How to build a WebFountain: An architecture for very large-scale text analytics

by D. Gruhl
L. Chavet
D. Gibson
J. Meyer
P. Pattanayak
A. Tomkins
J. Zien

WebFountain is a platform for very large-scale text analytics applications. The platform allows uniform access to a wide variety of sources, scalable system-managed deployment of a variety of document-level "augmenters" and corpus-level "miners," and finally creation of an extensible set of hosted Web services containing information that drives end-user applications. Analytical components can be authored remotely by partners using a collection of Web service APIs (application programming interfaces). The system is operational and supports live customers. This paper surveys the high-level decisions made in creating such a system.

This paper describes WebFountain as a platform for very large-scale text analytics applications. WebFountain processes and analyzes billions of documents and hundreds of terabytes of information by using an efficient and scalable software and hardware architecture. It has been created as an infrastructure in the text analytics marketplace.

Analysts expect this market to grow to five billion dollars by 2005. The leaders in the text analytics market provide easily installed packages that focus on document discovery within the enterprise (i.e., search and alerts) and often bring some level of analytical function. The remainder of the market is populated with smaller entrants offering niche solutions that either address a targeted business need or bring to bear some piece of the growing body of corporate and academic research on more advanced text analytic techniques.

Lower-function commercial solutions typically operate in the domain of a million documents or so, whereas higher-function offerings exist at a significantly lower scale. Such offerings focus primarily on the enterprise and secondarily on the World Wide Web through the mechanism of small-scale focused "crawls."

When large-scale exploitation of the World Wide Web is required, individuals and corporations alike turn to undifferentiated lower-function solutions such as hosted keyword search engines.[1,2] Typically, such solutions receive a small number of keywords (often one) and are unaware that the query comes from a competitive intelligence analyst, or an economics professor, or a professional baseball player.

Users with a business need to exploit the Web or large-scale enterprise collections are justifiably unsatisfied with the current state of affairs. Web-scale offerings leave professional users with the sense that there is fantastic content "out there" if only they could find it. Provocative new offerings showcase sophisticated new functions, but no vendor combines all these exciting new approaches—truly effective solutions require components drawn from diverse fields, including linguistic and statistical variants of natural language processing, machine learning, pattern recognition, graph theory, linear algebra, information extraction, and so on. The result is that corporate information technology departments must struggle to cobble together combinations of different tools, each of which is a monolithic chain of data ingestion, processing, and user interface.

This situation spurred the creation of WebFountain as an environment where the right function and data can be brought together in a scalable, modular, extensible manner to create applications with value for both business and research. The platform has been designed to encompass different approaches and paradigms and make the results of each available to the others.

A complete presentation and performance analysis of the WebFountain platform is unfortunately beyond the scope of this paper; instead, we adopt the approach taken by the book *How to Build a Beowulf,*[3] which laid out in high-level terms a set of architectural decisions that had been used successfully to produce "Beowulf" clusters of commodity machines. We now describe the high-level design of the WebFountain system.

Requirements

The requirements for a very large-scale text analytics system that can process Web material are as follows:

1. It must support billions of documents of many different types.
2. It must support documents in any language.
3. Reprocessing all documents in the system must take less than 24 hours.
4. New documents will be added to the system at a rate of hundreds of millions per week.
5. Some required operations will be computationally intensive.
6. New approaches and techniques for text analytics will need to be tried on the system at any time.
7. Since this is a service offering, many different users must be supported on the system at the same time.
8. For economic reasons, the system must be constructed primarily with general-purpose hardware.

Related literature

The explosive growth of the Web and the difficulty of performing complex data analysis tasks on unstructured data has led to several different lines of research and development. Of these, the most prominent are the Web search engines (see, for instance, Google[1] and AltaVista[2]), which have been primarily designed to address the problem of "information overload." A number of interesting techniques have been suggested in this area; however, because this is not the direct focus of this paper, we omit these here. The interested reader is referred to the survey by Broder and Henzinger.[4]

Several authors[5–9] describe relational approaches to Web analysis. In this model, data on the Web are seen as a collection of relations (for instance, the "points to" relation), each of which are realized by a function and accessed through a relational engine. This process allows a user to describe his or her query in declarative form (Structured Query Language, or SQL, typically) and leverages the machinery of SQL to execute the query. In all of these approaches, the data are fetched dynamically from the network on a lazy basis, and therefore, run-time performance is heavily penalized.

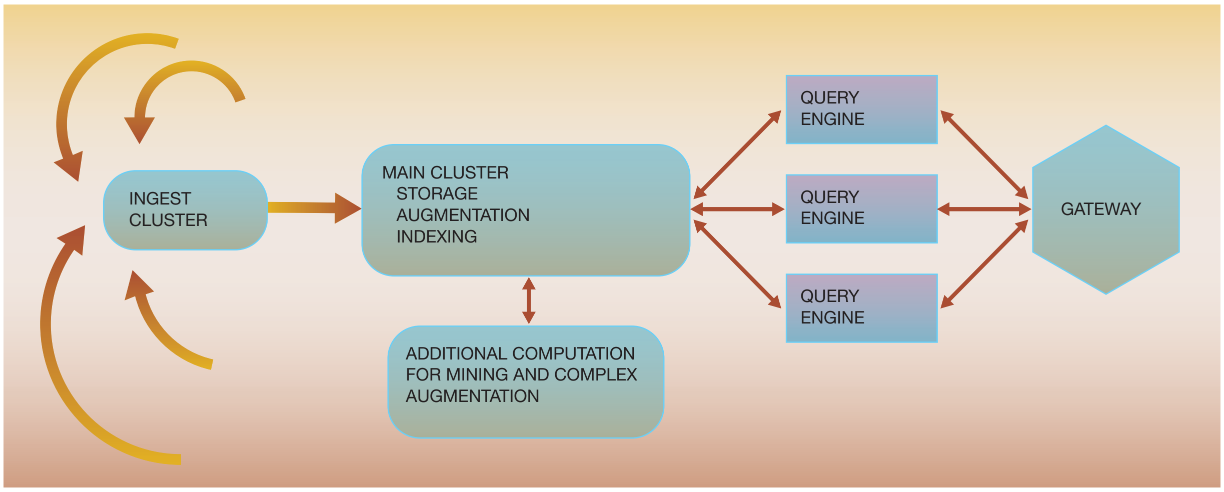
The Internet Archive,[10] the Stanford WebBase project,[11] and the Compaq Computer Corporation (now Hewlett-Packard Company) SRC Web-in-a-box project[12] have a different objective. The data are crawled and hosted, as is the case in Web search engines. In addition, a streaming data interface is provided that allows applications to access the data for analysis. However, the focus is not on support for general and extensible analysis.

The Grid initiative[13] provides highly distributed computation in a world of multiple "virtual organizations"; the focus is, therefore, on the many issues that arise from resource sharing in such an environment. This initiative is highly relevant to the WebFountain business model, in which multiple partners interact cooperatively with the system. However, architecturally WebFountain is a distributed architecture based on local area networks, rather than wide-area networks, and thus the particular models differ.

The Semantic Web[14] initiative proposes approaches to make documents more accessible to automated reasoning. WebFountain annotations on documents may be seen as an internal representation of standardized markup as provided by frameworks such as the Resource Description Framework (RDF),[15] upon which ontologies of markups can be built using mechanisms such as OWL[16] or DAML.[17]

Other research from different areas with significant overlap includes IBM's autonomic computing initiative,[18, 19] which addresses issues of "self-healing" for complex environments, such as WebFountain.

Figure 1 Information flow within the WebFountain environment



System design

The main WebFountain is designed as a loosely coupled, share-nothing parallel cluster of Intel-based Linux** servers. It processes and augments articles using a variant of the blackboard system approach to machine understanding.[20,21] These augmented articles can then be queried. Additionally, aggregate statistics or other cross-document meta-data can be computed across articles, and the results can be made available to applications.

The loosely coupled nature of the cluster makes it a natural for a Web-service style communication approach, for which we use a lightweight, high-speed Simple Object Access Protocol (SOAP) derivative called Vinci.[22]

We scale up to billions of documents by making sure that full parallelism can be achieved, and by adding a fair amount of hardware to solve the problem (currently, 256 nodes in the main cluster alone). This level of scaling is made possible because the same hardware and, in many cases, the same results of analysis are used to support multiple customers.

To support the multilingual requirement, all documents are converted to Universal Character Set transformation format 8 (UTF-8)[23] upon ingestion, allowing the system to support transport, storage, indexing, and augmentation in any language. We currently have developed text analytics for Western European languages, Chinese, and Arabic, with others being developed and imported.

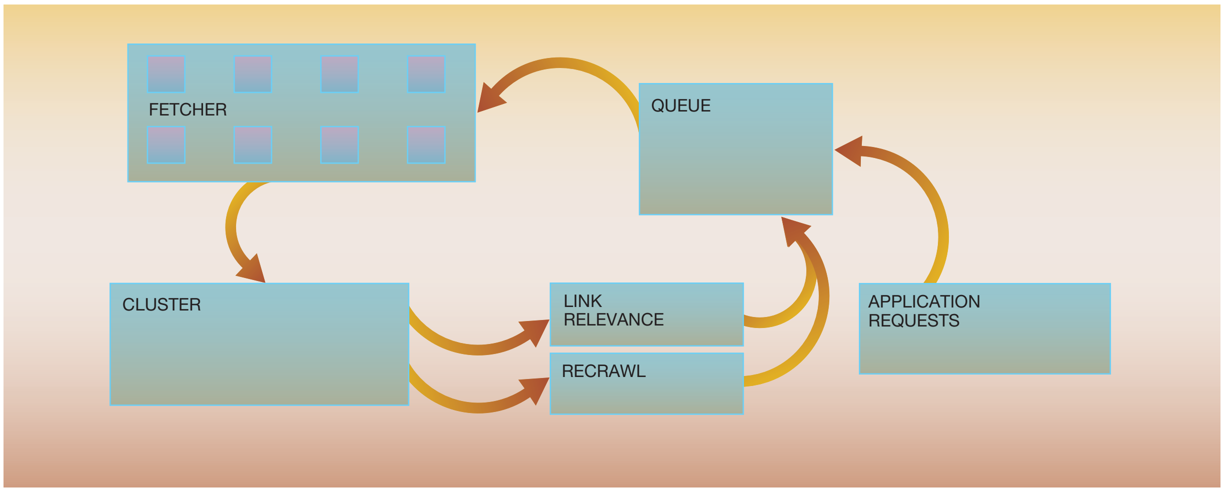
Ingestion is supported by a 48-node "crawler" cluster that obtains documents from the Web as well as other sources and sends them into the main processing cluster (see Figure 1).

Additional racks of SMP (symmetric multiprocessor) machines and blade servers supply the additional processing needed for more complex tasks. A well-defined application programming interface (API) allows new augmenters and miners (both described later) to be added as needed, and an overall cluster management system (also described later), backed by a number of human operators, schedules tasks to allow maximum utilization of the system.

Ingestion

The process of loading data into WebFountain is referred to as *ingestion.* Because ingestion of Web sources is so important to a system for large-scale unstructured data analysis, the ingestion subsystem is broken into two components. The first focuses on large-scale acquisition of Web content, for which the primary issues are the raw scale of the data and the heterogeneity of the content itself. The second focuses on acquisition of other sources, for which the primary concerns are extraction of the data itself and management of the delivery channel. We discuss these two components separately.

Figure 2 An example of crawling



Acquisition of Web data. The approach taken and the hardware and software used to acquire data are indicated in this discussion.

Approach. Crawling large portions of the Web requires a system that has performance points high enough to saturate the inbound bandwidth, but that also can perform a fair amount of analysis on the pages fetched to determine where to go next. Crawling is a cycle of queuing up pages to acquire, fetching, and then examining the results and deciding where to go next. These decisions can be quite complex to make because there is a desire to maximize the value of the pages fetched. (See Figure 2.) Such feedback is required to avoid traditional problems of crawling: automatically generated or trivially changed pages, sites with session identifiers (IDs) in the uniform resource locator (URL), content type of no interest to the user population, and so forth.

We achieve the performance through share-nothing parallelism in the fetcher (as well as share-nothing processing in the cluster). The "single point" is the *queue,* which fortunately is quite simple: URLs to crawl are communicated to it from the various evaluators along with a priority. These pages are then pulled from it by the various fetcher instances that are allocated work on a simple hash of the host name. Each fetcher node then maintains its own queue and selects the next URLs to crawl, based on freshness, priority, and politeness (avoiding heavily loading a Web server with multiple successive accesses to the same server).

Hardware. The processing portion of the crawler is part of the main mining and storage cluster and is thus discussed later. The queue resides on a single queue management machine (an IBM xSeries* Model x335 server, which uses a 2.4 gigahertz (GHz) Intel Xeon** processor with 4 gigabytes (GB) of read only memory, or RAM). Requested Web pages are dispatched via hash on the site name to 48 fetcher nodes, which are responsible for throttling load on the sites being examined, obeying politeness rules, and so forth. These machines connect to the Internet at large (at the moment) through 75 Mbps of an OC3 (optical Carrier level 3) line.

Software. The fetch cluster is coordinated through a Web service interface, with a DB2* system to hold information to support politeness policies. The fetcher itself is written in C++, as is the queue. We run multiple DNS (Domain Name System) servers and caches to reduce load on our upstream DNS providers. The queue supports priorities and has a transaction type semantic on work (preventing a failed fetch machine from resulting in data loss).

Acquisition of other data sources. WebFountain employs a number of other data sources as well: traditional news feeds, preprocessed bulletin boards, discussion groups, analyst reports, and a variety of both structured and unstructured customer data.

Approach. All of these sources come with their own unique delivery method, formatting, and so on. The ingestion task for all these methods consists of ra-

Figure 3 An example of entity data stored in a frame

```
UEID: 00005509873A.....
TYPE: Person
NAME: John Doe
HOMEPAGE: http://john.doe.com
AGE: 15
.
.
.
.
.
.
```

Figure 4 An example of XML representation for entity data

```
<ENTITY>
    <UEID>00005509873A....</UEID>
    <TYPE>Person</TYPE>
    <NAME>John Doe</NAME>
    <HOMEPAGE>http:// john.doe.com</HOMEPAGE>
    <AGE>15</AGE>
    .
    .
    .
</ENTITY>
```

tionalizing the content into Extensible Markup Language (XML), which can then be loaded into storage, or the store, through an "XML fetcher." For most sources, this operation includes an attempt to either provide or begin a chain of mining that will result in the running text being available to further mining chains (so called DetaggedContent). In this way, many of the high-level linguistic analyses can be run on all data, regardless of source.

Hardware. As might be imagined, "other" data sources require a variety of data access approaches. Some of the data comes on CDs (compact disks), some on magnetic tape, some on removable hard drives, some via Web site, much via FTP (File Transfer Protocol) (both pull and push), some via Lotus Notes* database replication, some via e-mail, and so forth.

Each of these delivery mechanisms may imply a single machine per source, or a machine shared across a small number of sources, to accommodate the particular needs of that source. For instance, particular operating system versions are a typical requirement. However, the data volume on these sources tends to be relatively small, so in most cases a single IBM xSeries Model x335 server is sufficient.

Software. Typical data sources require a specialized "adapter." Each of these adapters is responsible for reducing the input data to XML files, usually one per "document," as described in the preceding subsection "Approach."

Data storage

The task of the WebFountain Store component is to manage entities (where an entity is a referenceable unit of information) represented as frames[24] in XML files. This management entails storage, modification, and retrieval. In WebFountain, entities are typically documents (Web pages, newsgroup postings), but might also be concepts such as persons, places, or companies. Entities have two properties: a type and a set of keys, each of which is associated with a multiset of values. Interpretation of the semantics of a particular key depends on the entity type. Common to all entity types is a key called the Universal Entity Identifier (UEID), which holds the globally unique 16-byte identifier for that entity. See Figures 3 and 4.

Challenge. The WebFountain store must receive entities, modify them, and return parts of them as needed. The challenge is one of scale in the face of several very different access patterns that need to be supported. These access patterns can be classified as creating new entities, modifying existing ones, or reading parts of existing ones. Access for a particular client can be either a sequential pass through most or all of the data, or a sequence of random accesses.

The key problem is avoiding the overhead of a disk seek for each access of each client. Latency for a small seek followed by a read or write is dominated by the seek time, and thus limited to the low hundreds per second. The use of RAID5 (redundant array of independent disks, level 5) to help with maintenance and uptime just exacerbates the problem as each write to the array becomes three to four writes to the devices.[25]

The traditional approach for dealing with these types of patterns has been to cache heavily. This does not help as much as we might like as the data scale is too large and the typical access pattern too random for cache hits to result in substantial savings.

Approach. Regardless of other tricks used to address this problem, it is always desirable to have access to numerous disk heads working in parallel. If the problem can be spread over many disk heads (or arrays), and the order in which data are returned is unimportant, this spreading can result in linear increases in speed. We hash a unique part of the entity (in most cases the UEID) to determine storage locations, providing uniform distribution across all the devices, and take full advantage of share-nothing parallelism on the later mining and indexing steps.

On the disk itself, we take a compromise position of storing data together in *bundles* of a few hundred entities. The number of entities in the bundle is a tuning parameter that can be adjusted to match the overall workload. This approach allows sequential access when the whole data set needs to be examined, at the expense of a small penalty for each random access. Random access is achieved by hosting a UEID to bundle lookup using a fully dynamic B+Tree data structure on disk where all the nonleaf nodes are in memory. For storage devices that handle fewer than five million UEIDs, the entire lookup tree is kept in memory for faster access.

The physical storage is fronted by a Web service interface that performs some access optimization and pooling of data access. It provides separate sequential and random access APIs and uses a different family of optimization techniques in each case.

Hardware. We are using 2560 72-gigabyte drives, arranged into 512 five-disk RAID5 arrays. These drives are hosted by IBM TotalStorage* IP Storage 200i iSCSI (Internet Small Computer System Interface) devices, and the actual storage interface is via 256 network-attached IBM xSeries Model x335 Linux servers, grouped into eight-server nodes, connected via a jumbo frame gigabit network. This works out to approximately 0.5 terabyte per "node" of formatted space, which is a suitable mix of processor and storage for the augmentation to be done later (see the next section). The storage is formatted with a ReiserFS V2 file system. We are running various Linux kernels in approximately the Version 2.4.15 to Version 2.4.20 range.

Software. The serialization format used to transfer data over the network is the same format used to store frames on disk. This helps to reduce processing overhead because little or no parsing of the data is needed. This low CPU utilization approach is important because the iSCSI storage approach does require a certain amount of computation itself, and we need to do augmentation and indexing on the nodes as well.

Table 1 Performance of storage per node (documents/second)

Access	Read	Create	Modify
Sequential	440	200	350
Random	420	200	150

The storage server is multithreaded both in the client requests and disk access (via asynchronous I/O), so as to achieve deep disk queues and the corresponding increases in speed.[26]

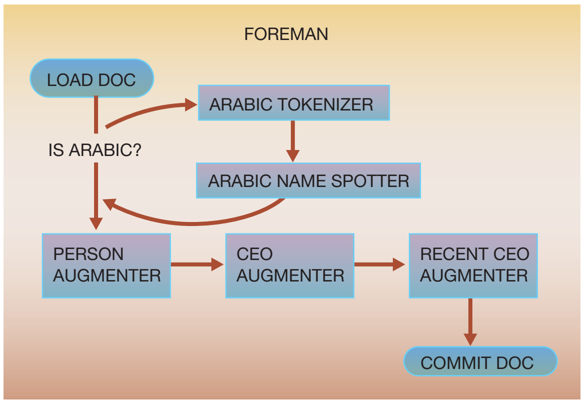
For sequential access, storage reads an entire bundle (typically 8 MB or larger) to increase read size, and also performs prefetching of the next bundle when appropriate. The use of bundles, however, requires periodic reorganization to remove deleted records. Thus, an on-line reorganization thread needs to run in the background (see, for example, Sockut and Iyer[27]).

The last feature is a very simple skip selection function used with sequential access. Certain commonly accessed collections have a tag embedded in their directory information that allows storage to provide (for example) all the nonduplicate, nonpornographic English language pages (a common starting point for marketing research).

Performance. Table 1 shows current performance numbers in documents per second, where a document is a Web page. As the table shows, read access is nearly the same in both access patterns, but random writes are considerably more expensive.

Future directions. Future explorations will include the trade-offs of direct-attached versus network-attached storage (NAS), testing of new NAS approaches such as RDMA (Remote Direct Memory Access), the use of hardware iSCSI controllers, the possibility of either changing the file system to ReiserFS V4 or XFS from Silicon Graphics, Inc. (SGI) or dispensing with it entirely and managing the raw device. Lastly, kernel-based asynchronous I/O is developing and may

Figure 5 The Foreman process



provide another considerable decrease in processor I/O overhead.

Augmentation

Augmenters are special-purpose programs that extract information from entities in storage and add new key or value pairs to these entities. Each augmenter can be thought of as a domain-specific expert. The blackboard approach is traditionally implemented as a set of experts considering a problem while sitting in front of a virtual blackboard, each adding a comment to the blackboard when that expert understands part of the problem.[20,21] However, a straightforward implementation of this approach leads to inefficiencies caused by contention and excessive disk I/O, as each expert examines and augments each entity independently. WebFountain takes a slightly different approach of moving the blackboard past each expert and giving each a chance to add comments as the data stream by.[28] This approach turns what was a contention problem into a pipeline, at the price of somewhat decreased interaction among the experts.

We refer to these experts by the somewhat less pretentious term "augmenters." For example, an augmenter might look at an article and extract the names of the people mentioned therein. The next augmenter might match the extracted names against a list of chief executive officers (CEOs), and a third augmenter might further annotate certain CEOs with the year in which they acquired the position.

A similar chain of augmenters might recognize syntactic structures, such as important domain-specific noun phrases or geographical entities, or might augment a page with a version of the content translated from Korean to English.

Where appropriate, these augmentations are then merged and indexed. The index query language will accept simultaneous requests for augmentations produced by different augmenters, allowing queries that might, for example, determine all pages with a reference to a European location, a new CEO, and a synonym for "layoff."

Challenge. Each augmenter is an independent process that must run against some subset of the entities in the system. The challenge is to perform these augmentations as quickly and efficiently as possible, taking advantage of ordering, batching, and so forth, given that the augmenters themselves are at times not "hardened" (production-level) code and thus must run isolated ("sandboxed"). Augmenters may have different access control restrictions, may require different subsets of the data, may exhibit dependencies on one another, and may run on different physical machines, perhaps because of operating system requirements. In this complex space, the system must determine the optimal manner in which to run ("gang together") augmenters.

Approach. The chief tool available to the optimization engine is a Foreman process that spawns and monitors a sequence of augmenters and passes data through the sequence, incurring only a single read/write operation for the entire chain (Figure 5).

We put a number of augmenters together in this pipeline until their memory and CPU requirements match the disk I/O and call the resulting set a "phase" of augmentation. The data are processed through multiple phases, depending on the kind of data. The Foreman process allows conditional branching, so for example, Arabic tokenization is only run on the entity if it is in Arabic.

The Foreman process provides "sandboxing" and monitoring by keeping limits on process memory usage and processing times and restarting processes that appear to be hung or misbehaving. Errors are logged, together with recent input data, to facilitate reproducing the error in a debugging environment. This process gives the system a high degree of robustness in the presence of possibly unreliable augmenters.

Grouping augmenters into phases, given the requirements above, is an ongoing research problem; our current groupings are determined manually.

Hardware. As noted above, augmentation occurs whenever possible on the same hardware as the storage. Sometimes "off-node" hardware will be used for particularly computationally intensive augmentation, or for augmentation that cannot be run locally for other reasons (permissions, code ownership, legacy operating system requirements, and so forth).

Software. There is no single approach to writing an augmenter, nor any single language in which the augmenter must be written. A number of libraries seek to simplify the task by allowing the augmenter author to write a function that takes an entity and returns an entity with the augmentations added. Augmenters written in primary supported languages (currently C++, Java**, and Perl) need only provide a *processOnePage()* method, and can then be automatically run across the distributed cluster, monitored, and restarted. Pointers to entities that cause crashes can be passed to the author.

The Foreman process itself represents a part of the augmenter software stack. It allows augmenters to connect to a stream of new entities coming into the system, run over the whole storage, or run over the result of a query to storage without needing any changes to the augmenters themselves.

Performance. Augmenters running over the whole storage can see around 300 entities per second per node. Augmenters that are processing the results of a query can see around 100 entities per second per node. As noted earlier, there are 256 nodes on the system for a total rate of 76 800 entities per second or 25 600 for query processing.

Future directions. Most of the future work on augmentation will be to enhance the ease of authoring augmenters and miners, as well as to enhance the sandboxing and debugging tools to identify problems and help with their resolution.

Index

The WebFountain indexer is used to index not only text tokens from processed entities (including, but not limited to Web pages), but also conceptual tokens generated by augmenters. The WebFountain indexer supports a number of indices, each supporting one or more different query types. Boolean, range, regular expression, and spherical are typical. More complex queries include graph distance (e.g., as in the game of how many people between a person and the actor Kevin Bacon), spatial (e.g., pages within San Jose, California), and relationships (e.g., people who work directly for John Smith, CEO of XYZ Company).

Challenge. Indexing in this environment presents five challenges. Indices must:

- Build quickly
- Build incrementally
- Respond to queries promptly
- Use space efficiently
- Deal with result sets that may be many times larger than machine memory

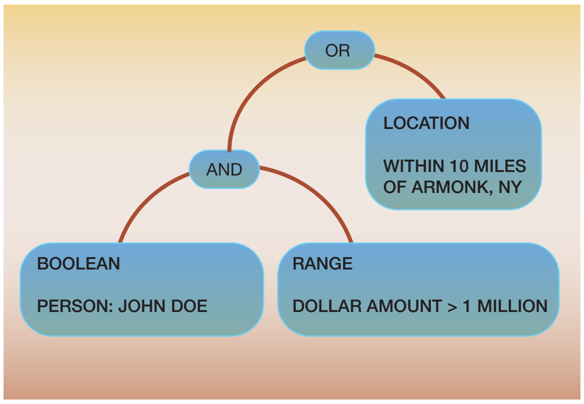
All of these requirements must be addressed in an environment with trillions of total indexed tokens and billions of new or changed tokens every week.

Approach. We will address only the main indexer for the rest of this section. Because the main index is a fully positional index, the location of every token offset within every entity (document) is recorded, along with possible additional attributes that can be attached by augmenters to each token occurrence. The indexing approach is scalable and not limited by main memory because we adopt a sort-merge approach in which sorted runs are written to disk in one phase and final files are generated in a merge phase. To allow larger-than-memory result sets, most analytical queries return results in UEID order.[29] This allows most joins in the system to be performed as merges, without buffering of one of the result sets—a great convenience when result sets may represent hundreds of billions of entities and take days to consume.

The indexer supports the WebFountain Query Language (WFQL), the language that allows processes within the system to specify declarative combinations of result sets from different parts of the system (see Figure 6). Fragments of WFQL can be pushed down into the indices themselves; for example, Boolean subtrees can be handed in their entirety to the Boolean index, limiting the amount that needs to be sent over the network. However, when results must be aggregated over multiple indices, the WebFountain Joiner is responsible for handling the resulting joins. See the section "Querying" later for more details.

Hardware. Again, because hardware is the main cluster of nodes, indices are built where the data are

Figure 6 Sample WFQL query



stored and augmented for both performance and convenience reasons. This siting is relaxed for some specialized indices, which can run on stand-alone IBM xSeries model x350 machines (using 700 MHz four-way Intel Xeon processors with 4 GB of RAM) when appropriate. Currently, we migrate indexing away from the data for indices in development or when we index a relatively small set of the entities.

Software. The main indexer is implemented as a distributed, multithreaded C++ token stream indexer[30] run on each of the cluster nodes. It employs an integer-compressed-posting-list format to minimize not only storage requirements, but also I/O latency on complex joins that are passed down to it.

Future directions. The more WFQL that can be pushed down to the index the better. Doing this intelligently will require better costing estimates and models for setup and transport times on various queries. This improvement in turn leads to identification of frequent queries (for caching), as well as more involved query optimization.

Miners

We begin with a description of the distinction between entity-level operations (augmentation) and corpus-level (cross-entity) operations (mining). Augmenters have a specific data model: they process each entity in isolation without requiring information from neighboring entities. As described previously, they are easily parallelizable and can provide significant programming support. Tokenization, geographic context discovery, name extraction, and full machine translation are all examples of tasks that execute one page at a time. Miners, in contrast, perform tasks such as aggregate statistics, trending, relationship extraction, and clustering. They must maintain state across multiple entities in order to do their job. Typically, such miners would begin by running an augmenter over the data, possibly just to dump certain extracted information. However, the miner would then begin aggregate processing of all the dumped information. Finally, the miner can either upload data back to storage (to the processed entity, or to another entity such as a Web site, or a corporation), or the miner can present the results of its mining as a Web service without writing back to storage.

Challenge. The primary challenge in mining is scalability. Additionally, even sorting takes a significant amount of time (measured in hours or days), and quadratic time algorithms are essentially infeasible until the data have been dramatically distilled. Furthermore, in a multibillion entity corpus, even the problem of separating the relevant information from the noise may become a large data question.

Approach. With many cross-entity techniques, a multitier approach must be used to reduce data-scale to more manageable levels. A simple example is to query all entities that match some trivial selection (such as, “Must mention at least one Middle-Eastern country”) and then look at this subset (which may be less than one percent of the whole Web) for further processing.

If several “sieving” approaches can be used, a data set several orders of magnitude smaller can be considered, and many more complicated techniques are then available.

Hardware. These cross-entity approaches generally run on a separate rack of IBM xSeries Model x350 machines that allow more computation than the storage nodes themselves. Some miners use data warehousing to move data of interest to a DB2 database for further (often OLAP, or on-line analytical processing) style investigation. Additionally, some whole-Web-graph problems are run on an Itanium** system, which has a very large amount of memory installed. In short, these mining problems can be run on whatever platform is appropriate to the task at hand.

Software. Likewise, these approaches often require specialized hardware or software. Because of the diverse nature of mining operations, little generic support can be provided by the system. In general, the author of the miner must be aware of issues of persistence and distributed processing that can be hidden in abstract superclasses for augmenters. Additionally, the task of scheduling miners requires a more involved workflow specification because the miner usually is triggered after a particular suite of augmenters complete and dump useful information. Once the miner operation is completed, there may be a final upload of the resulting data back to storage.

For example, consider a link-based classifier such as Hyperclass.[31] Such a classifier would dump feature information for each entity (using an augmenter), would interact with a distributed Web service providing connectivity information (see, for instance, the Connectivity Server[32]) to generate neighborhood information, and would then perform an iterative cycle with a data access pattern much like power iteration to compute final classes. Once the classes for each entity have been computed, yet another augmenter would run to upload the classes for each entity back to storage.

Performance. The cross-page mining rate is somewhere between 25 thousand and 70 thousand entities per second.

Future directions. The most challenging future problems for WebFountain lie in the mining space. The techniques of data mining, graph theory, pattern recognition, natural language processing, and so forth are all amenable for reapplication to the new domain of very large-scale text analytics. In most cases, the traditional algorithms require modification to maintain efficiency; this domain therefore represents a fruitful opportunity to apply existing techniques in new ways on a timely and valuable data set.

Querying

As introduced earlier, WebFountain supports a query language that generates augmented collections of entities by combining results from various services within the platform. A query consists of selection criteria and a specification of the particular keys that should decorate the resulting entities. For notational purposes we use an SQL derivative to capture most common queries that the system can process. A typical query might be as follows:

```
SELECT URL, UEID, Companies
  FROM Web
  WHERE
    Person='John Smith'
  AND Location WITHIN
    '10 miles of San Jose, CA'
```

The results are then returned as an enumeration of XML fragments containing the requested data.

Challenge. Recall that queries must run against terabytes of data stored on hundreds or even thousands of nodes. The example above is easy because it can be sent in parallel to all the nodes. A more complex query, requiring a more complex data flow, would be "Give me all the pages from sites where at least one page on the site is in Arabic."

A second challenge arises from the possible size of the result sets. These sets may need to be shared between multiple clients and may need to deal with clients crashing and restarting where they left off. This robustness is easier, thanks to the loose coupling, but still requires a fair amount of complexity, especially in deciding when to drop queries or fragments of result sets as "abandoned."

Finally, as in structured data queries to relational databases, efficient computation of result sets relies heavily on the optimizer. In a loosely coupled system, the cost of moving entries that will later be trimmed from one machine to another can easily dominate query execution time, resulting in situations that are not standard territory for database optimizers. Further, the system allows the dynamic introduction of engines to perform intermediate stages of query processing, and these lightweight distributed engines must be capable of specifying enough information about their performance to allow the optimizer to generate an efficient distributed query plan.

Approach. Although expressive, there are times when SQL is not expressive enough; therefore, the common query format is WFQL, an XML query plan. It allows more complex interactions between services to be scripted. SQL-type queries are converted to this format before processing by a front end.

After the WFQL proposal is received, the tree is optimized. Optimization includes tree rewriting, rebalancing where appropriate, and determination of subtrees that can be "pushed down" in large amounts to the leaf services (such as indices). This new WFQL plan is then executed, and the results served up to the client as an enumeration of XML fragments.

Hardware. The query engine (called a joiner) runs on its own dedicated IBM xSeries Model x350 machine. Because these queries are independent, additional instances of the joiner can easily be added to the system until the cluster is saturated.

Software. The query engine performs a three-step process: taking in the query in a variety of formats and translating it to WFQL (the front end), optimizing it (the middle end), and executing the resulting plan (the back end). Currently the middle end is fairly trivial, but see below for some discussion of how this may change.

Performance. The current joiner has a latency of around 20 milliseconds for relatively simple queries. Queries that require a resorting of the results can take much, much longer, as $n \log n$ (where n is a billion) can run to a few days.

Future directions. As noted earlier, accurate query cost estimation, possibly requiring statistics gathered during query execution, is a key requirement for optimization. Two key future directions in this area are the following: First, integration of the DB2 DataJoiner product for query rewriting, which would allow us to support more ad hoc queries without worrying about bringing down the cluster; second, introduction of multiquery optimization—for example, should a number of queries need to be run every night, could they be combined in some way to limit the number of data accesses.

Web service gateway

The scale of hardware required for WebFountain makes it infeasible to build a separate instance for each customer. Instead, WebFountain is a service offering that performs data processing in a single centralized location (or a few such locations) and then delivers results to clients from these locations. Given the existing Web service approach used internally, it is natural to leverage that decision by providing result data to clients through a Web service model as well.

Challenge. There are three primary challenges in the design of the gateway. First, and most important, access must be as simple as possible to encourage developers to write programs that make use of the platform. Second, the gateway must provide access controls, monitoring, quality of service guarantees, and other user management tasks. Third, the gateway must provide performance sufficiently high to meet the needs of users.

Approach. We do this data processing through a SOAP[33] Web service gateway. For each service to be exposed externally, a WSDL[34] (Web Services Description Language) is published. Connection to the gateway is by SSL2 MAC (Secure Socket Layer Version 2 mutually authenticated certificate). Clients register with the gateway and negotiate the commands they are allowed to execute, the load they are allowed to place on the system, and the families of parameters they are authorized to specify. The gateway monitors quality of service and bandwidth for each client, based on a logging subsystem that captures queries, response times, and meta-data that arrive with each completed internal query.

Commands exposed through the WSDL need not map directly to commands inside the cluster. For example, an external query for the current crawling bandwidth might result in an internal query to all 48 crawlers.

Hardware. The gateways are set up as a set of IBM xSeries Model x330 machines (using 1.13 GHz dual Intel Pentium** III Processors with 2 GB of RAM), dual network zoned, and behind firewalls. The number of gateway machines can be trivially scaled up to meet demand, dynamically if necessary.

Software. The gateways themselves are written in the C++ and Java languages. They perform the task of authenticating the queries, logging them, translating them to the requisite xtalk queries, dispatching them to the cluster, aggregating the results, rephrasing them as SOAP, and returning them.

Performance. The current performance point is tens of queries per gateway per second. This number obviously varies considerably depending on the size of the result set being returned.

Future directions. Future work for the gateway includes supporting higher degrees of granularity on querying (resulting in a more complex set of supported queries), better performance by running similar queries together, faster deployment of new functionality through dynamic plug-ins to the gateway, and integrated load balancing, sharing, and grouping of requests.

Cluster management

Although having a large number of machines available allows us to overcome a problem of several orders of magnitude of scale, it introduces several orders of magnitude of complexity in keeping the system running. Maintaining an overview of 500 machines can be taxing, particularly coupled with the requirement that the system be resilient to some number of failed nodes.[35]

Nonetheless, we need to identify problems, fix them automatically when possible, call in human support otherwise, and allow institution of workarounds while the problem is being dealt with. Additionally, the cluster management subsystem is responsible for the more mundane automation tasks that surround workflow management, automatically distributing processes across nodes of the cluster, monitoring for progress, restarting as necessary, and so forth.

Challenge. The complications of running such a system include the heterogeneous nature of both hardware environment and software deployment (a variety of code versions running all at the same time). Because of requirements on turnaround time and the vicissitudes of Web data, software failures in the augmenters and miners are inevitable. The system must not rely on an error-free execution. If an augmenter fails once, the system will log the error, restart the process, and go on. If it fails several times on the same entity, the author will be notified, and the entity will be skipped. Miners, in contrast, may or may not provide mechanisms for restart. Without such a mechanism, the system will merely retry the miner and request operator assistance based on the priority of the miner task.

In a complex system the root cause of a problem is often elusive. Cascade failures are quite common, and looking at crash logs can be a time-consuming task. Although rigorous testing and debugging is done before introducing new code, by definition no test is complete until it has run on the whole Web. We always find things that do not scale as we wish in production, or that are tripped up by truly odd pages. Additionally, resource contention problems are hard to model in test and often only appear in production.

Approach. The cluster management subsystem runs a special service on each machine known as a *nanny.* This process forks and runs all the services needed on a machine and monitors their performance, CPU and memory usage, ends them on request or when thresholds are exceeded, and reports on their status when queried. Any major changes that the nanny undertakes (e.g., install new code or start a new service) are authenticated and logged.

A central coordinator checks with all these nannies every few seconds and creates an aggregate view of the production cluster. In addition to services, each nanny also monitors overall system status, including disk status, CPU load, memory usage, swapping, and network traffic, as appropriate. This information is logged centrally to a "cluster flight recorder," which can be replayed to find unusual performance bottlenecks.

Simply managing hundreds of machines is a conceptual challenge as well. For the operators and technicians, nodes are grouped into sets of eight, which represent a "rack" for administrative purposes. Visual displays use this grouping to allow rapid drill down to machines and problems.

Hardware. A single IBM xSeries Model x335 machine serves as the central coordinator; one instance of the nanny runs on every main cluster node and every ingester.

Software. Surprisingly little work has been done on managing large, loosely coupled clusters (although grid research[13] is beginning to look promising). As a result, the nanny or coordinator is a custom implementation in a mix of C++ and Java languages. A number of commercial Web service monitors can be used to monitor the SOAP gateway, but these monitors still need to be examined by the central coordinator to provide a uniform view.

Performance. The current cluster supports a half dozen "clients" at a time and requires 7.5 people to run. It is unclear what the scaling relationship is, but the goal is to be highly sublinear.

Future directions. Our primary goals for the future in cluster management are improved problem determination and enhanced speed of resolution, including autonomic swapping of hot spares to facilitate automatic failover. The goal is to provide better utilization of the hardware with a smaller operations staff.

Conclusion

The WebFountain system currently runs and supports both research and a set of customers who are involved in "live" use of applications hosted in the production environment. As such, the architecture has completed the first phase of its development: going live with a limited set of customers.[36,37] In this paper, we have disclosed at a high level the architectural decisions we have made to complete this first phase of execution, with an eye to the rapid expected growth in both data and load (measured as number of applications, number of partners, and number of users).

We adopted the Web service model because we needed the traditional benefits of such an architecture: modularity, extensibility, loose coupling, and heterogeneity. So far, we have delivered multiple real-time services backed by 100 terabytes of data with debugging cycles measured in days and extensibility that exceeded our expectations. Although our requirements will only become more severe, we do not anticipate needing to revisit this basic architecture in order to meet them.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Linus Torvalds, Intel Corporation, or Sun Microsystems, Inc.

Cited references and notes

1. Google, http://www.google.com.
2. AltaVista, http://www.altavista.com.
3. T. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese, *How to Build a Beowulf*, The MIT Press, Cambridge, MA (1999).
4. A. Broder and M. R. Henzinger, "Algorithmic Aspects of Information Retrieval on the Web," in *Handbook of Massive Data Sets,* M. R. J. Abello and P. M. Pardalos, Editors, Kluwer Academic Publishers, Boston, forthcoming.
5. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The Lorel Query Language for Semistructured Data," *International Journal of Digital Libraries* **1,** No. 1, 68–88 (1997).
6. J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hilldrum, D. Maden, V. Raman, and M. A. Shah, "Adaptive Query Processing: Technology in Evolution," *IEEE Data Engineering Bulletin* **23**, No. 2, 7–18 (June 2000).
7. G. Arocena, A. Mendelzon, and G. Mihaila, "Applications of a Web Query Language," *Proceedings of the 6th International World Wide Web Conference (WWW6),* Santa Clara, CA (1997), pp. 1305–1315.
8. E. Spertus and L. A. Stein, "Squeal: A Structured Query Language for the Web," *Proceedings of the 9th International World Wide Web Conference (WWW9)* (2000), pp. 95–103.
9. G. Mecca, A. Mendelzon, and P. Merialdo, "Efficient Queries over Web Views," *Proceedings of the 6th International Conference on Extending Database Technology (EDBT),* Valencia, Spain, *Lecture Notes in Computer Science* **1377,** Springer-Verlag (1998) pp. 72–86.
10. The Internet Archive, http://www.archive.org.
11. J. Hirai, S. Raghavan, A. Paepcke, and H. Garcia-Molina, "WebBase: A Repository of Web Pages," *Proceedings of the 9th International World Wide Web Conference (WWW9)* (2000), pp. 277–293.
12. *Web-in-a-Box, Web Archeology*, Hewlett Packard SRC Classic Lab, Palo Alto, CA, http://research.compaq.com/SRC/WebArcheology/wib.html.
13. I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Lecture Notes in Computer Science* **2150** (2001).
14. *Semantic Web Activity: Advanced Development, Technology and Society Domain*, W3C, http://www.w3.org/2000/01/sw/.
15. O. Lassila and R. R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification,* W3C Recommendation, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ (February 1999).
16. D. L. McGuinness and E. F. van Harmelen, *OWL Web Ontology Language Overview,* W3C Candidate Recommendation, http://www.w3.org/TR/owl-features/ (August 18, 2003).
17. The DARPA Agent Markup Language (DAML) Homepage, http://www.daml.org.
18. A. Wolfe, "IBM Sets Its Sights on Autonomic Computing," News Analysis, *IEEE Spectrum* (January 2002).
19. P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology,* IBM Corporation (October 15, 2001), http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
20. A. Newell, "Some Problems of the Basic Organization in Problem-Solving Programs," *Proceedings of the Second Conference on Self-Organizing Systems,* Washington, DC (1962), pp. 393–423.
21. L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay Speech Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Surveys* **12**, No. 2, 213–253 (1980).
22. R. Agrawal, R. Bayardo, D. Gruhl, and S. Papadimitriou, "Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications," *Proceedings of the Tenth International World Wide Web Conference (WWW10),* Hong Kong, China (2001), pp. 355–365.
23. F. Yergeau, *UTF-8, A Transformation Format of ISO 10646,* Internet Engineering Task Force (January 1998), http://www.ietf.org/rfc/rfc2279.txt.
24. M. Minsky, *A Framework for Representing Knowledge,* Technical Report, MIT-AI Laboratory Memo 306, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA (June 1974).
25. D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM Conference on Management of Data (SIGMOD)* (June 1988), pp. 109–116.
26. M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," *Proceedings of the USENIX Winter 1990 Technical Conference,* USENIX Association, Berkeley, CA (1990), pp. 313–324.
27. G. H. Sockut and B. R. Iyer, "A Survey of Online Reorganization in IBM Products and Research," *IEEE Bulletin of the Technical Committee on Data Engineering* **19**, No. 2, 4–11 (1996).
28. D. Gruhl, *The Search for Meaning in Large Text Databases,* Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA (2000).
29. User queries may be returned in rank orders that are appropriate for viewing, but long-running queries that are processed to completion are returned in UEID order.
30. C. Clarke, G. Cormack, and F. Burkowski, "Shortest Substring Ranking (MultiText Experiments for TREC-4)," *Proceedings of the Fourth Text Retrieval Conference* (November 1995).

31. S. Chakrabarti, B. Dom, and P. Indyk., "Enhanced Hypertext Classification Using Hyper-Links," *ACM SIGMOD International Conference on Management of Data* (1998), pp. 307–318.
32. K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian, "The Connectivity Server: Fast Access to Linkage Information on the Web," *Proceedings of the 7th International World Wide Web Conference* (April 1998), pp. 14–18.
33. *Simple Object Access Protocol (SOAP) 1.1*, W3C, http://www.w3.org/TR/SOAP/.
34. *Web Service Definition Language (WSDL)*, W3C, http://www.w3.org/TR/wsdl.
35. Since each node represents less than a half percent of our data, having one or even two down does not materially impact the quality of queries that develop an aggregate statistical understanding over a broad data set.
36. For information on the particular set of mining and applications, please contact the WebFountain team directly.[37]
37. *WebFountain Overview*, IBM Corporation, Almaden Research Center, http://www.almaden.ibm.com/webfountain.

Accepted for publication August 12, 2003.

Daniel Gruhl *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (dgruhl@almaden.ibm.com).* Dr. Gruhl is a researcher at the Almaden Research Center. He earned his Ph.D. in electrical engineering from the Massachusetts Institute of Technology in 2000, with thesis work on distributed text analytics systems. His interests include stegonography (visual, audio, text, and database), machine understanding, user modeling, and very large-scale text analytics. Dr. Gruhl is the chief architect for WebFountain, with responsibility for overall hardware, software, and systems design. He is also co-architect of IBM's Unstructured Information Management Architecture.

Laurent Chavet *Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052 (fsort@fsort.com).* Prior to joining Microsoft, Mr. Chavet worked on the WebFountain infrastructure at the IBM Almaden Research Center. His expertise in all aspects of search technology comes from his work at Alta Vista and with WebFountain. His primary interests are design and implementation of large-scale high-performance applications. He received three M.S. degrees: two from Ecole Polytechnique France in computer science and mathematics and one from E.N.S.E.E.I.H.T. France in computer science.

David Gibson *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (davgib@us.ibm.com).* Dr. Gibson is a researcher at the Almaden Research Center, where he has been involved in several projects related to Web mining and visualization, including early experiments with the HITS link analysis algorithm, browser enhancements with the WBI (Web Intermediaries) intelligent proxy project, and the SemTag Web-scale semantic annotation project. Currently he is working on WebFountain, an IBM initiative to provide Web data mining to corporate customers. He has a Ph.D. from the University of California, Berkeley.

Joerg Meyer *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (jmeyer@almaden.ibm.com).* Mr. Meyer is a software engineer at the Almaden Research Center. He has worked on projects involving Web application programming for Web Intermediaries (WBI), XML transcoding, and P2P. For WebFountain, he is responsible for various modules within the WebFountain indexer. He holds a Diploma Engineering degree in computer science from the University of Applied Sciences in Hamburg, Germany.

Pradhan Pattanayak *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (ppattana@us.ibm.com).* Mr. Pattanayak is a software engineer at the Almaden Research Center. His interests include parallel processing, distributed computing, and Web technologies. He has worked at IBM, Hewlett-Packard Company, Oracle Corporation, and the Centre for Development of Advanced Computing (CDAC). He received his Master of Technology degree from the Indian Institute of Technology, Bombay.

Andrew Tomkins *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (tomkins@almaden.ibm.com).* Dr. Tomkins is a member of the Principles and Methodologies group at the Almaden Research Center. His research interests include algorithms, particularly on-line algorithms, disk scheduling and prefetching, pen computing and OCR (optical character recognition), and the World Wide Web. He is one of the founders of WebFountain. He received his Ph.D. from Carnegie Mellon University and his B.S. in mathematics and computer science from the Massachusetts Institute of Technology.

Jason Zien *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (jasonz@almaden.ibm.com).* Dr. Zien has been at the Almaden Research Center since 1997, working first on a Java-based thin server platform for the Open Services Gateway Initiative (OSGI), then on WebFountain, and now on the TREVI search engine. His research interests include graph partitioning, scalable text indexing, search, and information retrieval. He received his Ph.D. in computer engineering from the University of California, Santa Cruz and his B.S. in computer engineering from Carnegie Mellon University.