WebStore: Efficient Storage and Access of the Web

Daniel Golovin, Sandeep Pandey

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{dgolovin, spandey}@cs.cmu.edu

Abstract

Research on enormous networks such as the web often suffers from a lack of appropriate tools to exploit the tremendous amounts of available data. Since networks such as the web are not centrally planned and are undergoing constant evolution that is intimately linked to economic and social developments, it is not surprising that research into their structure and ongoing evolution often more closely resembles some areas of social science than traditional algorithmic or systems research. As in the social sciences, sophisticated tools for data analysis are required. Towards this end, we present an extensible platform, called WebStore, for storing multiple time snapshots of huge networks that supports excellent compression and fast execution of several access primitives.

1 Introduction

In recent years, there has been much investigation into the structure and evolution of the web and other large evolved networks [2, 3, 4, 12, 17, 20]. Discoveries such as the heavy-tailed distribution of node degree in such networks have led to many practical ideas. These include ideas for improved routing protocols [20], improved crawling strategies for search engines [9], and further understanding of the spread of various phenomena (e.g. biological diseases, computer viruses, new technologies, gossip, destructive voltage spikes and other cascading failures) in various networks (e.g. social, computer, and electrical) [3, 18, 19, 20, 24], with applications to fighting epidemics, speeding the adoption of new technologies, and efficient viral marketing, to name a few. In light of the above, the study of network evolution and structure is likely to be a significant research area for years to come.

Despite the impressive progress made in research on massive evolved networks, such research is still in a very early stage. Only very recently have some standardized tools been built to aid in the analysis of these enormous datasets. See section 2 for details. Typically, when designing such a system, the designer must consider

- Target use of the system
- Scalability
- Storage methods and compression
- Access modes
- Support for updates

Existing systems vary in the tradeoffs they provide. We focus on providing support for research into web growth and structure. Thus, WebStore aims to achieve scalability, excellent compression for temporal sequences of web snapshots, and fast support for basic access modes commonly used in research – at the expense of sophisticated support for concurrent updates.

2 Related Work

As research into the growth and structure of massive networks progressed, researchers developed several tools to aid them [5, 8, 11, 13, 15, 21, 23]. Much work has been done by search engine developers, who have investigated ways of storing vast indexes of the web that support fast and accurate search, and crawlers that keep the index recent and relevant. Much of this work is proprietary, and is carefully guarded intellectual property, however Brin and Page provide a good glimpse into this area of research (circa 1998) [6]. From the perspective of the WebStore project, the drawback of this work is that it is focused on maintaining current indices which support extremely limited search capabilities. Researchers must be able to work with a sequence of temporal snapshots of the web, and must be able to issue queries other than those available to public search engine users.
On the opposite end of the spectrum, conventional databases offer extensive querying capabilities. However, they are inappropriate for the network data under consideration. It would be extremely inefficient to execute such queries as “count the number of nodes at distance less than $k$ in the network” in a database system. Furthermore, it is unclear how compression of the data could be implemented efficiently. Since the datasets WebStore must handle will be several terabytes, using a database system would be slow and unwieldy at best.

Since both search engine indices and conventional databases were insufficient, other tools were built. See the appendix A for details. However, none of these tools is a suitable alternative to WebStore as a platform for research into the growth and structure of large evolved networks.

3 WebStore Architecture

3.1 Storage

WebStore aims to support a small set of fundamental access modes on temporal sequences of network data with very high performance, in a way that scales well indefinitely. To this end, intelligent storage is critical. Both the link graph $G_w$, representing the structure of the Web, and the content of the web (the HTML documents) must be stored. In general, WebStore will be appropriate for studying any network with information associated with the nodes of the network. Because the networks under consideration are so large, compression becomes a key factor in scalability. Thus we find that compression of the web is a well studied problem in an of itself [1, 14, 22, 25], and in the context of work on search engines [5, 23]. In the latter context, it has been shown that clever compression schemes can actually increase performance while decreasing storage costs.

3.1.1 Storage of Link Graph

To store the link graph, we build upon the reference encoding strategy of Adler and Mitzenmacher [1]. Their idea is as follows. First, for each pair of pages $p, q$, they consider using a page $p$ as a reference for page $q$, that is, they consider encoding $q$ relative to $p$. Let $\Gamma(u) = \{v| (u, v) \in E[G_w]\}$ be the set of pages $v$ points to, and let $N(u) = |\Gamma(u)|$. Then we can associate a bit vector of length $N(p)$ with $q$, denoting those pages pointed to by both $p$ and $q$. We further associate an identifier for $p$ with $q$, as well as identifiers for all pages $v$ such that $q$ points to $v$ but $p$ does not. Thus using $p$ as a reference for $q$ will require $c(p, q)$ bits, where

$$c(p, q) := N(p) \cdot \lceil \log n \rceil \cdot (|N(q) - N(p)| + 1)$$

Using $c(\cdot, \cdot)$, Adler and Mitzenmacher then define the Affinity graph, which is the complete graph on $V[G_w]$ with edge $(u, v)$ weighted by $c(u, v)$, and compute the minimum cost arborescence (rooted directed spanning tree), $T$, on it, using a dummy node as the root. The root of $T$ is then stored, along with the reference encoding of each nonroot $p \in V[T]$ relative to its parent in $T$. Decoding a node relative to its reference, which we call dereferencing is straight-forward. To decode a node $p$, we must dereference each node in the $p$ to root path, working from the root down. Various improvements come to mind, such as computing the cheapest spanning tree of bounded depth, to bound the number of reference decodings required to decompress a node, or using multiple references per node to improve compression ratios. Unfortunately, unlike the standard version, these variants of min-cost arborescence are $NP$-hard. However, various heuristics are plausible for making this compression scheme amenable to computation on the compressed form of the data.

Considering that WebStore is designed for a sequence of snapshots of the web, the reference based encoding scheme is a natural choice. Let $p_t$ be the page version $p$ at time $t$. In the vast majority of cases, page $p_t$ will be an excellent reference for page $p_{t+1}$. We realize such efficiency by dividing the data into chunks called columns. Each column will reside in a separate file. Let $G_t$ be the link graph snapshot for time $t$. A column is specified by its temporal range, $[t_1, t_F]$, and its region, $R \subseteq \bigcup_{t=t_1}^{t_F} V[G_t]$. It contains a version of every node $v \in R$ for each snapshot $t$, $t_1 \leq t \leq t_F$, in which $v$ exists. Let region($C$) denote column $C$’s region, and range($C$) its range. Each column has a base layer storing its entire region in reference encoding format. This base layer does not correspond to a snapshot, in order to simplify the handling of pages being added and deleted in the web over time. If each page $p$ in the base layer is labeled $p_0$, then $p_t$ will use $p_r$ as a reference, where $r$ is the maximum value in
{0, 1, …, t − 1} such that \( p_r \) exists. The elements in the base layer form an rooted arborescence (created as though \( \text{region}(C) \) where the whole network). We call the base layer forming a rooted arborescence a checkpoint for its column’s region. Determining when and how often to checkpoint will be discussed in section 3.4. Depending on the size of a column’s region, it may be too expensive to compute the minimum cost arborescence (which requires \( O(n \log n + \sum_v (\delta(v))^2) \) time \([1]\), where \( \delta(v) \) is the in-degree of \( v \)). In this case, we can partition the vertices of \( \text{region}(C) \), and compute the minimum cost arborescence for each partition, and use these arborescences in place of one large arborescence.

WebStore creates columns on initialization (upon receiving one or more snapshots as input) and during checkpointing, using much the same process in both cases. As such all columns have temporal ranges from one checkpoint to the next. To create the columns, WebStore partitions the first time snapshot after the current checkpoint marker (or simply the first snapshot on initialization). The user provides parameters for column size in memory, and the desired column width (i.e. the size of its region) to height (i.e. the size of its temporal range) ratio. The former will depend on hardware, while the latter depends on the focus of the research – whether the researcher wishes to focus more on long term growth patterns, or structural features within a narrow time frame. Based on these parameters, during initialization the system determines a desired column region size, and will try to partition the vertices \( V := \bigcup_i V[G_i] \) appropriately. While balanced cuts are hard to compute in general, web graphs and other massive evolved networks are typically very sparse (often with constant average degree). Therefore a good partitioning will be easy to find. In \([22]\), a partitioning scheme based on URL similarity is presented. In it, pages are initially partitioned by domain, e.g. pages with URLs starting with “www.cmu.edu” form one partition, those starting with “www.us.gov” another, and so on. These are then refined upon as needed by using longer URL prefixes, e.g. splitting the “www.cmu.edu” partition into “www.cmu.edu/research/”, “www.cmu.edu/alumni/” , etc. Alternate strategies that will also work for other large networks include various clustering algorithms, such as using the MST algorithm in \([16]\) for hierarchical clustering, or various sophisticated graph cutting procedures, such as those in \([7]\). After partitioning, WebStore maintains a mapping from vertices to columns, in a hash table. When checkpointing, WebStore will attempt to use the partition created on initialization to define regions for the new columns. However, some pages may exist in the current snapshot that did not exist before. These will be assigned to the existing partition most appropriate for it, unless the user requests that a new partitioning be computed as in initialization.

### 3.1.2 Storage of Page Content

In addition to the link graph, WebStore stores the content of the pages. As in the case of the link graph, we seek a compression scheme that balances speed of access and compression ratio. There are two major design points, namely the storage of each version of each page content, and the placement of pages on the disk.

With regard to storing multiple versions of a page’s content, there are several possibilities. Let \( W_1, W_2, …, W_k \) be the versions of one pages content, let \( \Delta(A, B) \) be the difference between \( A \) and \( B \), as in the Unix “diff” command, and let \( X_1, X_2, …, X_k \) be the data we store.

1. We may store a copy of each version: \( X_i = W_i \).
2. We may store a copy of the first version (a base copy), and differences from it for each later version: \( X_1 = W_1, (\forall i > 1)(X_i = \Delta(W_1, W_i)) \).
3. We may store differences between successive pages: \( X_1 = W_1, (\forall i > 1)(X_i = \Delta(W_{i-1}, W_i)) \).

In addition to these three options, there are several others involving storing the differences in differences. In practice however, we have found that options #4 and #5 described below perform worse than #2 and #3, respectively, and include them for completeness only:

4. \( X_1 = W_1, (\forall i > 1)(X_i = \Delta(X_{i-1}, \Delta(W_1, W_i))) \).
5. \( X_1 = W_1, (\forall i > 1)(X_i = \Delta(X_{i-1}, \Delta(W_{i-1}, W_i))) \)

Option #1, which merely stores the uncompressed data, is infeasible. Options #2 and #4 offer reasonable compression, and require at most one or two calls, respectively, to a “patch” function to decompress any page. Options #3 and #5 offer the best compression, but require up to \( k \) or \( k + 1 \) calls, respectively, to a “patch” function to decompress any page. To achieve desirable decompression speed, we use option #2.

In conjunction with the strategy for storing multiple versions of a page’s content, WebStore must intelligently organize data on disk. In addition to achieving a good compression ratio while keeping the amount of work to compute on the compressed data low, we must balance two competing design goals:

**Spatial locality:** Content for nodes close together in the network should reside close together on disk.

**Temporal locality:** Multiple versions of a page’s content should reside close together on disk.

Our solution is as follows. For a page \( p \), let \( X_1^p, X_2^p, …, X_k^p \) be the data containing (the compressed form of) each of the \( k \) versions of \( p \)’s content.
For each column \( C \), for each time \( t \in range(C) \), we maintain a directory containing \( |region(C)| \) files, one for each \( p \in region(C) \). These files contain \( X_t^p \) for each \( p \in region(C) \). We then compress each such directory into a single file, to occupy a contiguous portion of the disk. Note that the problem of achieving spatial and temporal locality still exists to some extent, with the focus moving from individual pages to entire regions. Let \( dir(C, t) \) denote the directory containing \( X_t^p \) for each \( p \in region(C) \). We achieve some temporal locality by placing \( dir(C, t + 1) \) directly after \( dir(C, t) \). The remaining disk placement is done arbitrarily. Note that better performance could possibly be achieved by using the techniques of [10] to embed a metric approximately representing the distance between regions into a line metric. Treating the disk as a large one dimensional array, this yields a disk placement with good spatial locality.

### 3.2 Access Modes

WebStore provides fast support for a few important access modes. By providing extremely fast support for these fundamental queries, WebStore can serve as a foundation for more sophisticated query clients. The supported access modes are as follows:

- Random access of the links and/or content of a page \( p \) at time \( t \).
- The link history and/or content history of a page \( p \) over some time range.
- The links and/or content of all pages within distance \( d \) from page \( p \) at time \( t \).
- The link and/or content history of all pages within distance \( d \) from page \( p \) at time \( t \).

WebStore also provides a **load** function for placing a page’s column into memory, and a **decompress** function to decompress a snapshot of a column’s region and store it in memory. A decompressed snapshot can serve as a resident checkpoint for later versions, dramatically speeding up queries on it and later snapshots in the column. To decode the references for later versions of a page will take constant time, provided one checkpoints after a constant number of snapshots, offering excellent compression of all later snapshots with nearly the same speed as computing on the uncompressed form. The **load** and **decompress** functions therefore make WebStore an excellent storage subsystem for arbitrarily complex data mining applications for large networks.

### 3.3 Updates

In the pursuit of superior compression ratios and access speeds, WebStore compromises on updates, providing only modest capabilities. We do not allow additions or deletions of individual pages to the system, however arbitrary modifications to may be made to memory resident copies of the data if desired. In practice, researchers do not need support for full database style transactions to study network structure and growth phenomena. WebStore thus provides support for adding a new snapshot or a series of snapshots, and for deleting a snapshot or a series of snapshots.

To add a snapshot, WebStore first determines if this will be a checkpoint or not. If so, it determines if this snapshot requires a partitioning of the vertices to generate new column regions. This occurs only upon initialization or at the user’s explicit request. Once a partition is generated as described in section 3.1.1, a base layer is created using the reference encoding strategy of [1], and pages corresponding to the current snapshot reference their base layer versions. If a repartitioning is not required, pages that have no earlier versions are added to the partition that would most maintain spatial locality, e.g. the partition from which most of its neighbors belong. This will require the addition of a base layer version of such pages to the base layers of their columns, enlarging \( region(C) \). This is one advantage of maintaining a base layer not corresponding directly to any particular snapshot. For the content, a new directory is created for the current snapshot, as described in section 3.1.2.

If the snapshot is not to be a checkpoint, then we must modify the partitions and \( region(C) \) for each column \( C \) as before to handle newly created pages that are in the current snapshot but have no earlier versions. Recall that \( p_t \) denotes the version of \( p \) at time \( t \). Let \( p_0 \) denote the base layer version of \( p \). Then for the new snapshot \( t \), for each \( p \in region(C) \), \( p_t \) references \( p_r \), where \( r \) is the maximum value in \( \{0, 1, \ldots, t - 1\} \) such that \( p_r \) exists. This reference encoding is then appended to the file containing column \( C \). The content is compressed via one of the schemes in 3.1.2, typically resulting in “diff” style output files, which are then stored in a new directory created for the current snapshot, to be compressed again via a general compression algorithm.

Deletion of a snapshot is straight-forward. We simply alter the references above it and delete the vertices. That is, if we wish to delete snapshot \( t \), and a page \( p_t \) references \( p_a \) and is referenced by \( p_b \) (so \( a < t < b \)), we must compute the encoding of \( p_b \) using \( p_a \) as a reference and delete \( p_t \), for each \( p_t \) in the snapshot. The base layer of a column is deleted automatically when every snapshot in that column’s temporal range is deleted. Since columns always have ranges between checkpoints, if this occurs it will occur for a large set of columns at once.

### 3.4 Checkpointing

Checkpointing strategy for the link graph provides a way for the user to trade space for speed and vice-versa. To obtain the links of a node \( v \), we must decode the reference encoding for every page on the path from
v the root of the arborescence that contains it. This path can be divided into two parts: the path from v to its version in the base layer, v₀, and the path from v₀ to the root of the arborescence. The latter can be shortened in numerous ways, but at the cost of a worse compression ratio. For example, we could iteratively cut all edges (u, v) such that v was at distance d from the root r, and add edges (u, r) in their place. The former, however, can only be shortened if there are frequent base layers – that is, if checkpointing occurs frequently. Ultimately, Webstore allows the user to specify checkpoint frequencies for link information.

Checkpointing strategy for content offers a similar tradeoff if the user chooses to store differences between successive pages (scheme #3 in section 3.1). However, by default WebStore uses diff from the base version (scheme #2). In this case, increasing checkpointing frequency can decrease storage requirements if major changes in content are occurring. WebStore provides an adaptive checkpointing scheme for content. If enabled, it will checkpoint once the diff file grows larger than some fixed percentage p of the size of the uncompressed version. Alternately, it can be set to checkpoint after a fixed number, f, of snapshots. Parameters p and f are user specified.

4 Results

We conducted experiments to measure the performance tradeoffs involved in various design decisions presented above. The experimental setup and results are presented below.

4.1 Compression of the Link Graph

For link graph compression, we report results for one column, created from a subset of the web containing 2725 pages selected from a single large website. We experimented on 15 snapshots, corresponding to weekly crawls between September and December 2003.

Looking at figure 2, we see the size increasing linearly in the number of snapshots. Uncompressed, a snapshot of the link graph requires 3Mb, or 1101 bits per page, while a reference encoded snapshot requires 0.54Mb, or 198 bits per page. Under the temporal reference encoding scheme, later snapshots require an average of 0.19Mb to store, or 69 bits per page. Our encoding scheme requires a base layer and an encoding of the first snapshot to store one snapshot, which requires 0.73Mb to store. Our encoding scheme thus achieves a compression ratio of 4.1 for the first snapshot, and 15.8 for later snapshots. Furthermore, our encoding achieves compression ratio 2.8 times better than simply reference encoding each snapshot individually, justifying the underlying design of WebStore’s link graph storage scheme.

4.2 Compression of the Content

We report results from the same 2725 page subset of the web used to generate the column for the link graph compression results reported above. As before, we experimented on 15 snapshots, corresponding to weekly crawls between September and December 2003. The content for each snapshot of this portion of the web is approximately 68MB in size, for a total of 1GB of data. We then compressed the data using each of the 4 diff schemes described in section 3.1, and compared the compressed size against the 1GB uncompressed size. The “diff of diff” schemes performed poorly, with scheme #4 requiring more storage than the uncompressed data, and scheme #5 requiring nearly as much. For schemes #2 and #3 we show the sizes of the diff of different snapshots in figure 3. As expected under scheme #2 the size of diff increases with the number of snapshots after the last checkpoint. While
under scheme #3 the size of diff always remains small, since two consecutive copies of pages are expected to be similar to each other. As mentioned before scheme #3 offers better compression, but significantly greater cost to decompress the data. Hence, we use scheme #2 in WebStore, with the option of checkpointing more often in order to get better compression. In figure 4 we show the sizes of compressed data for different number of checkpoints, using a 1GB dataset. As one can see from the figure, checkpointing twice in 15 weeks achieves the best compression ratio. Note that beyond a certain threshold the number of checkpoints starts degrading the compression performance. This happens because the benefit achieved by a checkpoint, in terms of space, becomes smaller than the extra space consumed while creating the checkpoint.

4.3 Access Modes

For the access modes, we do a theoretical analysis. This is in light of the fact that performance will depend heavily on the underlying storage media, the (possibly distributed) filesystem upon which WebStore will operate, and the size of the data. Thus, isolated wall clock times for a particular hardware setup would not be very helpful. Additionally, since there are no systems directly analogous to WebStore, direct wall clock comparisons of WebStore to other systems is infeasible for data containing multiple snapshots of the web.

The cost of loading a column into memory is simply the cost of loading its file into memory. For simplicity of exposition, let time zero correspond to the latest checkpoint before \( t \). To obtain the content for page \( p_t \in C, \, \text{dir}(C,t) \) and \( \text{dir}(C,0) \) must be decompressed (recall \( \text{dir}(C,t) \) are compressed via a general compression algorithm), and then patched with the base version. This requires a single patch operation. Comparable schemes storing each version of a page together in a compressed file would require at the minimum a decompression of one file and a patch. WebStore achieves much better spatial locality at the cost of decompressing two files. To obtain the content history of a page over a time range, we must load and decompress the files containing \( \text{dir}(C,0) \) and \( \text{dir}(C,t) \) for each \( t \) in the desired time range, reducing the average cost of accessing one version of a page’s content. Obtaining the content for an entire region necessitates the patching of each page in the region rather than only one. Since patching in memory is faster than loading the file into memory and decompressing it, this is only slightly more costly than obtaining the content for a single page.

With regard to the link graphs, the cost of decompressing a snapshot layer \( t \) in a column, after loading the column, is that of decoding the base layer, and decoding each page in layer \( t \) relative to it. Decoding the base layer requires \(|\text{region}(C)|\) deference \((p_0, q_0)\) operations, and each deferencing operation will be very fast since it requires one memory lookup and one memory write for each link in \( p_0 \). Decoding \( p_t \) relative to \( p_0 \) is similarly very fast, requiring \( t \) deference operations. A similiar analysis shows the cost of a random access of a node’s links to be to be the cost of a hash table lookup, to determine the corresponding column, plus \( k \) deference operations, where \( k \) is the length of the page to root path. Fortunately, due to “small world phenomena” in most large evolved networks, \( k \) is likely to be small on average [26] – not much more than the expected diameter of \( O(\log |\text{region}(C)|) \). As mentioned in section 3.1, if \( k \) is too large, we can alter the arborescence used for the reference encoding, using several heuristics. Note that accessing the link history of a page takes the same cost as accessing the most recent copy. Finally, accessing a set of pages costs roughly the cost of accessing each page individually, with possible savings from memoization of the dereferenced nodes in the base layer.

5 Conclusion

Research into the structure and growth of large evolved networks such as the web require new analytic tools to effectively leverage the huge amounts of available data. As shown in section 2, previous systems only partially address this need. For example, web search related research tools tend to focus only on the current version of the web, and do not allow for the study of network evolution. Many other systems provide scalability, but only limited support for data mining. Finally, those that supply extensive data mining support are not scalable.

The WebStore system provides an excellent storage platform upon which the needed analytic tools can be built. It achieves scalability, excellent compression for temporal sequences of web snapshots, and fast support for basic access modes commonly used in research – at the expense of sophisticated support for concurrent updates. It thus provides researchers with two approaches. They can build data mining clients on top of WebStore that translate high level queries to
the fast access primitives WebStore provides. Alternatively, they can use WebStore as a storage subsystem for arbitrarily complex data mining applications for large networks, using primarily the load and decompress functions. Used either way, WebStore represents an invaluable new tool for research into large evolved networks.

References
Appendix A
ANF system

The ANF system [21] aids research into network connectivity by quickly computing the approximate neighborhood functions (the ANFs) of a graph. The (generalized) individual neighborhood function for node \( u \) at distance \( k \) given \( C \subseteq V \), \( \text{IN}^+(u, k, C) \) is the number of nodes in \( C \) at distance \( k \) or less from \( u \): \( \text{IN}^+(u, k) = \{ v : v \in C, \text{dist}(u, v) \leq k \} \). The (generalized) neighborhood function for node \( u \) at distance \( k \) given \( S, C \subseteq V \), \( N^+(u, k) \) is the number of nodes \( u \in S \) at distance \( k \) or less from some \( v \in C \): \( N^+(k, S, C) = \{ (u, v) : u \in S, v \in C, \text{dist}(u, v) \leq k \} \). Usually, one is interested in the standard versions of the individual and standard neighborhood functions, where \( S = V \) and \( C = V \). The ANF system boasts an algorithm that provides provable accuracy with high confidence, can be parallelized, has low memory requirements, and is fast (over 700 times faster than the fastest exact computation for the internet graph on 268K nodes used by the authors). It is used to investigate such questions posed by the authors as

1. How robust is the Internet to failures?
2. Is the Canadian internet similar to the French?
3. Does the internet have a hierarchical structure?
4. Which set of street closures would least affect traffic?

While the ANF system represents major progress in investigations on network connectivity, it can only provide part of the picture. Clearly, the network cannot be completely characterized by its neighborhood functions. Other tools are thus necessary to fully understand these networks, and the ANF system therefore serves to complement rather than replace other tools.

The Connectivity Server

The Connectivity Server [5] is a tool for studying the topology of the web. It provides fast computation of all pages pointing to a page \( p \) and/or all pages \( p \) points to, along with visualization tools for so called neighborhood graphs. Like WebStore, it was designed as a platform for numerous applications. Among those listed by the authors are ranking, visualization, and classification. Unlike WebStore, it focuses only on current network structure and does not offer support for studying network growth phenomena.

GoogleFS

The Google File System [13] is a “scalable distributed file system for large distributed data-intensive applications.” Google Inc., one of the world’s leaders in search engine technology, is naturally interested in studying the structure of the web. GoogleFS was designed to support the storage needs within Google, both to run its search engine and to support “research and development efforts that require large datasets.” One may safely assume that these large datasets include parts of the web, and so it is not surprising that GoogleFS would be useful as a storage subsystem for a research tool into the structure and growth of massive networks. GoogleFS supports redundancy, fault tolerance, and excellent performance and scalability. It has a simplified consistency model relative some other distributed file systems, and focuses on providing efficient writes in only one case: that of appending to a file. Random writes are supported, but are slow. We will take a similar approach to updates in WebStore. However, GoogleFS is still a file system rather than a research tool. In the case of temporal sequences of snapshots of the web, there are properties of the data that a special purpose system such as WebStore can exploit that any general platform for dealing with enormous datasets, such as GoogleFS, can not.

WebBase

WebBase [15] is a repository for web pages developed at Stanford. An early version of WebBase was used as a backend storage system for the Google search engine. While also developed as a research platform, WebBase focuses on keeping a fresh copy of the Web indexed, and thus falls under search engine related work disjoint from WebStore.

WebGUIDE

WebGUIDE [11] supports recursive comparison of HTML documents. Unlike other systems, it is designed to study changes in the web over time. However it is designed to track only pages specified by the user, supports only very limited queries, and is not scalable.

WebQuery

WebQuery [8] is a system that provides a search interface which allows the user to visualize web structure and hopefully exploit it to achieve better search results. While this makes it a better research tool than standard search engines, it does not provide a rich enough set of access modes for researchers.