, a third of the number of URLs fetched earlier), we execute all operations needed to index links from all parsed documents and then compute page quality scores via our voting model [20] for all URLs and the currently known web linkage graph (see Section 5.3.1). After this the fetch queue is rebuilt (and organized into a priority queue) by filling it with yet unfetched URLs with best scores. Then document fetching and parsing continues with the new fetch queue starting with the best URLs according to the score estimates. Thus, we can avoid wasting resources on unimportant pages.

5.2.6 Post-Crawling Data Preprocessing

After we have fetched and parsed some desired set of pages, all indexing operations initiated by parsing must be performed so that we can proceed with further data processing. This involves building lists of backlinks, lists or URLs and sites located on a given host, and URLs belonging to a given site. A site is a collection of web pages assumed to be controlled by a single entity, a person or an organization. Presently any URL is assigned to exactly one site and we treat as sites whole hosts, home pages following the traditional `/~username/` syntax, and home pages located on the few most popular web hosting services. We also merge all hosts with the same domain suffix into one site when they are likely to be under control of a single commercial entity. In order to perform phrase extraction and indexing described later, direct page and link word indexes are also constructed. They are associated with URLs and contain word identifiers for all document words and words associated with links from the document.

5.3 Global Data Processing

After collecting a desired set of web pages, we perform several data processing steps that each work with all collected data of a specific kind.

5.3.1 Computing Page Quality Scores

We use a version of the developed voting model [20] to compute global query-independent quality scores for all known web pages. This model subsumes Google's PageRank approach [19] and provides us with a way to assess importance of a web page and reliability of the information presented on it in terms of different information retrieval tasks. For example, page scores are used to weigh word occurrence counts, so that unimportant pages can not skew the word frequency distribution. Our model uses the notion of a web site for determining the initial power to influence final scores and to properly discount the ability of intra-site links to increase site's scores. As a result a site can not receive a high score just by the virtue of being large or heavily interlinked, which was the case for the public formulation of PageRank [5, 28].

The score computation proceeds in several stages. The main stage is composed of several iterations, each of which consists of propagating score increments over links and collecting them at the destination pages. As with all other large volume data processing, these score computation steps are organized using our efficient batched execution method. Since in our approach later iterations monotonically incur smaller amount of increments to propagate, the number of increments and the number of web pages that are concerned also reduces. To exploit this we rely more on caching and touching only the needed data at the later iterations.

5.3.2 Collecting Statistics

Statistics collection for various data tables happens as a separate step and in parallel on all cluster nodes by sequential scanning of relevant data table partitions, then statistics for different data partitions are joined. The most interesting use of statistics is for estimation of the number of objects that have a certain parameter greater (or lower) than a given value. This is achieved by closely approximating such dependencies using the distribution of the values of such object parameters and the histogram approximations of these distributions. Most types of parameters we are interested in, such as the quality scores for web pages, more or less follow the power-law distribution [1], meaning that most objects have very small values of the parameter that are close to each other and few objects have very high values of the parameter. This knowledge is used when fitting distributions to their approximations and moving the internal boundaries of the collected histograms. As a result in three passes over data we can arrive at a close approximation that does not significantly improve with more passes.

5.3.3 Extracting and Indexing Phrases

To experiment with phrases for instance as keywords of documents we perform phrase extraction and index all extracted phrases. Phrases are simply sequences of words that occur frequently enough. Phrase extraction is done in several stages corresponding to the possible phrase lengths (presently we limit the length to four). Each stage starts with considering forward word/phrase indexes for all documents that constitute top 20% with respect to page quality scores from Section 5.3.1. We thus both reduce processing costs and can not be manipulated by low-score documents. All sequences of two words (or a word and a phrase of length $i-1$) are counted (weighted by document quality scores) as phrase candidates. The candidates with high scores are chosen to become phrases. New phrases are then indexed by checking for all "two-word" sequences in all documents if they are really instance of chosen phrases. Eventually complete forward and inverted phrase indexes are built.

This phrase extraction algorithm is purely statistical: it does not rely on any linguistic knowledge. Yet it extracts many common noun phrases such as "computer science" or "new york", although along with incomplete phrases that involve prepositions and pronouns like "at my". Additional linguistic rules can be easily added.

5.3.4 Filling Page Scores into Indexes

In order to be able to answer user queries quicker, it is beneficial to put page quality scores for all URLs into all inverted indexes and sort the lists of URLs in them by these scores. Consecutively, to retrieve a portion of the intersection or the union of URL lists for several words with highest URL scores (which constitutes the result of a query), we do not have to examine the whole lists.

Page quality scores are filled into the indexes simply by consulting the score values for the appropriate URLs in the URL information records, but this is done via our delayed batched execution framework so that the information records for all URLs do not have to fit into the memory of the cluster. To save space we map four-byte floating point score values to two-byte integers using a mapping derived from approximating the distribution of score values (see Section 5.3.2).

In addition we also fill similar approximated scores into indexes of all words (and phrases) associated with links pointing to URLs. To do this we also keep full URL identifiers of linking URLs in these indexes. This allows us to quickly assess the weight of a all words used to describe a given URL via incoming links. Sorting of various indexes by the approximated page score values is done as a separate data table modification step.

5.3.5 Building Linkage Information

We use a separate stage to construct the data tables about backward URL to URL linkage, forward web site to URL on other sites linkage, and backward URL on other sites to site linkage from forward URL to URL linkage data. At this time we also incorporate page quality scores into these link lists for later use during querying.

5.3.6 Extracting Keywords

We compute characteristic keywords for all encountered pages to be later used for assessing document similarity. They are also aggregated into keyword lists for web sites and ODP [27] categories. All these keyword lists are later served to the user as a part of the information record about an URL, site, or category.

Keywords are words (and phrases) most related to a URL and are constructed as follows: inverted indexes for all words that are not too frequent and not too rare (as determined by hand-tuned bounds) are examined and candidate keywords are attached to the URLs that would receive the highest scores for a query composed of the particular word. For all word-URL pairs, the common score boundary to pass as a candidate keyword is tuned to reduce processing, while yielding enough candidates to choose from. Since both document and link text indexes are considered similarly to query answering, extracted document keywords are heavily influenced by the link text descriptions. Thus, we are sometimes able to get sensible keywords for not fetched documents.

For all URLs the best of candidate keywords are chosen according to the following rules: we do not keep more than 30 to 45 keywords per URL depending on URL's quality score, and we try to discard keywords that have scores smaller by a fixed factor on the log-scale than the best keyword for the URL. We also discard keywords that have a phrase containing them as another candidate keyword of the same URL with a score of the same magnitude on log-scale. The resulting URL keywords are then aggregated into candidate keyword sets for sites and categories, which are then similarly pruned.

5.3.7 Building Directory Data

The Open Directory Project [27] freely provides a classification directory structure similar to Yahoo [39] in size and quality. Any selected part of the ODP's directory structure can be imported and incorporated by Yuntis. Description texts and titles of listed URL's are fully indexed as a special kind of link texts. All the directory structure is fully indexed, so that the user can easily navigate and search the directory or its parts, as well as see in what categories a given URL or a subcategory are mentioned. For some reason the latter feature appears to be unique to Yuntis despite the numerous web sites using ODP data.

One interesting problem we had to resolve was to determine the portion of subcategory relations in the directory graph that can be treated as subset inclusion relations. Ideally we want to treat all subcategory relations this way, but this is impossible since the subcategory graph of ODP is not acyclic in general. We ended up with an efficient iterative heuristic graph-marking algorithm, that likely can be improved, but behaves better in the cases we tried than corresponding methods in ODP itself or Google [11]. Note that all the directory indexing and manipulation algorithms are distributed over the cluster and utilize the delayed batched execution framework.

5.3.8 Finding Similar Pages

Locating web pages that are very similar in topic, service, or purpose to a given (set of) pages (or sites) is a very useful web navigation tool on its own. Algorithms and techniques used for finding similar pages can also be used for such tasks as clustering or classifying web pages.

Yuntis precomputes lists of similar pages for all pages (and sites) with respect to three different criteria: pages that are linked from high-score pages closely with the source page, pages that have many high-scored keywords common with the source page, and pages that link to many of the pages the source page does. The computation is organized around efficient volume processing of all relevant pieces of "similarity evidence". For example, for textual similarity we go from pages to all their keywords, find other pages that have the same word as a keyword, choose the highest of these similarity evidences for each word, and send them to the relevant destination pages. At each destination page all similarity evidences from different keywords for the same source page are combined and some portion of most similar pages is kept. As a result, all processing consumes linear time in the number of known web pages, and the exact amount of processing (and similar pages kept) can be regulated by the values that determine what evidence is good enough to consider (or to qualify for storage).

5.4 Data Compaction and Checkpointing

Data table partitions organized as heaps of variable-sized records usually have many unused gaps in them after being extensively manipulated: Empty space is reserved for fast expected future growth of records. Not all space freed after a record shrinking is later used for another record. To reclaim all this space on disk we introduced a data table compaction stage that removes all the gaps in heap-like table partitions and adjusts all indexes to the records in each such partition that are contained in the associated information partition. The latter is cheap to accomplish as each such information partition easily fits in memory.

All data preparation in Yuntis is organized in stages that roughly correspond to the previous subsections. To alleviate the consequences of hardware and software failures, we introduced data checkpointing after all stages that take considerable time, as well as the option to restart processing from any such checkpoint. Checkpointing is done by synchronizing all data from memory to files on disks and "duplicating" the files by making new hard links. When we later want to modify a file with more than one hard link, we first duplicate its data to preserve the integrity of earlier checkpoints.

5.5 Answering User Queries

User queries are answered by any of the web server processes; they handle HTTP requests, interact with database workers to get the needed data, and then format the results into HTML pages. Query answering for standard queries is organized around sequential reading, intersecting and merging of the beginnings of relevant sorted inverted indexes according to the structure of the query. Then additional information for all candidate and resulting URLs is queried in parallel, so that URLs can be clustered by web sites and URL names and document fragments relevant to the query can be displayed to the user. For flexible data examination, Yuntis supports 13 boolean-like connectives (such as `OR`, `NEAR`, `ANDNOT`, and `THEN`) and 19 types of basic queries that can be freely combined by the connectives. In many cases exact information about intra-document positions is maintained in the indexes and utilized by connectives. An interesting consequence of phrase extraction and indexing it that it can considerably speed up (for example, by a factor of 100) many common queries that (implicitly) include some indexed phrases. In such cases, the work of merging the indexes for the words that form the phrase(s) has been already done during phrase indexing.

6 Performance Evaluation

Below we describe the hardware configuration of our cluster and discuss the measured performance and sizes of the handled datasets.

6.1 Hardware Configuration

Presently Yuntis is installed on a 12-node cluster of Linux PC workstations each running a Red Hat Linux 8.0 with a 2.4.19 kernel. Each system has one AMD Athlon XP 2000 CPU with 512MB of DDR RAM connected by a 2*133MHz bus, as well as two 80GB 7200 RPM Maxtor EIDE disks model 6Y080L0 with average seek time of 9.4msec. Two large partitions on each

Documents Stored	4,065,923
URLs Seen	34,638,326
Hyper Links Seen	87,537,723
Inter-Site Links	18,749,662
Web Sites Recognized	2,833,110
Host Names Seen	3,139,435
Canonical Hosts Seen	2,448,607
Words Seen	30,311,538
Phrases Extracted	574,749
Avg. Words per Document	499.5
Avg. Links per Document	20.4
Avg. Document Size	13.1 KB
Avg. URLs per Site	12.2
Avg. Word Length	9.89 char-s
Total Final Data Size	87,446 MB
Avg. Data per Document	21,039 B
Compressed Documents	13,739 MB
Inverted Doc. Text Indexes	23,188 MB
Inverted Link Text Indexes	11,125 MB
Other Word Data	1,628 MB
Keyword Data	1,922 MB
Page Similarity Data	25,908 MB
All Linkage Indexes	2,861 MB
Other URL Data	4,613 MB
Other Host, Site, and Category Data	2,458 MB
Forward Word Indexes (not in total)	29,528 MB

Table 2: Data size statistics.

two disks are joined by LVM [22] into one 150G ext3 file system for data storage. The nodes are connected into a local network by full-duplex 100Mbps 24-port Compex SRX 2224 switch and 12 network cards. The ample 4.8Gbps back plane capacity of the switch ensured that it will not become the bottleneck of our configuration. The full-duplex 100Mbps connectivity of each cluster node has not yet become a performance bottleneck: So far we have seen sustained traffic of around 7MBps in and 7MBps out for each node out of the potentially available 12.5+12.5MBps. With additional optimizations or increase of CPU power leading to higher communication volume generated by each node, we might have to use a higher-capacity cluster connectivity, for instance, channel bonding with several 100Mbps cards per node. A central management workstation with an NFS server is also connected to the switch, but does not noticeably participate in the workloads of Yuntis, simply providing a central place for logs, configuration files, and sometimes the executables. The cluster is connected to the outside world via the 100Mbps campus network and a 155Mbps OC3 university link.

6.2 Dataset Sizes

Table 2 provides various size statistics. We can for example see that the bulk of the stored data falls on the text indexes, similarity lists, and compressed documents. This data and later performance figures are for a particular crawl of 4 million pages started by fetching 1.3 million URLs listed in the non-international portion of ODP. All these numbers simply provide order-of-magnitude estimates of the typical figures one would get for similar datasets on similar hardware.

We use the bzip2 library [6] for compressing individual web pages. This achieves a compression factor of 3.87. bzip2 is very affective when applied to individual pages: compressing the whole document data table partitions would save only 0.38% of space. We have not yet seriously considered additional compression of other data tables beyond compact data representation with bit fields.

6.3 Data Preparation Performance

As we have demonstrated [21], the batched data-driven approach to data manipulation leads to performance improvements by a factor of 100 via both better CPU and memory file cache utilization. It also makes the performance much less sensitive to the amounts of available memory.

Table 3 provides various performance and utilization figures for different stages of data preprocessing. The second to last column gives the amount of file data duplication needed to maintain the checkpoint for the previous stage. The data shows that phrase extraction, similarity precomputation, and filling of scores into indexes are the most expensive tasks after the basic tasks of parsing and indexing. Various stages exhibit different intensity of disk and network I/O according to their nature, as well as perform differently in terms of CPU utilization. We are planning to instrument the prototype to determine, for each stage, the exact contributions of the possible factors to non-100% CPU utilization, and then improve on the discovered limitations.

Peak overall fetching speed reaches 217 doc-s/sec and 3.6 MB/sec of document data. Peak parsing speed reaches 935 doc-s/sec. Sustained speed of parsing with complete indexing of encountered words and links is 304 doc-s/sec. Observed overall crawling speed when crawling 4M pages with fetch queue rebuilding after getting each 0.6M pages was 67 doc-s/sec. The speed to do all parsing and complete all incurred indexing is 297 doc-s/sec. The speed of all subsequent preprocessing is 90 doc-s/sec. The total data preparation

Processing Stage	Time (min)	CPU Utilization (%)			Disk I/O (MB/s)		Netw. I/O (MB/s)		Total Disk Data (GB)	
		User	Sys.	Idle	In	Out	In	Out	Cloned	Kept
Doc. Parsing & Indexing	168	55	7	35	1.2	1.9	1.2	1.1	9	73
Post-Parsing Indexing	69	46	9	42	1.8	2.6	1.2	1.2	54	76
Pre-Score Statistics	6	43	3	46	14.8	0.03	0.01	0	0	76
Page Quality Scores	69	27	13	57	1.6	6.4	0.77	0.77	3	76
Linkage Indexing	9	63	10	24	1.9	3.2	1.8	1.8	4	76
Phrase Extr. & Indexing	276	39	12	47	2.9	2.2	2.3	2.3	52	110
Word Index Merging	25	28	11	58	0.57	3.4	0	0	53	110
Scores into Word Index	87	57	16	25	2.3	2.5	3.9	3.9	53	110
Scores into Link Text Index	42	51	12	34	1.9	1.8	3.0	2.9	20	110
Word Statistics	33	57	2	37	6.8	0.09	0.30	0	1	110
Choosing Keywords	44	29	7	59	3.0	1.8	0.66	0.60	53	113
Building Directory Data	8	26	5	66	1.7	2.8	0.67	0.64	2	117
Word Index Sorting	23	28	8	60	0.4	3.4	0.02	0	54	117
Finding Similar Pages	116	44	11	43	2.7	2.3	2.5	2.5	0	143
Scores into Other Indexes	33	63	19	17	2.2	2.3	4.4	4.4	12	143
Sorting Other Indexes	5	33	2	54	0.05	4.8	0.09	0	14	143
Data Table Compaction	13	15	12	66	10.0	7.7	0	0	4	117

Table 3: Average per-cluster-node data processing performance and resource utilization.

speed excluding fetching of documents it thus 69 docs/sec. Provided this performance scales perfectly with respect to both the number of cluster nodes and the data size, it would take a cluster of 430 machines to process in two weeks the $3 \cdot 10^9$ pages presently covered by Google [11], which looks like a quite reasonable requirement.

6.4 Query Answering Performance

Because Yuntis was built for experimenting with different search engine data preprocessing stages, we did not optimize query answering speeds beyond a basic acceptable level. Single one-word queries usually take 10 to 30sec when no relevant data is in memory and 0.1 to 0.5sec when all needed data is in the memory of the cluster nodes. The longer times are dominated by the need to consult a few thousands of URL information records scattered on the disks. We currently do not have any caching of (intermediate) query results, except the automatic local file data caching in memory by the OS. The performance for multiword queries heavily depends on the specific words used: our straightforward sequential intersecting of word indexes would be significantly outperformed by a more optimized zig-zag merging based on binary search in the cases when very large indexes yield a much smaller intersection.

6.5 Quality of Search Results

To illustrate the usability of Yuntis we provide the samples in Table 4, report that Yuntis served 125 searches per day on average in February 2003, and encourage the reader to try it for themselves at `http://yuntis.ecsl.cs.sunysb.edu/`.

7 Enhancements and Future Directions

There are a number of general Yuntis improvements and extensions one can work on such as overall performance and resource utilization optimizations (especially for query answering), better tuning of various parameters, and implementation of novel searching services, for instance, classifying pages into ODP's directory structure [27]. As Table 3 shows, phrase extraction and indexing is one of the most important areas of performance optimization. One approach is to move phrase indexing into the initial document parsing and derive the phrase dictionary in a separate stage using a much smaller subset of good representative documents.

Another significant project is to build support for automatic work rebalancing and fault tolerance, so that cluster nodes can automatically share the work most equally, be seamlessly added, or go down during execution affecting only the overall performance. The approach here can be to consider sets of related partitions together with different batches of operations to them as the atomic units of data and work to be moved around.

Results for query university	**Keywords for** www.apple.com	**Pages linked like** www.subaru.com	**Pages textually like** www.cs.sunysb.edu
www.indiana.edu	apple computer inc	www.toyota.com	www.sunysb.edu
www.umich.edu	apple macintosh	www.vw.com	www.cs.uiuc.edu
www.stanford.edu	apple computers	www.saabusa.com	www.cs.umass.edu
www.wsu.edu	macintosh computer	www.pontiac.com	www.cs.berkeley.edu
www.uiuc.edu	macintosh computers	www.suzuki.com	www.cs.colorado.edu
www.cam.ac.uk	apple and	www.porsche.com	www.cs.man.ac.uk
www.about.bham.ac.uk	quick time	www.oldsmobile.com	www-cs.stanford.edu
www.cmu.edu	apple has	www.saturncars.com	www.cs.virginia.edu
www.msu.edu	computer the	www.volvocars.com	www.cs.unc.edu
www.cornell.edu	made with macintosh	www.mazdausa.com	www.suny.edu

Table 4: Top ten results for four typical Yuntis queries.

An important scalability (and work balancing) issue is posed by the abundance of power-law distributed properties [1] on the Web. As a result, many data tables have few records that are *very* large, for example, individual occurrence lists for most frequent words grow over 1GB at around 10 million document datasets, whereas most other records in the same table are under few KB. Handling (reading, updating, appending, or sorting) such large records efficiently requires special care such as not attempting to load (or even map) them into memory as a whole, and working with them sequentially or in small portions. In addition, such records reflect poorly on the ability to divide the work equally among cluster nodes by random partitioning of the set of records. A known solution is to split the records into subparts; for example, a word occurrence list can be divided according to a partitioning of the whole URL space. We are planning to investigate the compatibility of this approach with processing tasks that need to consider such records as a whole, and whether it is best to do this splitting for all records in a table or only for the extremely large ones.

8 Conclusions

We have described the software architecture, major employed abstractions and techniques, and implementation of the main processing tasks of Yuntis, a 167,000 lines feature-rich operational search engine prototype. We have also discussed its current configuration, its performance, and the characteristics of handled datasets, as well as outlined some existing problems and roads for future improvements.

The implementation of Yuntis allowed us to experiment with, evaluate, and identify several enhancements of our voting model [20] for assessing quality and relevance of web pages. The same is true for other search engine functions (such as phrase indexing, keyword extraction, similarity lists precomputation, and directory data usage), as well as their integration in one system — all while working with realistic datasets of millions of web pages.

The most important contributors to this success were the following: First, the approach of data partitioning and operation batching provided high cluster performance without task-specific optimizations leading to convenience of implementation and faster prototyping. Second, the modular, layered, and typed architecture for data management and cluster-based processing allowed us to build, debug, extend, and optimize the prototype rapidly. Third, the event-driven call/callback processing model was useful for allowing us to have a relatively simple, efficient, and coherent design of all components of our comprehensive search engine cluster.

Acknowledgments

This work was supported in part by NSF grants IRI-9711635, MIP-9710622, EIA-9818342, ANI-9814934, and ACI-9907485. The paper has greatly benefited from the feedback of its shepherd, Erez Zadok, and the USENIX anonymous reviewers.

Availability

The Yuntis prototype can be accessed online at `http://yuntis.ecsl.cs.sunysb.edu/`. Its source code is available for download at `http://www.ecsl.cs.sunysb.edu/~maxim/yuntis/`.

References

[1] Lada A. Adamic. Zipf, power-laws, and pareto - a ranking tutorial. Technical report, Xerox Palo Alto Research Center, 2000.

[2] The Apache Web Server, `www.apache.org`.

[3] The Berkeley Database, `www.sleepycat.com`.

[4] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The Connectivity Server: fast access to linkage information on the Web. In *Proceedings*

of 7th International World Wide Web Conference, 14–18 April 1998.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of 7th International World Wide Web Conference*, 14–18 April 1998.

[6] The bzip2 Data Compressor, www.digistar.com/bzip2.

[7] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. In *Proceedings of the Seventh World-Wide Web Conference*, 1998.

[8] The Common Object Request Broker Architecture, www.corba.org.

[9] The Distributed Component Object Model, www.microsoft.com/com/tech/DCOM.asp.

[10] The GNU Project Debugger, sources.redhat.com/gdb.

[11] Google Inc., www.google.com.

[12] Allan Heydon and Marc Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, 2(4):219–229, December 1999.

[13] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of web pages. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[14] The ht://Dig Search Engine, www.htdig.org.

[15] The Isearch Text Search Engine, www.cnidr.org/isearch.html.

[16] Dan Kegel. The C10K Problem, www.kegel.com/c10k.html.

[17] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 668–677, San Francisco, California, 25–27 January 1998.

[18] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track (USENIX-01)*, pages 141–154, Berkeley, California, June 2001.

[19] Maxim Lifantsev. Rank computation methods for Web documents. Technical Report TR-76, ECSL, Department of Computer Science, SUNY at Stony Brook, Stony Brook, New York, November 1999.

[20] Maxim Lifantsev. Voting model for ranking Web pages. In Peter Graham and Muthucumaru Maheswaran, editors, *Proceedings of the International Conference on Internet Computing*, pages 143–148, Las Vegas, Nevada, June 2000.

[21] Maxim Lifantsev and Tzi-cker Chiueh. I/O-conscious data preparation for large-scale web search engines. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*.

[22] The Logical Volume Manager, www.sistina.com/products_lvm.htm.

[23] The Linux Virtual Server, www.linuxvirtualserver.org.

[24] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the Web. In *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, May 2001.

[25] The Message Passing Interface, www-unix.mcs.anl.gov/mpi.

[26] The GNU Nana Library, www.gnu.org/software/nana.

[27] The Open Directory Project, www.dmoz.org.

[28] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the Web. Technical report, Stanford University, California, 1998.

[29] Jef Poskanzer. Web Server Comparisons, www.acme.com/software/thttpd/benchmarks.html.

[30] The Parallel Virtual Machine, www.csm.ornl.gov/pvm.

[31] Berthier Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Information Retrieval*, pages 105–112, Berkeley, California, August 1999.

[32] Web Robots Exclusion, www.robotstxt.org/wc/exclusion.html.

[33] The Simple Object Access Protocol, www.w3.org/TR/SOAP.

[34] The Standard Template Library, www.sgi.com/tech/stl.

[35] The Simple Web Indexing System for Humans, swish-e.org.

[36] The thttpd Web Server, www.acme.com/software/thttpd.

[37] The Webglimpse Search Engine Software, webglimpse.net.

[38] The eXternalization Template Library, xtl.sourceforge.net.

[39] Yahoo! Inc., www.yahoo.com.