November 16, 2001

SRC Research Report

175

The Link Database: Fast Access to Graphs of the Web

Keith H. Randall, Raymie Stata,
Rajiv Wickremesinghe, Janet L. Wiener

Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

http://www.research.compaq.com/SRC/

Abstract

The Connectivity Server is a special-purpose database whose schema models the Web as a graph graph where URLs are nodes and hyperlinks are directed edges. The Link Database provides fast access to the hyperlinks. To support a wide range of graph algorithms, we find it important to fit the Link Database into memory. In the first version of the Link Database, we achieved this fit by using machines with lots of memory (8GB), and storing each hyperlink in 32 bits. However, this approach was limited to roughly 100 million Web pages. This paper presents techniques to compress the links to accommodate larger graphs. Our techniques combine well-known compression methods with methods that depend on the properties of the web graph. The first compression technique takes advantage of the fact that most hyperlinks on most Web pages point to other pages on the same host as the page itself. The second technique takes advantage of the fact that many pages on the same host share hyperlinks, that is, they tend to point to a common set of pages. Together, these techniques reduce space requirements to under 6 bits per link. While (de)compression adds latency to the hyperlink access time, we can still compute the strongly connected components of a 6 billion-edge graph in under 20 minutes and run applications such as Kleinberg's HITS in real time. This paper describes our techniques for compressing the Link Database, and provides performance numbers for compression ratios and decompression speed.

1 Introduction

The Connectivity Server is a special-purpose database whose schema models the Web as a graph where URLs are nodes and hyperlinks are directed edges. The Link Database is the portion of the Connectivity Server that provides fast access to the hyperlink data. It allows us to run arbitrary graph algorithms over the Web graph in real time, usually by writing textbook implementations of the algorithms. Applications of the Link Database include PageRank [BP98], which ranks the importance of web pages based on their inlinks; Kleinberg's HITS [K98], which refines web query results by examining local subgraphs of the web; mirror detection, which uses links and URLs to find web sites that mirror each other's contents [BBDH99]; and studies of web structure [BKM00]. While the Link Database is critical for each of these applications, we know of no other database that can store as many links in as little space.

We have developed three versions of the Link Database over the last four years. The first version, Link1 [BBH98], encodes adjacency lists in 32 bits per link, or approximately 68 bytes per URL. (In our data, URLs have an average of 17 links.) With 8 GB of memory, Link1 allows experiments on collections of 100 million Web pages, which contained much of the Web in 1998. To scale with the growth of the Web since 1998, we considered switching the Link Database to a disk-based approach. Although there is active research in disk-based graph algorithms [CGG95, KS96, ABW98], memory-based graph algorithms tend to be simpler, easier to understand, and easier to implement. Therefore, we developed compression techniques to keep the Link Database in memory.

Many compression techniques currently exist. These include both symbolwise methods, such as Huffman codes, that replace symbols with (hopefully) shorter symbols, and dictionary methods, such as Lempel-Ziv, that replace sets of symbols with a single new symbol [WMB99]. We use both types of methods in the second and third versions of the Link Database, Link2 and Link3, as well as other special-purpose methods that depend on the properties of hyperlinks. Link3 reduces the original 32 bits per link to less than 6, while still providing fast decompression speeds. For example, we can compute the strongly connected components of the web graph, using an algorithm that accesses 6 billion hyperlinks in random order, in less than 20 minutes.

This paper describes our Link Database implementations. Sections 2, 3, and 4 describe Link1, Link2, and Link3, respectively. Section 5 presents measurements that show both the compression achieved by the various techniques and also their impact on performance. Section 6 describes related work and we conclude in Section 7.

2 Background and Link1

The Connectivity Server is a collection of three databases: the URL Database, the Host Database, and the Link Database. The URL Database contains a set of URLs and mappings from URLs to fingerprints (a hash of the URL) and to URL-ids (sequentially ordered integers). The Host Database contains a partitioning of the URL-ids in the URL Database into groups called "hosts" based on the hostname portion of the URLs. (The Host Database is not considered further in this paper.) The Link Database contains two adjacency lists for all URL-ids in the URL Database. For each URL-id, the *outlinks* adjacency list contains the list of URL-ids of other URLs which appear as hyperlinks on its web page, and the *inlinks* adjacency list contains the list of URL-ids of all URLs that have hyperlinks to it. This section provides background on the URL Database and the first version of the Link Database and also describes the input used to build them.

2.1 Links files

The Connectivity Server is built from a collection of *links files*. Each links file is a sequence of records; each record consists of a *source* URL followed by a sequence of *destination* URLs. A fragment from a typical links file might look like:

```
http://www.foo.com/
   http://www.foo.com/css/foostyle.css
   http://www.foo.com/images/logo.gif
   http://www.foo.com/images/navigation.gif
   http://www.foo.com/about/
   http://www.foo.com/products/
   http://www.foo.com/jobs/

http://www.foo.com/about/
   http://www.foo.com/css/foostyle.css
   http://www.foo.com/images/logo.gif
   http://www.foo.com/directions.html
   http://www.foo.com/about/
   http://www.foo.com/products/
   http://www.foo.com/jobs/
```

The order of destination URLs should match the order in which they appear on the HTML page. Although the Connectivity Server is independent of the source of the links files, we typically generate them in our crawler, Mercator [HN99], or by extracting them from HTML pages stored on disk.

If a page is crawled multiple times, it may appear as the source URL for multiple records. Therefore, the software that builds the Connectivity Server reads the links file in reverse chronological order, and only processes each source URL the first time it is encountered, which should be the most recent record for that URL.

2.2 The URL Database

The URL Database contains all source URLs that appear in the links files, plus any URLs that appear more than some threshold *T* number of times as destination URLs. By ignoring URLs that do not appear either as sources or in more than *T* pages, we avoid cluttering the Connectivity Server with non-existent URLs that are simply misspellings on the part of authors. The parameter *T* is set when the URL database is created; a typical value is 4.

The URL Database represents each URL in three ways: as text; as a 64-bit hash of the URL text which we call a *fingerprint*; and as a 32 bit integer *URL-id*. The URL Database provides six functions to map between all pairs of representations; that is, to map from text to fingerprints and back again, from fingerprints to URL-ids and back again, and from URL-ids to text and back again.

Before being hashed to fingerprints, the URL text is normalized, by converting hostnames to lower case, cannonicalizes port numbers, re-introducing them where needed, and adding a trailing slash to all URLs that do not have them. Normalizing increases the likelihood that two

URLs for the same page will have the same fingerprint. Functions that take URL text always normalize it. However, when returning text, the URL database always returns the *stored version* of the URL text, which is a version of the text as it appears in the links files. If the URL appears in the source field of the links files, then that version of the text will be stored; otherwise, one of its destination URLs is picked arbitrarily.

The URL-ids are sequentially assigned integers from 1 to *N*. This dense assignment is useful both in the Connectivity Server implementation and when writing applications, because URL-ids can serve as indexes into arrays. URL-ids are assigned to URLs as follows. First, we divide the URLs into three partitions based on their degree. The *indegree* of a URL is the number of inlinks it has; the *outdegree* is the number of outlinks it has. If the indegree or outdegree is higher than 254, we put the URL in the high-degree partition; if it is between 24 and 254, we put it in the medium-degree partition; otherwise we put it in the low-degree partition. (We chose 254 so that bookkeeping of the indegree could be stored in 1 byte; we chose 24 because 23 is the maximum number of links whose compressed form can be stored in 255 nybbles (see Section 3.3.) Within each partition, we sort URLs lexicographically. We then assign URL-ids first by partition and then by lexicographic order. Thus, the high-degree URL which sorts first lexicographically is assigned URL-id 1. If there are *H* high-degree URLs, then the medium-degree URL which sorts first lexicographically is assigned URL-id *H+1*. As we will show in Section 3.3 and Section 5, the URL partitioning has important consequences for our compression.

On disk, the URL database consists of the set of URLs in compressed form and three indices, one from fingerprints to URL-ids, another from URL-ids to fingerprints, and the last from URL-ids to URLs. On average, storing each URL requires 9 bytes for the compressed URL and 5, 8, and 0.5 bytes for the three indices, respectively.

2.3 The Link Database

The Link Database maps from each URL-id to the sets of URL-ids that are its outlinks and its inlinks. In the terminology of graph theory, these sets are the *adjacency lists* of two directed graphs. The outlinks are the adjacency lists of the Web graph as we usually think about it: the links are directed in the way that Web surfers follow them. The inlinks are the adjacency lists of the transpose of that graph: the links are directed in the opposite direction of the outlinks.

The Link Database provides two functions that take a URL-id as input and returns the URL-ids in an adjacency list. Inlink adjacency lists are returned in ascending order of URL-ids. Outlink adjacency lists can be stored and returned in the original order of the corresponding destination URLs. However, we typically use ascending URL-id order because it compresses better.

The Link Database is not designed to allow modification of the graph. Crawling the web to produce the Connectivity Server input files takes weeks or months whereas building the Connectivity Server database takes hours. Therefore, we assume that we may always create the Link Database in batch mode.

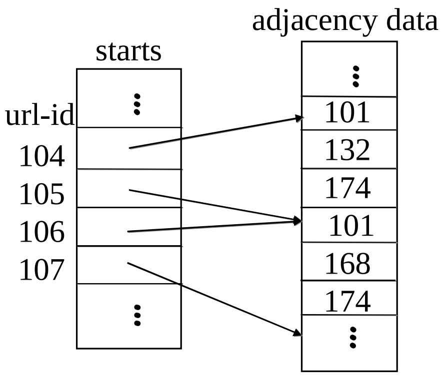

Figure 1: Simple encoding of adjacency lists

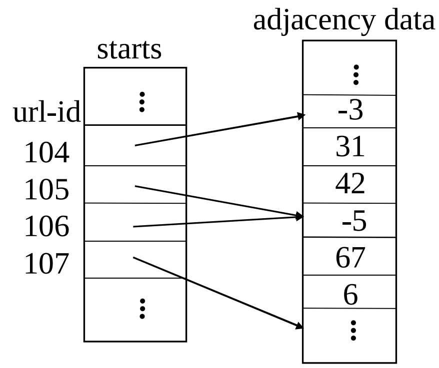

Figure 2: Delta encoding of adjacency lists

The original version of the Link Database [BBH98], Link1, stores the outlink and inlink adjacency lists as two arrays each (four arrays total). The first array, called *starts*, is indexed by URL-id and contains an index into the second array. The second array, called *adjacency data*, contains the URL-ids in the adjacency lists. Each array element is 32 bits long. The length of a URL's adjacency list is determined by subtracting the URL's entry in the starts array from the subsequent entry. Figure 1 shows a few hypothetical adjacency lists. In the figure, URL-id 104 has an adjacency list of length 3, containing the URL-ids 101, 132, and 174. URL-id 105 has an empty adjacency list.

3 Link2: single list compression and starts compression

Our first compression technique is single list compression, which compresses the representation of each adjacency list individually. We take advantage of two facts to achieve good compression. First, the vast majority of links on a page are *local* in that they point to other URLs on the same host. We have found empirically that 80% of all links are local. Second, the assignment of URL-ids to URLs respects, in large part, the lexicographic order of the URLs themselves. This URL-id ordering means that URL-ids on the same host tend to be close to one another in the URL-id space, and URLs that share a longer common prefix have even closer URL-ids. Combined, these facts mean that URL-ids in the same adjacency list tend to be close together in the URL-id space.

3.1 Delta values

Single list compression takes advantage of these facts. First, rather than storing absolute URL-ids in each adjacency list, we store the differences between neighbors of the list, which are called *delta* (or *gap*) values [WMB99]).

Figure 2 shows how the absolute values in Figure 1 are replaced by delta values. The first value in the adjacency list for URL-id 104, 101, is replaced by 101-104=-3. The second value, 132, is replaced by 132-101=31 and the third value, 174, is replaced by 174-32=42. In general, the first absolute value $a_{i,1}$ in each adjacency list for URL-id i is replaced by delta value $a_{i,1}$-i. Each subsequent absolute value $a_{i,j}$ is replaced by delta value $a_{i,j}$ - $a_{i,(j-1)}$.

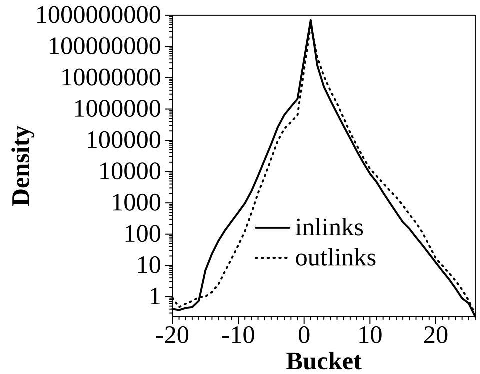

Figure 3: Distribution of delta values

Delta values tend to be small, much smaller than the corresponding absolute values. This point is illustrated in Figure 3, which contains a histogram of delta values. In this figure, delta values are put into buckets of exponentially increasing size: bucket 1 contains the delta value 1; bucket 2 contains the values 2 and 3; bucket 4 contains 4, 5, 6, and 7; and bucket *i* contains the delta values $2^{(i-1)}$ to 2^i-1. (We eliminate reflexive and duplicate links from our data set, so the delta value zero never appears.) The x-axis of Figure 2 gives the bucket number, and the y axis shows the density of each bucket, that is, the number of delta values in each bucket divided by the range of the bucket. In this data set, both inlinks and outlinks are stored in ascending order, so only the first delta of an adjacency list can be negative. Therefore, there are many fewer negative deltas than positive ones. (When outlinks are stored in original order, any delta can be negative, but a similar histogram applies and we get similar, although not quite as good, compression.)

3.2 Variable-length nybble codes

As illustrated in Figure 2, the distribution of delta values is highly concentrated and biased toward small numbers. We then store the delta values using a variable-length format that uses fewer bits for smaller values. Many symbolwise codes provide variable-length encodings, e.g., Huffman, Golomb, and arithmetic codes [WMB99]. However, these codes are optimized for large quantities of data. Our adjacency lists average only 68 bytes. Therefore, we experimented with the following two encodings.

The first encoding is a Huffman code (with a table of 20,000 delta values and an escape for values beyond that). The second encoding is a *nybble code*, which consists of strings of nybbles (4-bit quantities). The low-order bit of each nybble, called the *stop bit*, indicates whether or not there are more nybbles in the string; the remaining bits provide an unsigned number. Multi-nybble strings are combined in little-endian fashion. Where a negative number is possible, we use a sign/magnitude encoding in which the least-significant bit encodes the sign. For example, the nybble encoding of the delta value 28 is *0111 1000*. If the same value could have been negative, then the lowest data bit would be an additional 0, resulting in the string *1111 0000*. The delta value -6 is represented by the string *0011 1010*. As the performance numbers in Section 5 will show, the nybble code does not achieve quite as good compression as Huffman coding, but is much faster at decompression.

3.3 Starts array compression

In Link2, we also developed a mechanism for compressing the starts array. For each of the three URL partitions mentioned in Section 3.1, we use a different number of bits to encode the starts entries. For the high degree URL partition, the starts array consists of 32-bit indices as before. For the medium-degree partition, the starts array is divided into records consisting of a 32-bit base index, plus P 16-bit offsets from the previous index. The average size of an index is thus (32 + 16*P)/P bits. The starts array for the small-degree partition is similar to the medium-degree partition except that the offsets are stored in 8 bits instead of 16. Since adjacency lists start and end on nybble boundaries, in Link2, elements of *starts* are indices to nybbles (rather than indices to 32-bit integers as in Link1). As described in [BKM00], indegrees and outdegrees in the Web graph are distributed exponentially. Thus, most of the URLs (74% in our dataset) are contained in the small-degree partition. In other words, the majority of entries in the *starts* array are in the small-degree partition where indices take a little over 8 bits each. As a result, with P=16, the average size of an entry in the compressed *starts* array is 12 bits, rather than 32 bits as in Link1.

4 Link3: interlist compression

Our second compression technique, interlist compression, describes each adjacency list as a modification to a *representative* list. As in dictionary codes [WMB99], sets of symbols (adjacency lists) may refer to other sets. Unlike in dictionary coding, the other set need not ever appear in the data, and the two sets need not be exactly the same: we also encode the differences between the two lists. Interlist compression saves space because many redundancies exist across adjacency lists. Hyperlinks in the Web graph are not uniformly distributed. Some URLs are very popular, meaning they appear in many outlink adjacency lists [BKM00], while others are not. More generally, URLs on the same host tend to have many outlinks and inlinks in common. This section presents data showing that the redundancies exist and then describes our techniques for taking advantage of them.

4.1 Adjacency list redundancy and ordering

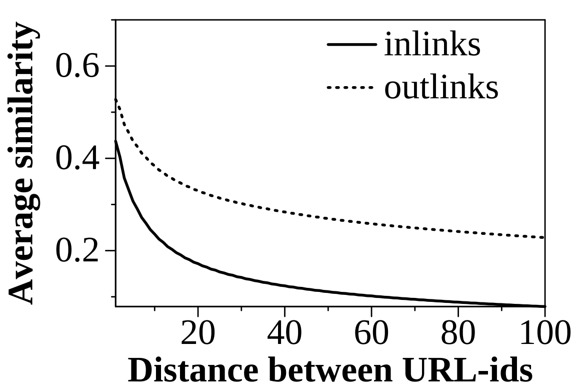

Figure 4: Similarity between neighboring adjacency lists

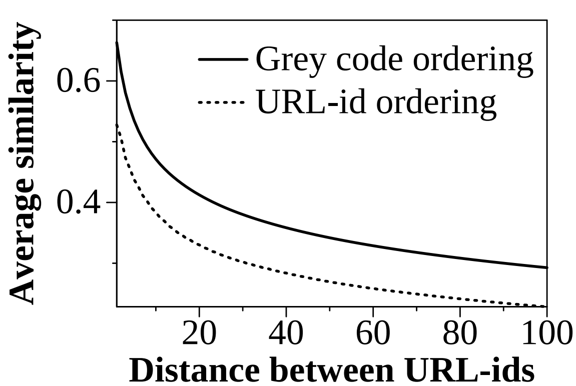

Figure 5: Similarity of outlinks given different node orderings

Figure 4, which contains a plot of the similarity between pairs of adjacency lists, shows that there is significant redundancy across adjacency lists and that this redundancy is easy to find. The x-axis of Figure 4 is the absolute value of the difference between pairs of URL-ids (e.g., the URL-id-pair consisting of 106 and 116 appears at $x=10$). The y-axis is the average similarity over all URL-id pairs at a given difference. We define similarity as the size of the intersection of the two lists divided by the size of their union; similarity 1 means the lists are identical; 0.5 means they have two-thirds of their elements in common, and 0.33 means that they have half of their elements in common.

As Figure 4 demonstrates, the adjacency lists of URL-ids with a distance of 1 have significant overlap, and the overlap decreases with increasing URL-id distance. The similarity of inlinks is both lower and decreases faster than the similarity of outlinks. We believe that there is less similarity among inlinks because there is more "entropy:" outlinks on a host are often controlled by an organization's policies, while inlinks are a product of millions of uncoordinated organizations. For any given x-value (distance) in Figure 4, the distribution of overlap measurements is bimodel; that is, URL-ids close to each other tend to have either very similar or very different adjacency lists.

Figure 5 shows that ordering adjacency lists by URL-id clusters adjacency lists with similar content. To determine whether other orders based on the contents of the adjacency lists would produce better clustering, we sorted adjacency lists by Grey code [L92] order and by lexicographic order. As shown for outlinks in Figure 5, Grey code ordering does slightly better than the URL-id ordering. (Lexicographic ordering, which is not shown, does slightly worse than Grey code ordering.) However, the difference is small, and there are significant practical advantages to using the URL-id ordering: it is easy to compute; it allows compression of the *starts* array, and it makes it easier to implement the Host Database. Thus, we chose URL-id ordering.

4.2 Select and Union compression

We explored two general methods for interlist compression. In our first method, *Select*, the representative list is a previous adjacency list. In our second method, *Union*, the representative list is the union of a set of adjacency lists. Both methods encode an adjacency list either as a reference to a representative plus a set of *additions* and *deletions* from the representative, or using the absolute encoding described in Section 3, whichever uses fewer bits. Note that deletions are cheaper to encode than additions because the space of possible deletions is small (the length of the representative list) relative to the space of possible additions (the set of all URL-ids)..

Select produces good compression, but may create chains of encodings that can increase decompression time. A chain is created when an adjacency list *i* is encoded relative to another list *j*, which is encoded relative to a third list *k*, and so forth. Although these chains are guaranteed to terminate (because we only look backwards for representative lists), they can get long. Therefore, we considered a few variations of Select. *Select-K*, chooses the best representative adjacency list from among the previous *K* URL-ids' adjacency lists. *BlockSelect-K-B* limits chains by grouping URL-ids into blocks of size *B* and only considering representatives from the same block. *LimitSelect-K-L* only allows chains of fewer than *L* hops.

Union-B groups URL-ids into blocks (as in *BlockSelect-K-B*) and creates the representative for each block as the union of all adjacency lists in that block. *Union-B* can be generalized to choose a representative list which contains a subset of the URL-ids in the union, e.g., only those that appear in at least 2 of the block's adjacency lists.

4.3 Performance model results

We built a model to predict the compression achieved by each method, so that we could explore many values for *K*, *B*, and *L*. The model predicts the amount of space required by the adjacency data (but not the *starts* data), but does not actually build the encoded database. The model assumes additions are encoded using a Huffman code.

Algorithm	Bits/link		Algorithm	Bits/link		Algorithm	Bits/link
Select-2	4.77		LimitSelect-2-4	4.95		Union-5	4.55
Select-4	4.34		LimitSelect-4.4	4.51		Union-10	4.34
Select-8	4.17		LimitSelect-8-4	4.30		Union-20	4.33
Select-16	4.03		LimitSelect-16-4	4.14		Union-40	4.42
BlockSelect-5.5	5.07		LimitSelect-8-2	4.42			
BlockSelect-10-10	4.67		LimitSelect-8-4	4.30			
BlockSelect-20-20	4.36		LimitSelect-8-8	4.23			
BlockSelect-40-40	4.16		LimitSelect-8-16	4.20			

Table 1: Simulated interlist compression results

Table 1 contains results from this model for different parameter values of all four method variations. This table shows that all of the interlist compression schemes can achieve roughly the same level of compression, with high enough values of *K*, *B*, and *L*. In Link3, we chose to implement *LimitSelect-K-L* because it avoids the open-ended chain problem of *Select* and achieves better compression with short chains than *Block-Select-K-B* and *Union*. (*Union* gets worse as *B* grows because the representative lists become very large.)

4.4 *LimitSelect-K-L* implementation and results

Our implementation of *LimitSelect-K-L* uses two arrays, one for *starts* and one for *adjacency data*. The *starts* array is compressed as described in Section 3. The *adjacency data* is stored in records that have three fields, each encoded in a variable-length format. The first field, *ref*, is the relative index of the representative adjacency list. The *ref* field must be between 0 and *K*. If *ref* is 0 then the adjacency list uses absolute encoding and the deletes field is omitted. The second field, *deletes*, encodes the set of URL-ids to delete from the representative list. The *deletes* field consists of a length followed by a delta-encoded sequence of indices of representative list elements to omit. The third field, *adds*, encodes the set of URL-ids to add to the representative list. The length of this field is determined implicitly by looking at the *starts* array to determine where the next adjacency-data record starts. The *adds* field is encoded as in Link2 (see Section 3). These adjacency-data records use up to four Huffman codes: one for *ref*; two for *deletes* (one for the length of *deletes* and one for the values themselves); and possibly one for *adds*. (*Adds* may instead use the nybble code.) Each of these codes has its own code table.

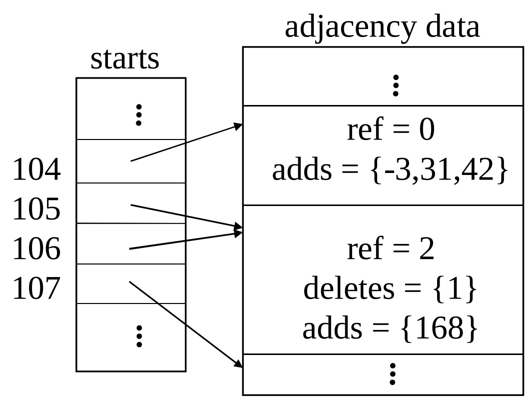

Figure 6. Interlist compression.

In Figure 6 we show the example adjacency lists from Figure 2 again, this time using the *LimitSelect-K-L* method for interlist compression.

Figures 7 and 8 show the effects of changing *K* and *L* in our implementation of *LimitSelect-K-L*. Figure 7 shows that once *K=8*, increasing K has only a slight effect on the database size but continues to increase the access time per link. Similarly, Figure 8 shows that increasing L beyond 4 has a slight effect on the database size, but results in a large increase in link access time. Based on these results, we chose *K=8* and *L=4* as our default parameters for Link3. (It would be interesting to see the performance measurements for K=2 and L=2, however.)

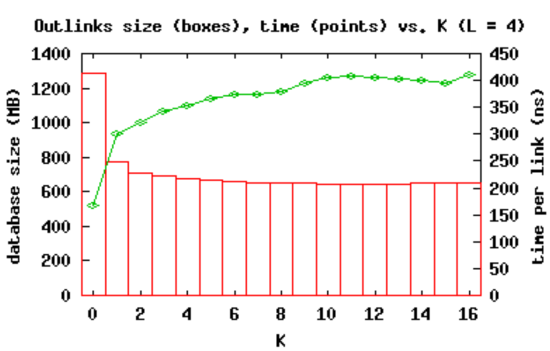

Figure 7: LimitSelect-K-4 space vs time

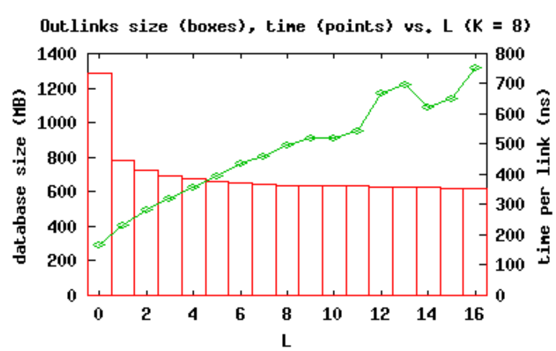

Figure 8: LimitSelect-8-L space vs time

5 Measurements

In this section we empirically compare the three implementations of the Link Database. We compare both the space requirements and the access times. Our dataset was produced by a 58 day Mercator [HN99] crawl in the summer of 2000. This crawl downloaded 332 million unique pages. The Link Database for it contains 351,546,665 URLs and 6,078,085,908 links. . A recent paper on Mercator [NW01] suggests that taking the first *N* days of a crawl is a good way to limit the amount of data to consider. Table 2 presents the results for the first 7 days of this crawl. This dataset contains 61 million URLs and 1 billion links.

Algorithm	**Size (avg bits/link)**		**Max DB (M pages)**	**Time (avg ns/link)**		**Time (s)**
	Inlinks	**Outlinks**		**Seq**	**Rand**	**SCC**
Link1	34.00	24.00	214	13	72	187
Link2	8.90	11.03	546	47	109	217
Link2-1part	9.02	12.81	488	49	117	217
Link2+huff	7.92	10.8	583	117	195	287
Link3	5.66	5.61	862	248	336	414
Link3+huff	5.39	5.55	868	278	367	451

Table 2. Space and time measurements for implementations of 7 day crawl dataset.

Table 2 presents the results. Each row presents data for a different implementation of the Link Database. Link1 uses no compression; Link2 uses single list compression and *starts* array compression as described in Section 3; Link3 uses the implementation of *LimitSelect-8-4* described in Section 4. . The "-1part" version of Link2 creates a database with 1 URL partition instead of 3. The "+huff" versions of Link2 and Link3 use Huffman codes rather than nybble codes to encode additions. (Link3 always uses Huffman codes to encode deletes.)

The first two data columns of Table 2 contain the sizes of the databases, reported as the total number of bits used by the link data, including the *starts* array and any Huffman tables, divided by the total number of links. The third data column contains an approximation of the maximum database size (in millions of Web pages) that each technique can support on a machine with 16 GB of RAM. This approximation assumes an average of 17 outlinks per URL, which is what we measured in our crawls. It also assumes that for a typical algorithm to run, such as SCC, (only) the inlinks must fit in memory, in addition to 8 bytes of algorithm-specific data per URL.

The last three columns contain timing data. All timing was done on a Compaq Alpha ES40 21264A with 2 667 Mhz CPUs and 16 GB of RAM. All measurements are the average of at least three runs with the Link Database in memory. The first two timing columns reports the time per link to complete a sequential scan and a random scan of all inlink adjacency lists. The last column contains the time to run a strongly connected components (SCC) algorithm. (A strongly connected component on the Web is a set of mutually-reachable pages: from any page in the component there is a path of hyperlinks to each other page. Our SCC algorithm [Tar72] finds all such components.)

Table 2 makes very clear the space-time tradeoff we face: each step from Link1 to Link2 to Link3 approximately doubles the number of pages we can handle on our 16 GB machine, but each step also costs us in access time. The timing results tell an interesting story. On modern machines, cache misses often dominate performance. In Link1 and Link2, the sequential scan is optimal with respect to caching behavior: the processor picks up on the sequential scan and reads ahead in memory, hiding the latency of cache misses. In this case, Link1 is clearly fastest because it requires very little processing. Link3 does not exhibit this good cache behavior because it goes marching back through memory to look at representative lists, and thus performs much worse.

When we switch to random access, at least two cache misses are introduced (one for the *starts* array, and one for the *adjacency data*), which are expensive. When this overhead is added, the relative performance gap between Link1 and Link2 begins to close. We suspect that the significant performance gains of Link2 relative to Link1 are because the compression of Link2 reduces the number of cache misses, making up for time spent decoding. The difference in performance between Link1 and Link2-1part also supports this suspicion. The overhead of the SCC algorithm itself (which includes even more cache misses) further closes the relative gap between Link1 and Link2, so much so that they are now fairly comparable. The relative performance gap between Link2 and Link3 also closes as we add more overhead, but not nearly as much.

Table 2 also illustrates why we do not use a Huffman code in practice. It saved 3-11% of space, but cost up to a factor of 2.5 in acocess time. We chose the faster option. Further, Table 2 also shows that using 3 URL partitions saves space, primarily because the *starts* array can be compressed.

Table 3 contains measurements for the full 58 day crawl for Link2 and Link3. (The Link1 implementation cannot support 6 billion links.) Although the details of the numbers in Table 3 differ from Table 2, the overall and relative trends remain the same. The timing measurements are over the inlink database. The build time for Link2 is about 30 hours, including building the URL Database. Link3 builds from Link2 in an additional 4 hours.

Algorithm	**Size (avg bits/link)**		**Time (avg ns/link)**		**Time (s)**
	Inlinks	**Outlinks**	**Seq**	**Rand**	**SCC**
Link2	8.51	11.08	45	119	1281
Link3	5.27	5.44	221	308	2509

Table 3. Space and time measurements for 58 day crawl dataset.

6 Related work

Many compression techniques exist [BYRN99, WMB98]. However, nearly all algorithms for general data compression are optimized for sequential access: in order to decompress any portion of the data, it must all be decompressed. Our goal is to support fast decompression for random access algorithms. Similarly, most algorithms are optimized for large datasets, while adjacency lists are quite small at less than 100 bytes.

Our work borrows from known techniques, such as delta codes, variable-length bit encoding, and dictionary codes. However, we also introduce novel measures specific to the Web graph dataset, such as partitioning adjacency lists into groups by the number of links they contain to compress pointers to them (the *starts* array). We also allow backward references (representative lists) which are not exact.

Although inverted list compression sounds similar – it compresses lists of document references – many techniques, like block addressing that coalesces multiple list elements, do not apply because the adjacency list of URL-ids must be exact. (Other techniques, like delta codes, we borrow.) Similarly, papers on general database compression focus on reducing disk-space and I/O requirements of disk-based databases [GRS98, WKHM00], not on memory requirements

While search engines such as Google [Google] must use compression techniques to store the Web graph, they are unpublished. Altavista [AV] has used our Link2 in production since 1999. Two other groups have published research on compressing the Web graph.

Adler and Mitzenmacher [AM01] contains a theoretical study of Web-graph compression. As in our *Select* method, they encode lists relative to representative lists chosen from among the adjacency lists of other nodes in the graph. However, rather than search locally to find representative lists, they formulate an optimization problem that minimizes the cost of encoding adjacency lists. They then reduce this optimization problem to finding a spanning tree of what they call the "affinity graph" of the Web. While their technique is not practical for use on the Web as a whole, it might be feasible if applied on a per-host basis, since our measurements show that the best representative list is likely to be the adjacency list of another URL on the same host.

Suel and Yuan [SY01] describe a Link Database with similar functionality to ours, although they only consider outlinks. They partition every adjacency list into two components, one containing global links (that cross host boundaries) and the other containing local links. For the global links, they use a Huffman code for a large number of the most popular destinations, and Golomb coding for the rest. For local links, they also use a Huffman code for the most popular destinations, and a delta code for the others. They report that this scheme requires around 14 bits per link. Although our approach does better than Suel and Yuan's, it might be interesting to see if our approach can be improved further by treating the most popular global destinations separately.

7 Conclusion

This paper describes methods for compressing the graph of URLs and hyperlinks on the Web. We present two techniques for compressing lists of hyperlinks. These techniques combine existing compression methods with new ideas to reduce the amount of space required from 32 bits to less than 6 bits per link – better than a ¼ compression ratio. Our techniques are effective on the small amounts of data in an adjacency list (an average of 68 bytes), allow random access to individual lists, and provide fast decompression speed. Using one technique, we can sequentially scan a graph of 6 billion hyperlinks in 4.5 minutes, and compute the strongly connected components (SCCs) of the graph using a depth-first search traversal in 22 minutes. Using both techniques, we can complete the sequential scan in 25 minutes, and compute the

SCCs in 42 minutes. That is, as we improve our graph compression, we increase access time, but can still compute complex algorithms over the graph quickly.

References

[**ABW98**] J. Abello, A. Buchsbaum, and J. Westbrook. A Functional Approach to External Graph Algorithms. *Proc. Annual European Symposium on Algorithms*, pages 332-343, 1998.

[**AM01**] M. Adler and M. Mitzenmacher. *Towards Compressing Web Graphs*. *Proceedings of IEEE Data Compression Conference (DCC)*, March 2001.

[**AV**] AltaVista Company, http://www.altavista.com/

[**BBDH99**] K. Bharat, A. Broder, J. Dean, and M. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. *Proceedings of the ACM Conference on Digital Libraries*, 1999.

[**BBH98**] K. Bharat, A. Broder, M. Henzinger, P. Kumar and S. Venkatasubramanian. The Connectivity Server: Fast access to linkage information on the Web. *Proc. of 7th Intl World Wide Web Conference*, 1998.

[**BK91**] A. Bookstein and S. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4), 1991.

[**BKM00**] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopolan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the Web. *Proceedings of 9th Intl World Wide Web Conference*, May 2000.

[**BP98**] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Proceedings of the 7th Intl World Wide Web Conference*, April 1998.

[**BYRN99**] R. Baeza-Yates and B. Tibeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, NY, 1999.

[**CGG95**] Y. Chiang, M. Goodrich, E. Grove, R Tamassia, D Vengroff, and J. Vitter. External-Memory Graph Algorithms. *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 139-149, January 1995.

[**Google**] http://www.google.com

[**GRS98**] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. *Proceedings of the IEEE Conference on Data Engineering*, pages 370-379, 1998.

[**HN99**] A. Heydon and M. Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, 2(4), December 1999.

[**K98**] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, January 1998.

[**KS96**] V. Kumar and E. Schwabe. Improved Algorithms and Data Structures for Solving Graph Problems in External Memory. *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, 1996.

[**L92**] T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, California 1992.

[**NW01**] M. Najork and J. L. Wiener. Breadth-first search crawling yields high-quality pages. *Proceedings of the 10th International World Wide Web Conference*, May 2001.

[**SY01**] T. Suel and J. Yuan. Compressing the Graph Structure of the Web. *Proceedings of the IEEE Data Compression Conference (DCC)*, March 2001.

[**Tar72**] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*,1(2), 1972.

[**WKHM00**] T. Westmann, D. Kossman, S. Helmer, G. Moerkotte. The implementation and performance of compressed databases. *Sigmod Record*, 29 (3), Sept., 2000, pp 55-67.

[**WMB99**] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, 1999.