The Scalable Hyperlink Store

Marc Najork
Microsoft Research
Mountain View, CA, USA
najork@microsoft.com

ABSTRACT

This paper describes the Scalable Hyperlink Store, a distributed in-memory "database" for storing large portions of the web graph. SHS is an enabler for research on structural properties of the web graph as well as new link-based ranking algorithms. Previous work on specialized hyperlink databases focused on finding efficient compression algorithms for web graphs. By contrast, this work focuses on the systems issues of building such a database. Specifically, it describes how to build a hyperlink database that is fast, scalable, fault-tolerant, and incrementally updateable.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*distributed databases*

General Terms

Performance, Reliability

Keywords

Web graph, hyperlink database, scalability

1. INTRODUCTION

One of the defining characteristics of hypertext is the existence of explicit hyperlinks between documents. These links are intended to allow users to navigate from one document to another, allowing them to explore related concepts and holding out the promise of serendipitous discovery of new information.

Web search engines have relied on hyperlinks from the very beginning: the web crawlers used by search engines are seeded with a small or medium-sized set of starting URLs, download these pages, and use the hyperlinks contained in the downloaded pages to continue the process [20]. But hyperlinks are useful to web search in many ways beyond crawling. For example, hyperlinks can be viewed as peer-endorsements: a link from page a to b tacitly suggests that the author of page a endorses the content of page b. Highly endorsed pages are deemed to be more authoritative, and therefore should be ranked more highly than less-endorsed pages. This basic idea is rooted in citation analysis used in bibliometrics. It was first transferred to the web setting by Marchiori [14], who suggested to simply keep count of the hyperlinks referring to a page, and subsequently developed into more and more sophisticated link-based ranking algorithms. Three of the best-known examples of this family of ranking algorithms are PageRank [22], HITS [11] and SALSA [13]. The first algorithm computes a query-independent measure of the "quality" of each page on the web, while the latter two compute a measure of the relevance of the results with respect to a query. Consequently, the latter two algorithms inherently have to be executed at query-time, and thus have to be quite fast.

Search engines have many other uses beyond crawling and ranking for the hyperlink structure between web pages: for example, the link structure can be used to detect communities of related web pages [12], and can be used to identify "spam" web pages [2].

Some of the algorithms operating over the web graph – the graph induced by the web pages and their hyperlinks – require only a very regular access to the edges of the graph, and therefore lend themselves to implementations where the links are read in a streaming fashion from disk. Modern hard disks are very well-suited for such streaming access patterns: their latency (the time required to position a disk head over a particular block of data) is high, on the order of 5–10 milliseconds, but their bandwidth (the amount of data that can be read after the head has been positioned) is quite large, on the order of 50–100 MB/sec. So, algorithms that lend themselves to a streaming consumption of links (*e.g.* PageRank) can be implemented efficiently using disk-based storage of the web graph; on the other hand, algorithms that inherently have random access patterns (such as HITS and SALSA) are not well-suited for disk-based implementations. Performing them even remotely efficiently requires storing the web graph in main memory. RAM has higher bandwidth and much lower latency than disks; unfortunately, it is also two orders of magnitude more expensive per byte, and contemporary commodity systems are limited (by their memory management units and the number of physical DIMM slots) in how much RAM they can hold — typically under 100 gigabytes, as opposed to the terabyte capacity afforded by even a single large hard drive.

The need for fast random access to nodes and edges in the web graph has led to a number of research systems; the memory constraints have also inspired a number of more theoretical inquiries into specialized compression algorithms for the web graph. The first such system we are aware of was the Connectivity Server [3], developed at DEC SRC. The Connectivity Server maintained an in-memory representation of the web graph on a single machine. It compressed the graph by mapping URLs to integer shorthands, storing the edges of the graph using these shorthands, and compressing URLs by sorting them lexicographically and eliding shared prefixes using delta-encoding. It did provide for a limited number

HT'09, June 29–July 1, 2009, Torino, Italy.

of updates to the store, but once the number of updates exceeded a certain threshold, the database had to be rebuilt from scratch.

Suel and Yuan's system [24] adapted a key idea of the Connectivity Server, namely to maintain a mapping between URLs and integer shorthands, and to represent the incoming and outgoing edges of each vertex in the graph as two lists of integer shorthands. Their system improved on the Connectivity Server by adopting more aggressive compression techniques. In addition to using prefix elision to shorten URLs, Suel and Yuan's system also used Huffman codes to compress the URL suffixes. In order to compress the adjacency lists, they first divided the vertex set into highly popular pages (web pages with many incoming links) and less-popular pages, and compressed the links using either Golomb or Huffman codes. Their system did not provide for updates to the web graph, although they mentioned accommodating updates as a future-work item.

The Link Database [23], also developed at DEC SRC, was a direct descendant of the Connectivity Server. It improved on the Connectivity Server by compressing the edge set using a variety of techniques: gap-encoding to convert the integer shorthands representing vertices into small values, nybble encoding or Huffman encoding to compress the gap values, and inter-list compression to exploit link-similarity between web pages. Unlike the Connectivity Server, the Link Database provided no mechanism for updates at all; the model was that the database would be periodically rebuilt from scratch using snapshots of the web graph.

The WebGraph Framework [4, 5] is another system building on the basic design of the DEC SRC Connectivity Server. The focus of this work is on aggressive compression of the adjacency lists used to represent the incoming and outgoing edges of each vertex. Like the previous systems, WebGraph maintains a mapping from URLs to integer shorthands, and like the Link Database it uses gap encoding to convert the shorthands to numbers with small absolute values, and inter-list compression to exploit redundancy between vertices with similar linkage. But instead of nybble or Huffman encoding, it uses arithmetic codes to compress the integer shorthands. Like Suel & Yuan's system and the Link Database, WebGraph makes no provision for incremental updates to the web graph, and like the previous systems, it runs on a single machine.

All of the above systems are non-distributed systems that store a compressed version of the web graph in main memory; consequently, they were not able to (or indeed intended to) scale up to web graphs with tens of billions of nodes and hundreds of billions of edges, such as the ones induced by the corpus of state-of-the-art commercial search engines. The goal of many of these earlier systems was to explore web graph compression techniques, and when considering time-space tradeoffs, they opted for high compression rates over fast decompression. Moreover, because they were tested on static reference collections of web pages (such as the Stanford WebBase), they largely ignored the issue of incremental updates.

The system described in this paper complements this earlier work. The primary focus is on addressing the systems issues that go along with integrating a web graph storage system into a large-scale search engine: allowing the store to scale up to arbitrary sizes by partitioning it over many machines; dealing with machine failures that will invariably occur in a distributed system; allowing for incremental updates to the web graph (in lock-step with the search engine's crawler); and making it possible to enlarge the cluster without interrupting service, to accommodate an ever-growing corpus. The compression techniques used in our system are not particularly inventive, following the basic ideas espoused in the earlier systems, but using less aggressive coding schemes, thereby choosing a point on the time-space tradeoff curve that is more biased toward decompression speed than space savings.

2. DESIGN GOALS

We designed the Scalable Hyperlink Store as a general-purpose system for performing graph algorithms on the full web graph, but we were particularly interested in enabling query-time link-based ranking algorithms along the lines of HITS [11] and SALSA [13]. In order to enable such algorithms, SHS has to satisfy multiple, often conflicting design goals:

1. *The system has to be fast enough to enable real-time execution of HITS-like link-based ranking algorithms.* Given the number of edges in a HITS neighborhood graph, this rules out any disk-based solution, as the seek times of modern hard drives are too slow by several orders of magnitude. To achieve the required performance, we have to maintain the web graph in main memory.
2. *The system has to employ graph compression techniques.* Given the decision to maintain the graph in main memory, and given that a given amount of RAM is about a hundred times more expensive than the same amount of disk, we should aggressively try to compress the graph. However, compression techniques tend to be computationally more expensive the more aggressive they are, so we have to choose the appropriate trade-off between memory footprint and decompression cost.
3. *The system has to scale to the full web graph.* Given that we decided to keep the graph in main memory, and given that the maximum memory size of commodity machines is limited to a few tens of gigabytes, this means that we have to partition the graph over multiple machines.
4. *The system has to be fault-tolerant.* The probability of at least one machine failing in a distributed system increases with the number of machines in the system. If we aspire to use our system as a production service, we certainly have to provide mechanisms to mask such failures.
5. *The system should permit incremental updates to the graph.* Search engines crawl the web continuously, to keep up with the creation of new pages and to maintain up-to-date snapshots of fast-changing pages. A hyperlink store should reflect the most recent linkage information available to the search engine. However, this goal complicates matters substantially. At the API level, we have to decide whether or not we want to provide transactional semantics, since the structure of the stored web graph might change while a computation using the graph is in progress. At the implementation level, the most space-efficient data structures for maintaining vertices and edges of the web graph are ill-suited for updates.
6. *The system should allow for the graph to be grown over time.* Given our choice of achieving scalability through distribution, this means that we should be able to grow the set of SHS servers without unduly interrupting the service.

The remainder of this paper will flesh out how we achieved these design goals, and where appropriate will provide a more detailed discussion of the design alternatives that were available to us and the trade-offs between them.

3. MAINTAINING STATIC WEB GRAPHS

SHS maintains the vertices and edges of the web graph in a distributed data structure called the *store*. An SHS store is composed of *cells*. Each cell maintains a set of vertices and all the edges that connect to each of the vertices. In order to achieve good compression, edges are not represented as pairs of URLs (*i.e.* strings).

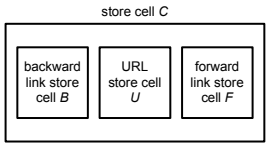

Figure 1: An SHS store cell.

Instead, each cell maintains a bijective mapping between textual URLs and numerical UIDs (unique identifiers), and edges are represented in terms of UIDs. The mapping between URLs and UIDs are kept in an *URL store cell*, the set of edges departing from each vertex in a *forward link store cell*, and the set of edges arriving at each vertex in a *backward link store cell*. Figure 1 shows these components of a cell.

The Client API

Client applications interface with SHS through a *clerk*, a piece of code that is linked into the client applications and facilitates all interactions with the SHS servers (see Figure 2). We provide a clerk implementation written in C++, and an alternative implementation written in C#. Both implementations provide a very similar API to client applications. The C# version looks as follows:

```
public enum ShsStore { UrlStore, FwdStore, BwdStore };
public class ShsClerk {
  public ShsClerk(string shsName) {...}
  public void Dispose() {...}
  public void Request(ShsStore s) {...}
  public void Relinquish(ShsStore s) {...}
  public int NumServers() {...}
  public long NumUrls() {...}
  public long NumLinks() {...}
  public int MaxDegree(bool fwd) {...}
  public long NumUrls(int serverId) {...}
  public long MinUid(int serverId) {...}
  public long MaxUid(int serverId) {...}
  public long NumLinks(int serverId) {...}
  public int MaxDegree(bool fwd, int serverId) {...}
  public long UrlToUid(string url) {...}
  public string UidToUrl(long uid) {...}
  public long[] BatchedUrlToUid(string[] urls) {...}
  public string[] BatchedUidToUrl(long[] uids) {...}
  public long[] GetLinks(bool fwd, long uid) {...}
  public long[] SampleLinks(bool fwd, long uid,
      int numSamples, bool consistent) {...}
  public long[][] BatchedGetLinks(bool fwd, long[] uids) {...}
  public long[][] BatchedSampleLinks(bool fwd, long[] uids,
      int numSamples, bool consistent) {...}
}
```

Clients create an instance of an ShsClerk object to connect to a particular SHS database. Creating the clerk establishes TCP connections to the SHS servers, but does not cause any stores to be loaded. Stores are loaded lazily on first use; this lazy behavior can be overridden by calling Request(s), which returns when all servers have loaded store s. Calling Relinquish(s) indicates to servers that the client does not need store s right now, and that s can be unloaded unless other clients are using it. Calling Request and Relinquish merely provides hints to the SHS servers, and has no effect on the correctness of the client program. The Dispose method is called at the conclusion of an SHS session; it closes the connections to the servers.

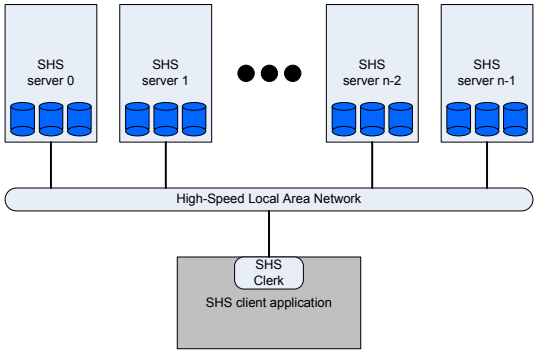

Figure 2: A client application connecting to a cluster of n SHS servers.

The zero-argument versions of NumServers, NumUrls, and NumLinks return the number of SHS servers, web graph vertices and web graph edges, respectively. MaxDegree(f) return the maximum out-degree (f=true) or in-degree (f=false) of any node in the graph. Variants of NumUrls, NumLinks and MaxDegree with an extra argument i indicating an SHS server return the corresponding result restricted to the portion of the web graph maintained by server i. Methods MinUid(i) and MaxUid(i) return the lowest and highest UID on server i. Taken together, the NumServers, MinUid and MaxUid methods enable clients to enumerate all UIDs of a web graph.

The UrlToUid method maps a URL to its UID, and the UidToUrl method provides the inverse functionality. Methods BatchedUrlToUid and BatchedUidToUrl allow clients to map large batches of URLs and UIDs, thereby allowing them to amortize the remote procedure call overhead (which is about 100 microseconds for a null RPC call in our cluster, using our custom RPC implementation).

Calling GetLinks(f,x) gets *all* forward- (f=true) or backward-links (f=false) associated with UID x. Calling SampleLinks(f,x,n,c) returns a sample of n such links (or all links if there are fewer than n). Sampling on the SHS server side greatly reduces network traffic. The sample is drawn uniformly at random if c is false, and consistently if c is true. Consistent sampling [6] is deterministic, unbiased, and it preserves the similarity between two sets that are being sampled. It is an extremely useful operation for numerous algorithms. Methods BatchedGetLinks and BatchedSampleLinks allow clients to look up the links of many UIDs at once, again to amortize the RPC overhead across many lookups.

The following short C# program illustrates how clients use the SHS API.

```
using System.Console;
public class ShsDemo {
  private static void List(long uid, ShsClerk shs, int dist) {
    if (dist == 0) {
      Console.WriteLine(shs.UidToUrl(uid));
    } else {
      long[] uids = shs.GetLinks(dist > 0, uid);
      for (int i = 0; i < uids.Length; i++) {
        List(uids[i], shs, dist > 0 ? dist - 1 : dist + 1);
      }
    }
  }
  public static void Main(string[] args) {
    if (args.Length != 3) {
      Console.WriteLine("Usage: ShsDemo <shsName> <dist> <url>");
    } else {
      ShsClerk shs = new ShsClerk(args[0]);
      long uid = shs.UrlToUid(args[2]);
      if (uid == -1) {
        Console.WriteLine("URL {0} not in store", args[2]);
      } else {
        List(uid, shs, int.Parse(args[1]));
      }
    }
  }
}
```

The program takes three arguments from the command line – the name of an SHS instance, an integer d, and a URL u – and prints all URLs that can be reached from u via d hyperlinks if d is positive, or that lead to u via $-d$ hyperlinks if d is negative. For reasons of

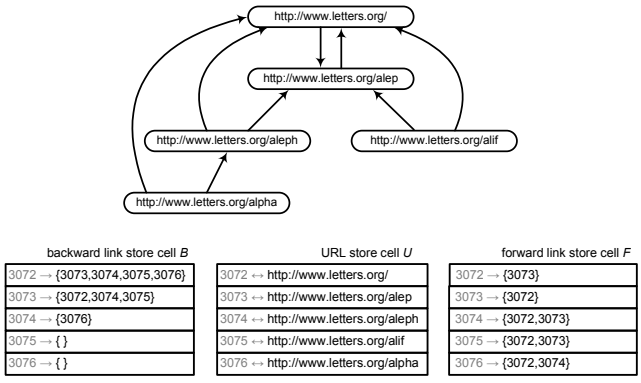

Figure 3: A small web graph and an abstract view of a store cell encoding that graph.

brevity, the program does not try to eliminate any duplicate URLs (*e.g.* URLs that can be reached from u via multiple paths).

URL store cells

UIDs are 64-bit integers, and the UIDs maintained by an URL store cell are drawn from a densely-packed space. If an URL store cell contains n entries[1] and the UID of the entry at position 0 is a, the UID of the last entry is $a+n-1$. The fact that UIDs are drawn from a contiguous range means that an URL store cell does not need to explicitly store them; the URL at position i in an URL store cell with starting UID a is implicitly equated with UID $a+i$. Likewise, if we view a forward link store cell as a mapping from a "key" UID u to a set of UIDs (the endpoints of all the edges that lead away from u), then it is not necessary to store any of the keys explicitly, it suffices to store the sets of destination UIDs. The analogous argument holds for backward link store cells.[2] Figure 3 shows an abstract view of a store cell with five vertices and their incoming and outgoing edges. The starting UID of this store is 3072. UIDs that are implied and thus don't need to be stored explicitly are grayed out. Some of the edges connect to vertices not maintained by this store cell; their UIDs are outside of the range from 3072 to 3076.

The URLs in an URL store cell are stored in lexicographically sorted order. This has two implications: First, all the URLs in a cell referring to the same web server ("host") are stored consecutively, and their associated UIDs form a range with no UID associated with a different host falling into that range. Second, adjacent URLs in the URL store cell will tend to have shared prefixes, and the larger the cell is (the more URLs it contains), the longer these prefixes will tend to be. This means that *front coding* [25, page 159ff] – omitting the prefix shared with the previous URL in the store and instead simply storing its length – can be used to reduce the size of the URL store cell. Our implementation represents each URL in the URL store cell by the length of the omitted prefix, followed by the length of suffix (using variable byte-length encoding to represent both numbers), and finally the suffix itself.

There are two variants of front coding: partial front coding, where every kth word is stored in full (and thus making it possible to decompress any word by examining at most the $k-1$ previous words), and *complete front coding*, where every word is front-coded.[3] Our implementation uses complete front coding. In order to be able to

[1]Throughout this paper, we use zero-based indexing.

[2]We will see later that things get more complicated once we allow for continuous updates to SHS.

[3]The first word is assumed to be preceded by the empty string.

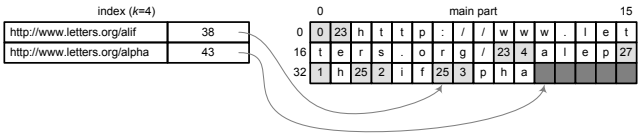

Figure 4: Concrete implementation of URL store cell from Figure 3.

map URLs to UIDs and vice versa without having to scan through the entire URL store cell, we maintain an *index data structure* in addition to the main part of the URL store cell. The main part of the cell is segmented into blocks of k URLs each. For each block, the index contains an entry capturing the *last URL* in that block and the starting position (a byte-offset from the beginning of the main part) of the *next* block. The index is organized as an array of uncompressed URL strings and offsets, *i.e.* its elements can be accessed at random in constant time. Figure 4 shows the representation of the URL store cell from Figure 3. White cells show ASCII (character) values; gray cells show numeric values. The arrows pointing from the index to the main part do not represent C pointers, they simply illustrate where offsets in the index refer.

Assume we are given an URL store cell with starting UID a. In order to map a URL u to its corresponding UID, we first perform binary search on the index to locate the entry at position b with the lexicographically largest URL u' in the index that is no larger than u (if there is no such b, we report that u is not contained in this cell). If $u=u'$, we can compute u's UID to be $a+bk-1$ and return it. Otherwise, we scan the main part starting at the position indicated by the index entry, and using u' as the starting value for undoing the front coding. If we find u at position i, we return its corresponding UID $a+bk+i$, otherwise, we report that u is not contained in this cell.

Mapping a UID v to a URL is even simpler. The URL will be contained in block $b=(v-a)\operatorname{div}k$ of the main part. If $b>0$, we retrieve URL u and offset o from position $b-1$ of the index, otherwise we set u to the empty string and o to 0. Then, we scan $(v-a)\bmod k$ URLs starting at offset o in the main part and using u as a starting value to undo the front coding, and return the last scanned URL.

Figure 5 illustrates how the choice of k affects the space/time tradeoff. The statistics are based on an SHS database containing 2.9 billion distinct URLs. The left graph shows the storage cost as a function of k; the right graph shows the lookup time as a function of k. In both graphs, the horizontal axis denotes the *index stride length* k (the number of URLs per block in the main part of the URL store), using a logarithmic scale. In the left graph, the vertical axis denotes the average storage cost (in bytes) of each front-coded URL in the main index, plus the fractional overhead attributable to the index entry. As can be seen, the cost asymptotically approaches about 18 bytes, and is less than 20 bytes (*i.e.* within 10%

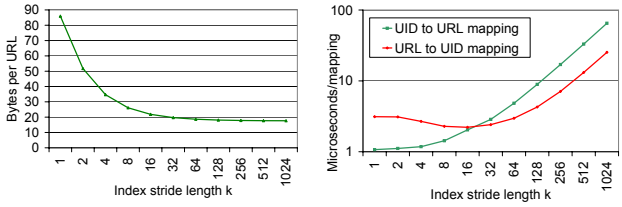

Figure 5: Impact of k on the space/time tradeoff for URL store cells.

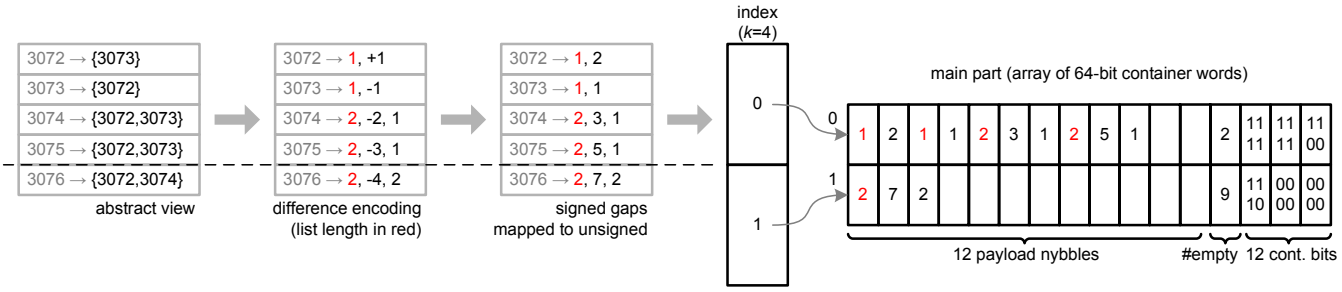

Figure 6: Concrete implementation of forward link store cell from Figure 3, and its derivation from the abstract view.

of the asymptotic limit) for $k \geq 32$. In the right graph, the vertical axis denotes the average time (in microseconds) required to map a URL to a UID or vice versa, again using a logarithmic scale. The green line shows the cost of the UID-to-URL mapping. The line is fairly straight for $k \geq 32$, suggesting a linear relationship between k and UID-to-URL mapping time. This make sense, since the implementation performs a linear scan of the block containing the URL, and on average has to scan half the block to obtain the URL. The red line depicts the cost of the URL-to-UID mapping; it is non-monotonic with a minimum at $k = 16$, and turns straight for fairly large values of k. The reason for the non-monotonicity is as follows: The mapping function first performs a binary search on the index followed by a linear scan of a block in the main part. Small values of k require more binary search probes per lookup, and because the index is less likely to fit into L3 cache, each probe is more likely to have to access main memory, which takes on the order of 100 CPU cycles. On the other hand, the cost of the linear scan is proportional to k; it will be higher (and eventually dominant) for large k.

Link store cells

As we said earlier, storing URLs in lexicographic order and assigning UIDs according to that order means that, within a cell, URLs referring to the same host will be equated with UIDs that are numerically close. This property, together with the fact that there is a great deal of link locality in the web graph (most hyperlinks refer to other pages on the same web server), is key to our approach to link compression. As we saw in Figure 3, each entry in a link store cell maps a UID u_0 to a set of n UIDs u_i ($1 \leq i \leq n$), each representing an edge from u_0 to u_i in the case of a forward link store cell, and u_i to u_0 in the case of a backward link store cell. As we mentioned before, UID u is implicit and thus does not need to be stored. We sort the set such that $u_1 \leq \cdots \leq u_n$ and represent it as a list $[\delta_1, \cdots, \delta_n]$ where $\delta_i = u_i - u_{i-i}$. This technique is known as *difference coding* or *gap coding* [25, page 114ff]. Because of link locality, the δ_i are likely to have low absolute values, and since we sorted the u_i, δ_i is guaranteed to be non-negative for $i > 1$. Because u_1 may be less than u_0, δ_1 may be negative, but its absolute value is still likely to be low. In the two's-complement representation used in standard computers, the bit-pattern of a negative number close to 0 has many leading 1's. To transform δ_1 into a non-negative value while preserving the fact that its absolute value is likely to be small, we move the sign-bit from the leading to the trailing position of the 64-bit word, and take the one-complement of the other 63 bits if δ_1 is negative. This can be done quite efficiently — the C expression

```
x < 0 ? (UINT64)(~x) << 1 | 1 : (UINT64)x << 1
```

maps the signed 64-bit integer `x` (δ_1) to an unsigned value, and the inverse operation is equally simple.

So, each entry in a link store cell can be represented by a list of positive and probably small integers $[n, \delta_1, \cdots, \delta_n]$. We can now choose between many alternative variable-length encoding schemes for compressing these integers, such as Huffman codes, Elias γ and δ codes, Golomb codes, Rice codes, and arithmetic codes [15]. We actually implemented two very simple coding schemes: variable-byte codes and variable-nybble codes.[4] Both schemes allow for very fast encoding and decoding, and given our data sets, their compression ratio is acceptable (although not superb). The computer representations of small positive numbers have long sequences of leading zeroes. In the variable-nybble scheme, the maximum possible number of leading all-zero nybbles is omitted, and only the remainder is stored. The data is stored in containers of 64-bit words, which are divided into 16 nybbles each. The first 12 nybbles hold the "payload" (the integers to be stored without their leading zeroes); the last 3 nybbles hold twelve *continuation bits*, each of which corresponds to a payload nybble and indicates whether that nybble contains the last nybble of a source integer; and the remaining nybble indicates how many of the payload nybbles are unused (this information is not needed for decompression, but useful as a sanity check). Figure 6 shows 2 container words containing the variable-nybble representation of the forward link store cell from Figure 3.

It is worth mentioning that SHS was designed to allow for multiple compression schemes. There are clean interfaces for URL and link encoding and decoding; and the URL and link store cells contain a field indicating the compression scheme used to construct them, allowing for different cells to use different schemes (and in particular, newer cells to use newer coding schemes).

Each link store cell is represented as a contiguous array of memory (consisting of 64-bit container words in the case of variable-byte and variable-nybble encodings). In theory, one could retrieve the UID set of any key UID by performing a linear scan from the beginning of the array, but that would be prohibitively expensive. Instead, we maintain an index data structure, much as we did for URL store cells. A link store cell index is a simple array of integers, one per key UID u, and each entry of the array contains the starting position of u's UID set in the main array. To map a key UID u to its links, one simply gets the value o of index entry $u - a$, and beginning at offset o in the main array decompresses integers $n, \delta_1, \cdots, \delta_n$; finally returning the link UID set $[u + \delta_1, u + \delta_1 + \delta_2, \cdots]$.

Figure 7 illustrates how the choice of k affects the space/time tradeoff. The statistics are based on an SHS database containing 17.7 billion hyperlinks; we used variable-nybble coding to compress the UID gap values. The left graph shows the storage cost as a

[4] A "nybble" is four bits or a half-byte, a pun inspired by the homophony between byte and bite.

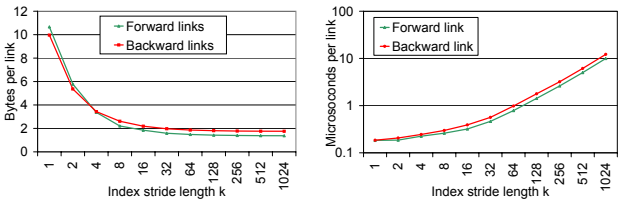

Figure 7: Impact of k on the space/time tradeoff for link store cells.

function of k; the right graph shows the lookup time as a function of k. In both graphs, the horizontal axis denotes the *index stride length* k (the number of UID-lists per block in the main part of the URL store), using a logarithmic scale. In the left graph, the vertical axis denotes the average storage cost (in bytes) of each compressed link in the main index, plus the fractional overhead attributable to the index entry. As can be seen, forward links compress slightly better than backward links, and the cost is less than 2 bytes and within 10% of the asymptotic limit for $k \geq 16$. In the right graph, the vertical axis denotes the average time (in microseconds) required to retrieve one forward link (green line) or backward link (red line) of a UID. It is slightly cheaper to look up forward links than backward links. Both lines are fairly straight for $k \geq 32$, not surprising since the implementation performs a linear scan of the block containing the key UID, and on average has to scan half the block to obtain the list of links.

Distributing the graph

Given a web graph (or subgraph), the SHS system distributes the data across a set of available SHS servers. Each SHS server maintains a cell (consisting of an URL store cell, a forward link store cell, and a backward link store cell), and together these cells contain the entire (sub)graph. For reasons that will become apparent later, we refer to these cells as a *row* or a *generation* of cells.

The graph consists of vertices (URLs) and edges (conceptually, pairs of URLs), and it is partitioned based on the URLs' host-components. We write $host(u)$ to denote the host component of URL u, and we write H_n to denote a hash function (fixed ahead of time) that maps host names to the half-open interval $[0,n)$. Given an SHS cluster consisting of n servers numbered 0 through $n-1$, and a web graph with vertex set V and edge set $E \subseteq V \times V$, the graph is partitioned as follows:

- The URL store cell on server i contains all URLs $u \in V$ where $H_n(host(u)) = i$
- The forward link store cell on server i contains a mapping $u \mapsto \{v : (u,v) \in E\}$ for each $u \in V$ where $H_n(host(u)) = i$
- The backward link store cell on server i contains a mapping $u \mapsto \{v : (v,u) \in E\}$ for each $u \in V$ where $H_n(host(u)) = i$

Partitioning the URL and hyperlink set according to the URLs' host component has several desirable properties. First, the partitioning scheme preserves link locality: all URLs belonging to a web server and all links between pages on the same web server (*i.e.* the majority of all links) are maintained by the same SHS server. Moreover, since the UIDs of a web server's pages form an uninterrupted numeric range, difference coding paired with variable-length coding works well. Second, clerks can determine the SHS server responsible for a URL by performing a simple local computation, without any communication with the service.

Likewise, we would like to be able to determine which SHS server "owns" a UID locally, without any communication. This information is encoded in the UID bit-pattern: Some high-order bits

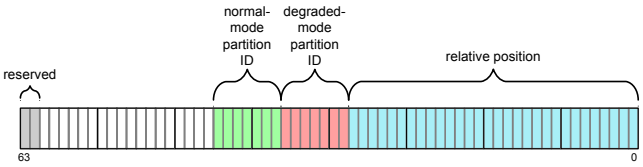

Figure 8: Bit layout of a UID.

in the UID contain a *(normal-mode) partition ID*, while the low-order bits contain the *relative portion* of the UID. The partition ID identifies the SHS server maintaining the corresponding vertex and its connected edges. Figure 8 illustrates the layout. So, given a UID, the clerk can determine which SHS server owns it simply by extracting the partition ID.

Providing Fault Tolerance

Any distributed system is vulnerable to machine failures, and the probability of a machine failing increases with the number of machines. In a system of n computers, if a single machine fails with probability p in a given time window, and assuming independent failures, the probability that at least one of the machines in the system has failed is $1-(1-p)^n$.

The graph partitioning scheme we described above does not have any redundancy, so if any of the SHS servers fails, the portion of the graph stored on that server would become inaccessible to client applications. This is unacceptable in a production setting, and undesirable even in a more-relaxed research setting. We would like to provide some measure of fault tolerance. Specifically, we would like to tolerate up to f machine failures for some low value of f, and we would like to survive complete machine loss (including the loss of all data stored on the failed machines).

The easiest way to provide fault tolerance would be to simply replicate the service. Running $f+1$ identical SHS clusters would allow us to tolerate (at least) f failures without service interruptions, simply by rerouting requests from failed machines to surviving replicas. There are circumstances where this strategy makes sense; for example, the service might be replicated anyway in order to increase the request throughput. However, full replication is costly, and we were interested in a more frugal solution.

The approach we took is predicated on the observation that RAM is far more scarce and expensive than disk. The cost of RAM was the driving factor in developing web graph compression schemes, and it is the reason why our in-memory representation of the web graph does not have any redundancy. Disk, by comparison, is virtually free – it is difficult to buy a hard drive that is *not* at least an order of magnitude larger than the amount of main memory. So, it is quite acceptable to keep several copies of the web graph on disk.

Given an SHS cluster of n machines, we provide tolerance of up to f failures by building *two* SHS stores: a *normal-mode store* with the graph partitioned over n machines, and a *degraded-mode store* containing precisely the same graph, but partitioned across $n-f$ machines (by hashing the URLs' hosts using H_{n-f} instead of H_n). On average, each cell in the degraded-mode store contains $f/(n-f)$ more data that a normal-mode cell (and cells must fit into RAM); if $n = 200$ and $f = 4$, this overhead is just 2%.

Each normal-mode cell and each degraded-mode cell is stored on disk by $f+1$ SHS servers. A cell associated by the host-hash with server i is stored on servers $i, i+1, \cdots, i+f$. (We use modular arithmetic in addressing servers; $n \equiv 0$.)

Figure 9 illustrates the behavior during a failure. The figure shows a system of 8 SHS servers, arranged in a circle, to suggest the modular or "clock" arithmetic used to address servers. There

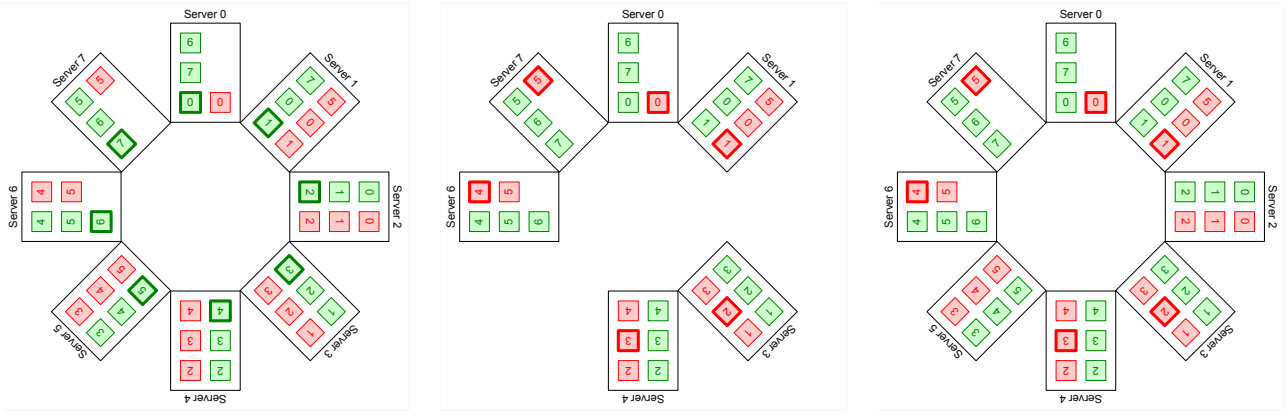

Figure 9: Dealing with failure.

are three copies of each cell, *i.e.* the system tolerates up to two failures. Normal-mode cells are shown in green, degraded-mode cells are red, and the cells currently loaded into RAM are bold. The left panel shows the system in normal mode, with the primary copy of each normal-mode cell loaded into main memory and used to answer requests. In the center panel, servers 2 and 5 have failed; the in-memory normal-mode store on the surviving machines contains only about three-quarters of the web graph. Therefore, $n-f$ of the surviving SHS servers load degraded-mode cells from their local disks, so that the entire web graph is back in main memory, but distributed according to a different host-hash. Clerks are aware of which machines have failed, and direct requests to the SHS server holding the data required to answer the request. Reading the degraded-mode store from disk takes about one minute, and the SHS service is interrupted during that time. Once the system is operating in degraded mode, two blank machines are provisioned from a pool of hot-spares, and the states of the failed machines are copied from surviving machines to the hot-spares (see right panel). Since there were only f failures, there is guaranteed to be at least one surviving replica of each cell on the failed machines. Once the states of the failed machines have been copied to the hot-spares, the system transitions back into normal mode (the left panel), experiencing another minute-long service interruption in the process.

The degraded-mode store is distributed over $n-f$ machines, but there may be more than $n-f$ surviving machines, so in general there are multiple possible mappings of degraded-mode cells to surviving machines. It does not matter which mapping is used, as long as both SHS servers and clerks agree on the same mapping. Given a URL u, clerks can determine which SHS server is responsible for it, by applying hash H_{n-f} to u's host and applying the aforementioned partition-to-survivor mapping to the result.

A degraded-mode cell maintains different URLs (due to the different host-hash function) than its normal-mode counterpart, so the mapping from URL to UID space is different. In particular, the relative portion of a UID is based on the lexical order of the URLs in a given URL store cell, and since degraded-mode cells contain URLs from a different (and on average slightly larger) set of hosts than normal-mode cells, a URL u will have different normal-mode and degraded-mode UIDs. One way to address this problem would be to adopt a transactional semantics. In this model, mapping a URL to its UID and any subsequent SHS queries using this UID must be in the same transaction; and if any SHS server fails during the transaction (thereby causing the system to transition into degraded mode and causing UID spaces to change), clients have to restart the transaction from the beginning.

However, we chose an alternative solution, in which SHS client applications are oblivious to failures. In order to achieve this, we have to address two issues: How to *direct* a UID to the right server, and how to *translate* between normal-mode and degraded-mode UID spaces.

Given a UID v, clerks cannot tell from the normal-mode partition ID of v which degraded-mode cell contains it. Therefore, the layout of each UID reserves some additional high-order bits to contain a *degraded-mode partition ID*, which stores the result of applying H_{n-f} to the host of v's URL. These degraded-mode partition IDs allow the clerk (transparently to client applications) to direct requests related to UID v to the SHS server responsible for v in degraded mode. Note that each URL still has a normal-mode and a degraded-mode UID; these two UIDs have identical normal-mode and degraded-mode partition IDs but (in general) different relative portions.

Client applications always use normal-mode UIDs, and the SHS clerk communicates each normal-mode UID to the responsible SHS server. If the system is in degraded mode, this will be the server addressed by the degraded-mode partition ID and the partition-to-survivor mapping. The server translates the normal-mode UID to a degraded-mode UID, retrieves the desired information from the degraded-mode cell, translates all degraded-mode result UIDs back to normal-mode UIDs, and sends these back to the clerk.

The translation mechanism between normal- and degraded-mode UIDs profits from the fact that we partitioned URLs according to their host component. All the URLs and links associated with a web server are stored in one normal-mode and one degraded-mode cell. So, for all URLs from a given host, the arithmetic difference between their normal-mode and degraded-mode UIDs is constant.

Each SHS server maintains two translation tables: A normal-to-degraded table that maps the lowest normal-mode UID of each host to its degraded-mode counterpart, and a degraded-to-normal table that maps the lowest degraded-mode UID of each host to its normal-mode counterpart. Each table is implemented as an array of UID pairs sorted by key UID; lookup amounts to binary search on the key UID column. Both tables only contain entries for those hosts that are covered by the degraded-mode cell of that SHS server. All normal-to-degraded UID mappings can be resolved locally, since an SHS server will only receive requests related to hosts it covers in degraded mode. Result UIDs, however, may belong to any host. Because of link-locality, most result UIDs can be translated using the local degraded-to-normal table. Translating the remaining results requires a round of RPC calls to peer servers; caching of the degraded-to-normal UID pair of "popular" hosts re-

duces the number of translations that cannot be performed locally.

Building a store row

The typical way to compile a web graph is to crawl the web and to write the URLs of crawled pages and their links to disk. Constructing an SHS store from this data is a multi-stage process:

1. One or more ShsBuilder processes read through the data produced by the crawler, and send each page and link URL u to the SHS servers identified by $H_n(host(u))$ and $H_{n-f}(host(u))$. Each SHS server receives a stream of URLs from the SHS builders, and appends each URL to a normal-mode and/or a degraded-mode file (performing the same two hashes to decide what file to append to).
2. Once all the crawled URLs have been transmitted, the SHS servers sort the normal-mode URLs lexicographically. Sorting is done in two phases: In phase 1, a large number of URLs are read from the normal-mode file into memory, sorted using Quicksort, and written to a separate temporary file; this process is repeated until the normal-mode file is exhausted; in phase 2, the temporary files are read in a merge-sort pattern, duplicate URLs are eliminated, distinct URLs are front-coded and written to the main part of an URL store cell, and the index is written *en passant*. Following that, the degraded-mode URLs are sorted in the same way.
3. The ShsBuilder processes make a second pass through the data produced by the crawler, and send each pair (u,v) of page URLs u and link URLs v to the servers identified by $H_n(host(u))$, $H_n(host(v))$, $H_{n-f}(host(u))$, and $H_{n-f}(host(v))$. By performing the same hashes, an SHS server can determine whether a received pair (u,v) belongs to a normal-mode or/and degraded mode cell on this machine, as well as to the forward- or/and backward-link store cell on this machine. Because of link-locality, most pairs belong to both the forward- and the backward-link store cell on this machine. If this is the case, both u and v can be mapped to their UIDs locally, by looking them up in the URL store cell constructed in step 2; otherwise, the lookup is performed by a peer SHS server, and many URLs are batched up to amortize the RPC overhead. UID pairs are written to one or more (depending on host-hashes) of four temporary disk files – a normal- and a degraded mode forward-file containing $uid(u), uid(v)$ pairs, and a normal- and a degraded mode backward-file containing $uid(v), uid(u)$ pairs.
4. Once all the crawled page/link pairs have been transmitted, each SHS server sorts the four temporary files according to key UID, eliminates duplicate links in the process, and builds four link store cells based on these files. All temporary files are discarded once they are no longer needed.
5. Finally, each SHS i server copies all the newly constructed cells to its peer servers $i+1,\cdots,i+f$, thus ensuring that there are $f+1$ copies of each file.

4. MANAGING CHANGE AND GROWTH

The SHS implementation described in the previous section was designed for immutable web graphs. The SHS store is built from a set of crawled web pages (as described in section 3) and subsequently becomes available to SHS clients, but once the store is built, it can no longer be changed. Performing incremental updates on an SHS store is hard due to its basic design: The URLs in an URL store cell are stored in lexicographic order, and UIDs are assigned according to that order. Inserting a new URL u into an URL store cell U would increment the UIDs of all those URLs in U that are larger than u, which in turn would require every occurrence of these UIDs in all link store cells to be adjusted. In other words, inserting even a single new URL into the store would require all link store cells to be rewritten.

In this section, we will describe an extension of the basic SHS design that allows for incremental updates to the store. However, this additional functionality comes at a substantial price: The URL and link compression ratio decreases, lookups of both URLs and links become more expensive, and the system is decidedly more complex. So, in domains where incremental updates are not required, the basic SHS design described in section 3 is preferable.

The key idea of our incremental update scheme is to put updates into a new store row (a set of cells partitioned across all SHS servers) rather than integrating them into the existing store, thereby avoiding any changes in the existing URL-to-UID mapping and the resulting need to rewrite existing link store cells. So, instead of a single row of cells, an SHS store is now a grid of cells, where the columns correspond to SHS servers and the rows correspond to "generations" of the vertices and edges. In order to avoid a rapid increase in the number of rows, updates are performed in appropriately-sized batches; each batch produces one new row. Moreover, several rows are periodically merged into a single row, further controlling the number of rows. We leverage this merge process to also allow us to change the number of machines in a running SHS cluster, *e.g.* to cope with an increase in the graph size.

Incorporating new and changed web pages

As in the static case, building a store row g is a multi-step process involving an ShsBuilder (which now processes a batch of updates) and n ShsServer processes. Step 1 of the process remains unchanged: the ShsBuilder sends each page and link URL to the server identified by the host-hash. In step 2, each server i sorts the received URLs as before; however, the merge phase of the sort is modified such that only new URLs (those that are not already contained in an existing URL cell on that server) are incorporated into the new URL cell $U_{i,g}$. This duplicate elimination requires a single, streaming pass through the existing URL cells on that server. Step 3 of the build process remains unchanged (except that the URL-to-UID mapping involves more cells, as described below).

Explaining the changes to step 4 of the build process requires some background. A batch of updates consists of a set of crawled pages and their embedded hyperlinks. Some of these pages may be new, other may have been crawled previously and their embedded links may have changed. Assume that the page u previously contained links $v_1,\cdots,v_p$ and now contains links $v'_1,\cdots,v'_q$. We reflect this by adding a mapping $u \mapsto v'_1,\cdots,v'_q$ to the forward link store. Updating the backward link store is not as straightforward; there are two alternative solutions.

The first solution requires no modification to the semantics of a backward link store entry, but is expensive both in terms of space and update time cost. We look up the old forward links $\mathbf{W}$ of every $v \in \{v'_1,\cdots,v'_q\}\backslash\{v_1,\cdots,v_p\}$ and add a mapping $v \mapsto \mathbf{W} \cup \{u\}$, and similarly look up the old forward $\mathbf{W}$ links of every $v \in \{v_1,\cdots,v_p\}\backslash\{v'_1,\cdots,v'_q\}$ and add a mapping $v \mapsto \mathbf{W} \backslash \{u\}$. If the average out-degree is δ and the fraction of changed links on a page is λ, this solution requires $2\lambda\delta+1$ lookups in the old forward link store, and adds an average of $2\lambda\delta^2$ link UIDs per changed page to the new backward link store cell.

A more efficient solution is to add *deleted-tags* to link UIDs. We write $\overline{u}$ to tag UID u as deleted. We add a mapping $v \mapsto \{u\}$ to the new backward link store cell for each $v \in \{v'_1,\cdots,v'_q\}\backslash\{v_1,\cdots,v_p\}$, and a mapping $v \mapsto \{\overline{u}\}$ for each $v \in \{v_1,\cdots,v_p\}\backslash\{v'_1,\cdots,v'_q\}$.

This solution requires just one lookup in the old forward link store (to obtain $v_1, \cdots, v_p$), and adds an average of $2\lambda\delta$ link UIDs per changed page to the new backward link store cell. However, looking up a UID u in the backward link store becomes somewhat more complicated, as we need to look up u in multiple backward link store cells and combine the UID lists to obtain the final result list.

The link store cell implementation described in section 3 exploited the fact that each URL in an URL store had exactly one corresponding entry in the forward link store cell and one entry in the backward link store cell. Now, however, some of the entries in the backward and forward link store cells will have "new" key UIDs (UIDs defined by the corresponding URL store cell), others will be "old" (corresponding to URLs from previous generations). In order to cope with this, each link store cell is divided into an "old key" and a "new key" portion. The implementation of the "new key" portion is unchanged from section 3. Keys in the "old key" portion are non-continuous, so they are stored explicitly in both the index and the main part.

Deleted-tags can only occur in link UIDs in the "old key" portion of a backward link store cell. They can be encoded efficiently by multiplying each link UID in this portion by 2, and using the now vacant lowest-order bit of the UID to represent the tag. Using the lowest-order bit means that two UIDs on the same server (deleted or not) continue to be numerically close and thus compress well.

A UID u has an associated generation g of the URL store cell that maps u to its corresponding URL. Looking up a UID u in the forward link store entails looking it up in the newest-generation cell on the server responsible for u. If u is not in that cell, the search progresses to the previous-generation cell, until u is found or generation g has been passed. Looking up u in the backward link store (and assuming the "deleted-tag" solution) entails looking u up in all backward link store cells starting at generation g and up to the current generation, and combining the partial results obtained from each cell into a total result.

Merging cell rows

Incorporating a batch of updates into the SHS server creates a new row of cells, and the cost of lookups is proportional to the number of rows. Therefore, we control the store growth by periodically merging the m newest rows. Merging rows (and in particular merging URL store cells) changes the mapping between URLs and UIDs; merging the *newest* rows has the desirable property that the only rows affected by this remapping are the rows that are being merged.

Merging m rows is a multi-step process: First, each SHS server merges its m newest URL store cells, which involves a linear scan through all these cells. The new URL store cell is written to disk as it is being assembled. In addition, the server constructs a translation table T for mapping old UIDs to new UIDs, as the merge sort is being performed. The new UIDs occupy the same numeric range as the old UIDs; specifically, the lowest new UID is identical to the lowest old UID. The most straightforward implementation of T consists of arrays of 64-bit integers, one array per old URL store cell and one entry per UID, each entry indexed by the old UID and containing the new UID. A more sophisticated implementation reduces the memory requirements by nybble-and gap-encoding the new UIDs (which conveniently are in sorted order in each subtable, with small deltas between adjacent entries), and uses an index data structure similar to the indices described in section 3 to avoid the need for a linear scan through the main array to undo the decompression.

Second, each server merges its m newest forward link store cells. Merging link store cells consists of interleaving entries from the old cells in the correct order (such that the new key UIDs are in sorted order), and remapping all UIDs that belong to the generations being merged (older UIDs are unaffected). Because of link locality, most of the remappings can be performed using this server's translation table; some fraction requires lookups in the tables of peer servers; these lookups can be batched up (to amortize the RPC cost), and caching popular mappings helps as well. Third, each server merges its m newest backward link store cells in much the same way. This concludes the merger of the normal-mode cells; each server now repeats the same process for the degraded-mode cells. Having done so, the new normal- and degraded-mode cells are copied to the server's f neighbors (to provide redundancy).

Once the merge is completed, the SHS server atomically switches over to the new store. This switch-over involves unloading the m cell rows that are now defunct, and loading the merged row that replaces them. This causes a brief service interruption similar to the transition between normal and degraded mode. However, unlike the transition between normal and degraded mode, this transition cannot be made transparent to the client. The UID space has changed, and the SHS servers make no attempt to translate old UIDs to new UIDs. Instead, the SHS client API is modified: methods such as UrlToUid or MinUid whose results are relative to a particular UID space now return an *epoch number* in addition to their result, and methods such as UidToUrl or GetLinks whose inputs are relative to a particular UID space now take an epoch number as an additional input. If the epoch number passed to such a method no longer matches the epoch of the SHS servers (because a merge has been committed), the method call throws an exception. Clients must catch these exceptions and restart at the call that produced the epoch number. In effect, this introduces transactional semantics to SHS.

The epoch number is increased whenever the SHS store changes visibly to the client, *e.g.* by incorporating a batch of updates possibly followed by a merge. Merge operations are triggered by updates to the store. If the binary representation of the new epoch number has m trailing zeros, then $m+1$ generations are merged as part of the update. So, half of the updates incur no merge overhead (since merging one row is a no-op), a quarter merge the two newest rows, an eighth merge the three newest rows, and so on. Since the cost of merging rows is proportional to the combined size of the old rows, merging rows in the way we described means that the amortized merge cost is logarithmic to the size of the store. Furthermore, the merge scheme controls the number of rows: at epoch e, there will be $\log_2 e$ store rows.

Without going into too much detail, the merge process is also used to garbage-collect URLs. During the URL store cell merge phase, any URLs that do not have any edges associated with them are not incorporated into the new URL store cell. Determining whether a URL has associated edges requires lookups in the link store cells that are in the scope of the merge; these lookups are not random, but rather stream through the link store cells in the same way the URL merging streams through the URL store cells; therefore this process of weeding out "dead" URLs is very efficient.

Changing the number of SHS servers

The amount of useful content on the web continues to increase,[5] and search engines respond to this by increasing their index sizes. In order to grow the web graph maintained by SHS in lock-step, we need to provision more servers. We can change the number of servers in an operating SHS cluster by leveraging the row merging mechanisms. The basic idea is that we do not attempt to repartition the entire store at once. Instead, batches of updates arriving after

[5]The amount of *useless* content has been infinite for a while now.

the cluster has been grown are hashed over all available machines. Moreover, during each merge operation the new row that replaces the m old rows is hashed over all available machines. This way, the web graph will be gradually repartitioned to utilize the extended cluster of SHS servers.

The details of this scheme for growing the cluster add yet more complexity to the design. Growing the cluster from n to $n+d$ machines means that now there are two normal-mode host-hash functions, H_n and H_{n+d}, and if the cluster was grown several times, there might be more. In order to perform a UrlToUid(u) call, SHS clerks need to contact all servers that could contain URL u based on the host-hash functions. Moreover, row merge operations that cross an expansion boundary are more complicated. URLs and their associated links have to be repartitioned to peer servers (which requires streaming network communication), and the UID translation tables have to be shuffled between the servers as well. A detailed description of the process is beyond the score of this paper; suffice it to say that the overhead of merging rows that cross cluster expansion boundaries is not prohibitive.

5. CONCLUSION AND FUTURE WORK

This paper described the Scalable Hyperlink Store, a scalable system that provides extremely fast access to the web graph. SHS partitions the nodes and edges that make up the web graph over a set of servers, keeps all the data in main memory to provide extremely fast access, and uses compression techniques that leverage properties of the web graph to reduce the memory footprint. We believe that the design presented in this paper will be a useful blueprint for future storage systems for the web graph.

The basic SHS architecture was developed in early 2003 [16]; the core system has been operational and in fairly constant use since 2005. We have used it to study the effectiveness of various query-dependent link-based ranking algorithms along the lines of HITS [10, 17, 18, 19, 21]. Other groups within Microsoft have used SHS as well, sometimes in unexpected ways. For example, SHS has been used to maintain a "click graph" – a graph that associates search queries with the results that users click on. Performing random walks on this graph surfaces keywords related to the query, which is useful for query suggestion, ad targeting, and keyword sales [9]. SHS has also been used to study structural properties of the MSN messenger graph.

There are numerous avenues for future work. We are currently experimenting with alternative URL and link compression schemes and measuring the resulting space/time tradeoffs. We are continuing to investigate query-dependent link-based ranking algorithms; in the past, such work provided motivation to extend the SHS API; for example, we incorporated random and consistent sampling of links into the SHS server after it became apparent that this operation was used by many HITS-like ranking algorithms, and that pushing it from the client into the server would result in substantial performance gains. We are looking for improvements to the incremental update scheme that preserve more of the performance advantages of the basic system for maintaining static graphs. Finally, we are considering whether there are efficient ways for attaching extra information to nodes and edges (examples include anchor texts and weights), a feature that has been requested by several users. Unfortunately, our link compression scheme in particular is highly optimized, and attaching arbitrary labels (with a distribution that is not known *a priori*) would have a severe impact on the compression ratio. One solution would be to allow clients to supply label-specific coders and decoders.

6. REFERENCES

[1] M. Adler and M. Mitzenmacher. Towards Compressing Web Graphs. In *11th IEEE Data Compression Conference*, March 2001, pages 203–212.

[2] L. Becchetti, C. Castillo, D. Donato, R. Baeza-Yates, and S. Leonardi. Link Analysis for Web Spam Detection. *ACM Transactions on the Web*, 2(1), 2008.

[3] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The Connectivity Server: fast access to linkage information on the Web. In *7th International World Wide Web Conference*, April 1998, pages 469–477.

[4] P. Boldi and S. Vigna. The WebGraph Framework I: Compression Techniques. In *13th International World Wide Web Conference*, May 2004, pages 595–601.

[5] P. Boldi and S. Vigna. The WebGraph Framework II: Codes For The World-Wide Web. In *14th IEEE Data Compression Conference*, March 2004, page 528.

[6] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-Wise Independent Permutations. *Journal of Computer and System Sciences* 60(3):630–659, 2000.

[7] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web. In *9th International World Wide Web Conference*, May 2000, pages 309–320.

[8] G. Buehrer and K. Chellapilla. A Scalable Pattern Mining Approach to Web Graph Compression with Communities. In *1st Intl. Conf. on Web Search and Data Mining*, February 2008, pages 95–106.

[9] A. Fuxman, P. Tsaparas, K. Achan, and R. Agrawal. Using the Wisdom of the Crowds for Keyword Generation. In *17th International World Wide Web Conference*, April 2008, pages 61–70.

[10] S. Gollapudi, M. Najork, and R. Panigrahy. Using Bloom Filters to Speed Up HITS-like Ranking Algorithms. In *5th Workshop on Algorithms and Models for the Web-Graph*, December 2007, pages 195–201.

[11] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1998, pages 668–677.

[12] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the Web for Emerging Cyber-Communities. In *8th International World Wide Web Conference*, May 1999, pages 11–16.

[13] R. Lempel and S. Moran. The stochastic approach for link-structure analysis (SALSA) and the TKC effect. *Computer Networks and ISDN Systems*, 33(1–6):387–401, 2000.

[14] M. Marchiori. The quest for correct information on the Web: Hyper search engines. In *Computer Networks and ISDN Systems*, 29(8–13):1225–1236, 1997.

[15] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

[16] M. Najork. *System and method for maintaining a distributed database of hyperlinks*. US Patent 7340467; filed April 2003, issued March 2008.

[17] M. Najork. Comparing the Effectiveness of HITS and SALSA. In *16th ACM Conference on Information and Knowledge Management*, November 2007, pages 157–164.

[18] M. Najork and N. Craswell. Efficient and Effective Link Analysis with Precomputed SALSA Maps. In *17th ACM Conference on Information and Knowledge Management*, October 2008, pages 53–61.

[19] M. Najork, S. Gollapudi, and R. Panigrahy. Less is More: Sampling the Neighborhood Graph Makes SALSA Better and Faster. In *2nd ACM International Conference on Web Search and Data Mining*, February 2009, pages 242–251.

[20] M. Najork and A. Heydon. High-Performance Web Crawling. In *Handbook of Massive Data Sets*, Kluwer Academic Publishers, 2002.

[21] M. Najork, H. Zaragoza, and M. Taylor. HITS on the Web: How does it Compare? In *30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, July 2007, pages 471–478.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[23] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The Link Database: Fast Access to Graphs of the Web. In *12th IEEE Data Compression Conference*, April 2002, pages 122–131.

[24] T. Suel and J. Yuan. Compressing the Graph Structure of the Web. In *11th IEEE Data Compression Conference*, March 2001, pages 213–222.

[25] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes* (2nd edition). Academic Press, 1999.