Scalable Manipulation of Archival Web Graphs

Yasemin Avcular
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
yavcular@cis.poly.edu

Torsten Suel
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
suel@poly.edu

ABSTRACT

In this paper, we study efficient ways to construct, represent and analyze large-scale archival web graphs. We first discuss details of the distributed graph construction algorithm implemented in MapReduce and the design of a space-efficient layered graph representation. While designing this representation, we consider both offline and online algorithms for the graph analysis. The offline algorithms, such as PageRank, can use MapReduce and similar large-scale, distributed frameworks for computation. On the other side, online algorithms can be implemented by tapping into a scalable repository (similar to DEC's Connectivity Server or Scalable Hyperlink Store by Najork), in order to perform the computations. Moreover, we also consider updating the graph representation with the most recent information available and propose an efficient way to perform updates using MapReduce. We survey various storage options and outline essential API calls for the archival web graph specific real-time access repository. Finally, we conclude with a discussion of ideas for interesting archival web graph analysis that can lead us to discover novel patterns for designing state-of-art compression techniques.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

The web graph is a directed graph structure where vertices represent the documents and edges represent the hyperlinks. This structure is commonly used for the analysis of the web. For example, web search engines crawl the web and construct a web graph for the collected information. By analyzing this graph, an importance score is estimated for each document to provide the best search results. Some of the well known examples of rank based algorithms are PageRank [29] and HITS [21]. Other than document importance score estimation, the web graph analysis also plays a very important role to solve problems such as spam detection [1] and related page identification [13].

Many frameworks are designed for manipulation of the web graphs. In general, they fall into two groups, the ones designed for online computations with providing real-time fast access, and the ones designed for offline analysis of large-scale graphs. The Connectivity Server [3] and several others [34, 32, 31, 4, 6] are examples of online frameworks providing efficient real-time access to the graph via simple interfaces. The common ground among these frameworks is first preprocessing the initial web graph and then compressing the resulting structure. This way, larger portions of the graph fit into main memory and can be accessed quickly. In the preprocessing step, every page is labeled with a unique numeric identifier (*id*). The unique ids should be assigned in a way so similar pages have consecutive numbers to archive better compression. One way to assign ids is simply sorting the documents in lexicographical order and another way is clustering the documents according to their similarity and then mapping these clusters onto linear space. On the other hand, several examples of the offline computation frameworks are MapReduce [12], Dryad [19], Pregel [26], and Pegasus [20]. MapReduce and Dryad are highly scalable, generic computation models commonly used for graph analysis, such as PageRank computation. Recently specialized graph oriented frameworks have been proposed, which provide simple interfaces for performing large-scale (offline) computations on web graphs.

While a lot of work is focused on current snapshot of the web, there is an increasing interest in archival web graphs. Archival web graphs include detailed historical information about the web and can improve performance of several tasks, such as document ranking [2], spam detection, broken link identification [33]. Also, analysis of the evolving link structure sheds light on design of other applications, for example web search engines detect importance of a newly crawled page and model smart incremental crawl policies accordingly [11, 10]. Moreover, [7, 28, 22, 16] reported interesting characteristics of the archival web graphs, such as the birth and death rate and the life length distribution of web pages, factors influencing persistence of a web page over time. Compared to the regular web graph, the archival web graph with multiple snapshots of the web is much larger in terms of size. Therefore, the archival web graph representation requires additional interfaces for accessing and use of special techniques for compressing the version information.

In this paper, we study scalable ways for building, updating and analyzing archival web graphs using MapReduce framework. We also propose a simple representation format that is compatible with efficient storage schemes. Our discussion follows the outline shown in Figure 1. We first discuss how we extract the initial version of web graph format (b) using the raw web graph data (a). Next, the initial web graph format is further processed to generate a layered graph representation (c) which serves as an efficient structure for both online and offline computations. This representation is also used for applying incremental updates. Finally, considering a set of analysis algorithms (f), we discuss the large-scale storage systems (d) for online computations and the distributed computing frameworks (e) for offline computations as the future directions.

LSDS-IR'11, October 28, 2011, Glasgow, Scotland, UK.

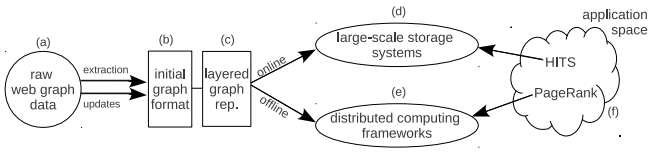

Figure 1: Outline of Execution Flow

The paper is arranged as follows: Section 2 introduces previous work in more detail; Section 3 describes how the web graph is built using MapReduce; Section 4 discusses the layered representation; and, in Section 5 shows how updates can be applied using MapReduce framework. Experimental results are presented in Section 6; Section 7 introduces several options for storage and access schemes with useful API calls and Section 8 presents our conclusions and identification of future work.

2. RELATED WORK

Repositories specifically designed for the web graphs provide real-time access for online computations. To the best of our knowledge, Connectivity Server [3] developed at DEC SRC is the first study that describes a representation of web graph for fast random access. The idea of preprocessing the web graph and representing documents with a unique numeric id is first introduced with the Connectivity Server. For the id assignment step, first all unique documents are sorted lexicographically within a single machine and then the ids are assigned in increasing order. The document-id mapping is later compressed using delta encoding. After to Connectivity Server, more work focused on efficiently compressing the web graph, so large portions can fit in main memory for fast access.

LinkDB [32], a follow-up study from DEC SRC partitions the documents into three groups according to their degree, and then sorts each group lexicographically to assign ids. The ids assigned first by the group's id and then by the document's position in the group. This way documents with high degree are assigned lower ids and overall structure can be compressed efficiently using *starts array compression*. Three versions of LinkDB developed using several compressing techniques such as delta codes, inter-list compression (similar to dictionary codes), variable-length nybble codes and Huffman codes. The WebGraph Framework [4, 6], a state-of-art compression technique for the web graph, uses gap encoding for the graph representation and then applies inter-list compression benefiting the similarity between adjacency lists of lexicographically ordered urls. Later arithmetic encoding is used to compress numeric ids.

Raghavan and Garcia-Molina introduced a new format for the web graph representation, called S-Node [31]. With this technique, documents are partitioned into chunks called *Supernodes* using two methods; URL split and clustered split. First URL split method applied to partition the documents according to url prefixes, and then clustered split method used to partition further into fine grain pieces using k-means clustering algorithm. Adjacency lists are used as the similarity measure for *k-means* algorithm. After partitioning step is completed, Huffman-based compression, reference encoding and other bit level compression techniques such as gap encoding and run length encoding (RLE) are applied to compress the resulting S-Node representation. More recently, Najork designed a scalable, fault tolerant storage system for the web graphs, called Scalable Hyperlink Store (SHS) [27]. SHS is composed of *cells* in which set of documents are stored along with their mapping and adjacency list. The id mapping and adjacency list are compressed using front encoding and gap encoding respectively. Additionally, the id mapping is further compressed using variable-nybble and variable-byte codes.

According to the representation techniques used in a web repository, various access schemes are designed to provide useful interfaces. For example, based on the S-Node representation model, Raghavan and Garcia-Molina developed a cost based optimizer and execution engine to efficiently execute web queries over large repositories [30]. They modeled a specific algebra for expressing complex web queries. On the other side, the SHS store defines the set of user interfaces for random access and real-time updates. In the SHS model, the graph is distributed among nodes to balance the workload and provide quick access.

To perform offline computations on the web graph, general purpose large-scale distributed computing frameworks such as MapReduce [12] and Dryad [19] are used. MapReduce [12] is a fault tolerant, highly scalable framework inspired by functional programming where map and reduce functions are commonly used. MapReduce provides a simple framework for designing embarrassingly parallel computations running on large data sets. This model is found to be efficient in implementing the offline web graph algorithms, such as PageRank. More recently, frameworks specific to the web graphs, Pregel [26] and Pegasus [20] are proposed. Pregel is an efficient, scalable, fault tolerant computation model for analysis of large-scale graphs. Within this model, a computation is consist of multiple iterations in each of which, every node can pass a message to its neighbors and these messages are used in the next iteration. Pegasus, an open-source library, developed at CMU, is implemented on top of Hadoop. This library provides implementation of large scale graph mining algorithms, such as PageRank.

Several techniques are proposed for the archival web graph representation. With the TimeLink [23] representation, each link is attached a start and an end date. The end date is defined as the date of the latest crawl where the link existed. Using this representation, evolving link structure of 7.2 billion link graph from 1995 to 2001 is analyzed. Bordino et al. [7] analyzes evolution of the UK web using monthly snapshots from May 2006 to 2007. This data set is collected for the DELIS project and its building details are described in [5]. The WebGraph compression framework [4, 6] is used for the representation of this collection. The WebGraph framework was originally designed for regular web graphs. For including the time information, a single bit vector of length 12 is attached to each link to identify existence of the link at a given month. These bitvectors are separately compressed with Huffman codes. Moreover, hyperlink birth and death rates are studied in [16], and the life cycle of web objects including hyperlinks is studied in [22].

3. GRAPH CONSTRUCTION

The raw web graph data (a) consists of many captures. A capture is a one time retrieval of a specific document holding all information about the specific retrieval, including the time stamp, and a list of documents linked[1]. In the initial graph format (b), each document is identified with two attributes; a *url*, the string representation, and a *uid*, the unique numeric identifier. Initially, the documents in the raw web graph (a) are identified only by their urls and in this section, we explain how to construct an equivalent web graph where documents are identified by their uids which is called the initial graph format (b).

The graph construction consists of two main steps, a *uid assignment* step, where a unique uid is assigned for each distinct url, and a *replacement* step, where each url in the web graph is replaced with its corresponding uid. A sequential solution to this problem would be to construct a url-uid mapping in a single machine by first sorting and identifying all the unique urls, and then assigning uids for each url starting from zero in increasing order. Later, this mapping can be loaded into memory and each url in the raw web graph (a) can be replaced with its corresponding uid via traversing the whole

[1] *destination* document list

web graph and using the url-uid mapping in the memory. However, with large data sets this solution is not acceptable since the size of the url-uid mapping overgrows the memory size. We implemented a parallel graph construction algorithm within MapReduce to address this problem. Being in a distributed environment adds one new preparation step to overall computation; *partitioning*. Before graph construction starts, the data should be partitioned in a smart way so required communication among all nodes is minimized and also the work load across all nodes is balanced.

Partitioning: If web graph could be partitioned in a way so each document appears only in a single node, then there would be no network traffic required and each node could independently run the sequential graph construction algorithm (explained above) without any conflicts. Due to the nature of web graphs, partitioning them into perfectly disjoint sets is not possible. Using the fact that very high percentage of all the linked documents are *local*; meaning they are in the same domain with the source document, we partitioned the web graph according to the domain of source document. Even though this approach minimizes the required communication among nodes, we observed unbalanced workload [25] due to zipf distribution on the number of pages each domain has. As a solution, we identify a number of boundary urls for partitioning according to the total order. First a super-sample is generated with random sampling of the raw input graph (a). Then the super-sample is sorted and systematic sampling is applied to generate the range boundaries. According to these range boundaries and the predefined number of reduce tasks, any MapReduce job can easily setup its partitioning function for balanced data distribution. Once the partitioning function is defined, the subsequent MapReduce jobs of the graph construction algorithm use the same function to minimize the shuffling cost. Since the partitioner function is defined using the total order, output of the MapReduce jobs using this partitioner function is also sorted in total order.

Uid Assignment: First, the unique urls are extracted with a single MapReduce job that scans through the raw web graph data (a). The urls produced as the result of this MapReduce job are in total order due to the partitioner that was designed in the previous step. As the next step, prefix sum of each range defined in the range boundary file is computed to calculate an offset for each range. Once the offset values are calculated, another MapReduce job loads them into memory and reads the unique urls file as the input to compute the uid of each url. As the output of this MapReduce job, the url-uid mapping is generated.

Replacement: With the raw web graph data (a) consist of captures and the url-uid mapping in hand, each url in the web graph is replaced with its corresponding uid within two MapReduce jobs, as shown in Figure 2 with an example. The first MapReduce job takes

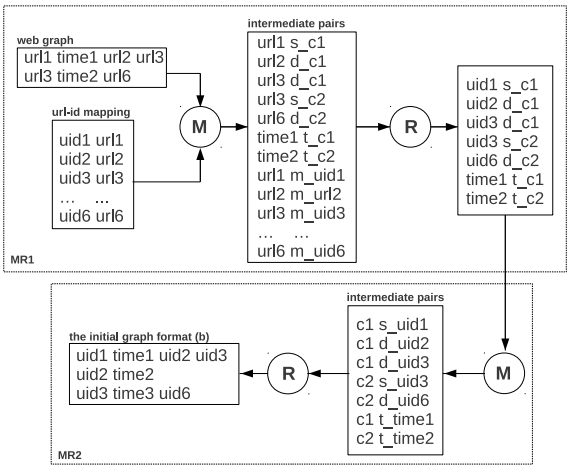

Figure 2: Graph Construction with MapReduce

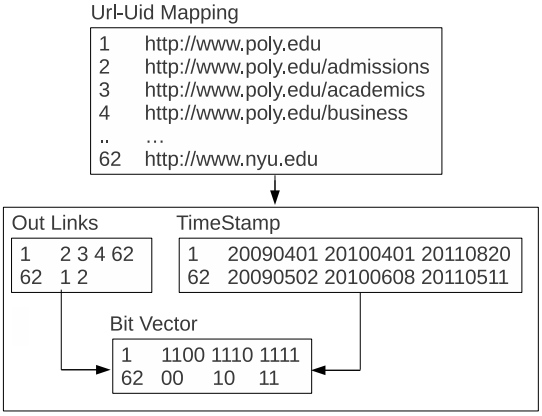

Figure 3: Layered Graph Representation

these two files as input, and generates <url, payload> tuples as the intermediate pairs. For each document in the url-uid mapping, one tuple is generated with the url as the key; and for each capture in the raw web graph, at least two tuples are generated, one for the source url, one for the time-stamp[2], and one for each destination url if there is any. Since keys are the urls, all the payload information belongs to the same url is processed within a single reducer. If a given <key, payload> tuple is generated from the graph file, then the payload includes the capture id[3]; if the tuple is generated from the mapping file, then the payload includes the uid of the key. Each payload is also attached a type identifier[4] to specify the capture and mapping attributes. According to the type identifiers, each each reducer replaces its key, a url value, with corresponding uid and outputs <uid, payload> tuples. This time, the payload of each tuple includes a type identifier along with a capture id. The second MapReduce job receives these tuples and outputs <capture-id, payload> tuples as the intermediate pairs. With selecting the capture id as the key, all information belongs to a single capture is collected within one reducer to build the initial web graph format (b).

4. GRAPH REPRESENTATION

We further structure the uid based initial web graph format (b) and construct the layered web graph representation (c) by separating the time stamp and the destination document list attributes attached with each capture to achieve a good compression and provide a generic format for the use of both online and offline computations. This representation organizes information in three layers, *Out Link (OL)*, *Time Stamp (TS)* and *Bit Vector (BV)*.

Every layer holds a layer specific information about each unique url crawled. For every crawled document, OL layer holds the union of all destinations document seen at any of its captures. The set of destination documents in the union are stored in sorted order according to uid values. Similarly, TS layer holds a set of time stamp values for each document. The time stamp set is also stored in sorted order. Finally, the BV layer glues the information in the OL and the TS layers. Each bit vector in BV layer has a corresponding time stamp in TS layer and identifies which destination documents appeared at that specific time stamp. Figure 3 shows relationship among layers with a sample graph. Once the layered representation is generated, each layer is compressed to achieve space-efficient storage. The compression phase also handles increasing bitvector length due to new versions being appended over time.

[2]Since the time-stamp tuples do not require any replacement, they can be written directly to the disk without sending to the reducer. They will still be used by the second MapReduce job.

[3]The capture ids are generated by concatenating the source document url and time stamp and hashing this value with MD5

[4]The payload identifiers separated by a _: *s*, *d* , *t*, *m*, and *c#* stand for source url, destination url, time-stamp, mapping, and capture id respectively.

Data Sets	Captures	Links	Unique Nodes	
			Overall	Crawled
Jun'06-May'07	109.6M	4.0B	43.0M	9M
Jun'06-Jan'07	36.5M	1.2B	41.2M	9M
Feb'07-May'07	73.0M	2.7B	42.5M	9M
archive.org	223M	7.4B	928M	170M

Table 1: Summary of data sets in numbers[5]

Data Sets	Mapping	Layers			
	URL-ID	OL	BV	TS	TS'
Jun'06-May'07	410M	242M	142M	27.7M	541M
Jun'06-Jan'07	406M	240M	104M	26.4M	363M
Feb'07-May'07	395M	243M	68M	26.0M	187M
archive.org	10.6G	6,1G	835M	1.2G	-

Table 2: The Layer Size

These three layers are constructed with a single MapReduce job that reads the web graph of uids as the input. As mentioned before, all three layers hold the information about the same set of documents, and the information in each layer are aligned using the exact same ordering. The layered representation (c) is ideal for processing and analysis in MapReduce and related frameworks (e) for offline computations due to this design decision. Also, the layered representation can be loaded easily into a large-scale storage system (d) providing access for online computations. Moreover, each layer is nicely compressible because the layered representation bundles the similar information into separate layers. Finally, as we show in the next section, quick updates can be implemented using these layers.

Three main layers (OL, TS and BV) along with the url-uid mapping are enough to represent the initial web graph without losing any information. However, for various applications, such as [21, 9, 13], constructing an in-link structure is necessary. Applying a similar three layer approach for in-link representation results in a very sparse bit vector matrix due to unshared time stamp values. Indeed, if the data is coming from one single crawler, then there is only one in-link exists per time stamp. Due to this characteristic, we propose to use time ranges rather than exact time stamps for in-link representation. The exact range can be specified according to the application requirements. Overall, in-links are represented with three layers, *In-Link (IL)*, *Time Range (TR)*, and *In-Link Bit Vector (IBV)* layers. IL layer holds the union of all documents pointing the specified document in any capture. This list is stored in sorted order. Each time stamp in TR layer indicates the start time of the prefixed time range value. The list of time stamps are also stored in sorted order. Similar to BV layer, the IBV layer connects IL layer with TR layer using a single bit vector for each time range in the TR layer. Each bit vector identifies the set of documents pointing the specified document in that specific time range. A single MapReduce job can also be implemented to build the three layers of the in-link representation.

5. UPDATES

The web crawlers continuously collect the most recent data from the web and update the existing web graph using this up-to-date information. The update process would be simple if every url had a universally unique id without any specific requirement. However inserting a new url into the layered web graph representation (c) is more expensive operation due to the lexicographically assigned uids. Inserting one url in the middle of the url-uid mapping would require each following url to increase its value.

With relaxing the lexicographical id assignment requirement, new urls could be sorted among themselves and assigned increasing uids starting from the first uid value currently available in the graph representation. This way the existing web graph can be updated safely. However, this technique comes with a cost; a badly compressed web graph representation. Sticking with ordered id assignment, we implemented a chain of MapReduce jobs to allow efficient updates on the layered web graph representation (c).

The update procedure starts with generating a layered web graph representation (c) for the new raw web graph data (a) via running the graph construction algorithm described in Section 3. With the existing and the newly constructed layered web graph representations in hand, the updated web graph representation is constructed in three main steps; *global id assignment*, *old uid replacement*, and *the big merge*.

In the first step, *global id assignment*, three mapping files are generated, the global url-uid mapping for the updated web graph representation, and two old-uid to global-uid translation tables one for the existing representation, and one for the new representation. At first, the unique urls along with their current uids are extracted from the existing and the new url-uid mappings within a single MapReduce job. Then, the global uids are assigned for each unique url using the same procedure described in Section 3. And finally, the global url-uid mapping and the two translation tables are extracted using a separate MapReduce job. In the second step, *old uid replacement*, the uids in the existing and in the new layered representations are replaced with their corresponding global uids, using the translation tables. For the replacement of OL layers, the implementation from Section 3 is used. Replacement of TS and BV layers is simpler and implemented within a single MapReduce job. In the final step, *the big merge*, all three layers from both representations are merged using a single MapReduce job, and the layered representation for the updated graph is generated.

6. EXPERIMENTS

Throughout this work, the data set provided by DELIS [5] project and the data set we received from archive.org are used. The DELIS data set consists a union of 12 month snapshots from UK domain between June 2006 to May 2007 and archive.org data set has captures of UK domain from 2004 to 2007 without any prefixed snapshot interval. For the experiments, we used 9 million node subset of the DELIS data set and its several partitions; the first 8 months split, the last 4 months split. Table 1 summarizes details of both of the data sets and splits.

Algorithms for the graph construction, extracting the layered representation and updates are implemented within Hadoop [17], an open source MapReduce framework. The experiments conducted on a 6 nodes cluster, each having four 2.83GHz cores and 16G memory.

For experimenting scalability of the construction algorithm, both the four and eight month splits and the twelve months full data set are used. Results show time taken to generate the representation is increasing linearly to the growing number of links in the input data. Figure 4, plots time taken to generate the layered graph representation (c). Each layer is compressed using gzip compression and Table 2 shows the layer sizes of each data split. Size of the url-uid mapping increases proportional with the total number of unique documents and similarly the size of BV layer increases proportional to the number of links each data set has. Size of the TS layer is also expected to increase proportional to the average number of captures per document. This is not observed with the DELIS dataset because of monthly basis time stamps are used; eg. there are only 12 different time-stamp values existing. Therefore, each document has the same set of time stamps in each data split which is embarrassingly compressible. To show the results of fine

[5]M=Millions B=Billions

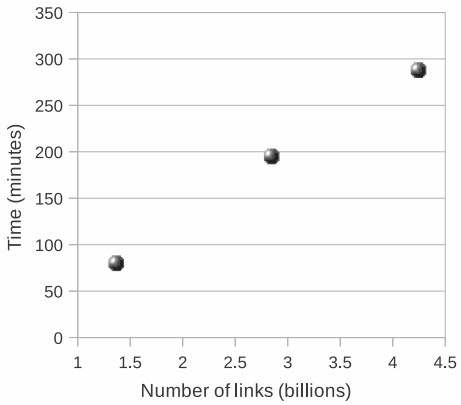

Figure 4: Scalability of Graph Construction Algorithm

grained time-stamps, we simulated second-based time stamp values with randomly assigned day and time values for each capture. The results are listed in the TS' column. Finally, the non-varying OL layer size can be explained in two ways; the graph splits do not have much of a link flux, the outlink lists compress well.

With the layer representation of each graph split, we also conducted experiments for updates. The first 8 months split is updated with the last four months split, and the results are shown in Figure 5. We observed that the update algorithm is very efficient and the actual merging part is very quick compared to the graph construction parts. Indeed, this experiment shows building the 12 months graph via updating is even faster than building it from scratch. To be able to better analyze the efficiency of updates, we also run another experiment using four 3 months splits. Initially, the first two 3 months splits are merged, and then the third and the fourth split are merged one by one. The results are shown in Figure 6. This time, building with updates is slightly slower than building from scratch. However the actual merging part is still very fast. We concluded that partitioning the web graph into very small pieces results a slower overall performance because of the overhead. If there are more than two versions are exists, similar to the situation in the second experiment, rather than merging them one by one, merging all of them at once, as in external n-way merge technique of I/O efficient computing, would result in better performance.

7. STORAGE AND USER INTERFACE

As discussed before, the layered web graph representation is very suitable for processing and analysis using distributed computing frameworks (e); such as Hadoop, a MapReduce based framework. While offline computations can be performed within frameworks similar to MapReduce, the layered representation can also be used for online computations via loading into a large scale storage system (d) and providing an API similar to the Scalable Hyperlink Store (SHS) by Najork. Below, we listed the necessary API calls for the archival web graph analysis:

iterator<int> getUids()
iterator<int> getUids(datetime start, datetime end)
int getUid(string url)
string getUrl(int uid)
int[] getLinks(int uid, bool isOutlinkList)
int[] getLinks(int uid, datetime start, datetime end, bool isOutlinkList)
int getLinkCount(int uid, bool isOutlinkList)

The getUids call returns an iterator for the set of all documents; and if the time arguments are specified, it returns an iterator for the documents crawled only in the given specific time range. For url to uid and uid to url translations getUid and getUrl calls are provided. To provide the two-way url-uid translation, only Uid-Url mapping is stored. This was, uid to url translation is accessed in constant time and for url to uid translation, a binary search wrapper can be

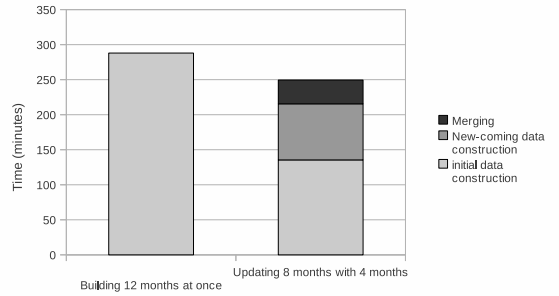

Figure 5: Updating first 8 months data with last 4 months

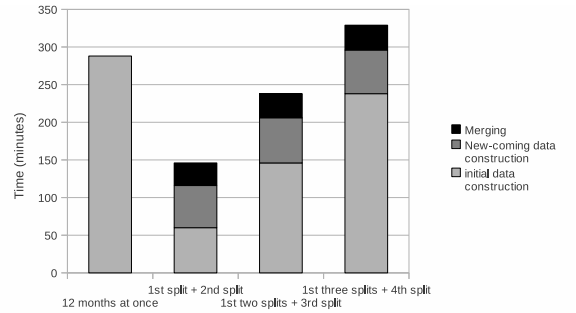

Figure 6: Updates with four 3 months data splits

implemented. Since uids are assigned to lexicographically sorted urls, binary search wrapper simply runs on uids to find the right url. Access to outlink and inlink lists is provided with getLink call. The boolean parameter sets the link direction. Links seen within a specific time range can also be retrieved via using datetime parameters. getLinkCount returns number of links from the specified direction. Additional to API calls introduced with SHS, time-aware queries are supported for archival analysis with getLinks and getUids methods.

To support listed API calls for online applications, use of a distributed, highly scalable, fault tolerant repository is needed. To the best of our knowledge, there is no large-scale repository specifically designed for archival web graphs. The Scalable Hyperlink Store by Najork is a distributed repository designed for single version web graphs. On the other hand; there are large-scale, highly scalable, distributed storage systems [8, 15, 14] proposed for general-purpose data storage, known as non-relational storage services. The design of the layered representation allows use of an open sourced non-relational storage system for real-time access to the archival graph.

The non-relational storage services are mainly inspired by two early models, Google's BigTable [8] and Amazon's Dynamo [14]. BigTable introduced a new data model, called column families, where the data is stored with multiple columns in sorted order. Dynamo has a simpler data model, called key-value storage, where each key is associated with a value block. To use the advantages of nicely structured layered representation, the data should be loaded into the system in sorted order using a user-specified partitioner function. Also, information from all layers belonging to a specific document should be written next to each other in disk level, since they will be fetched together for most of the scenarios. Considering these requirements, we survey the three open-sourced candidate systems; HBase [18] as the candidate of column-family model, Project Voldemort [35] as the candidate of key-value model, and Cassandra [24] as the hybrid model.

Voldemort is a key-value store that does not support storing information, the <key, value> tuples in sorted order. However information from all layers belonging to a specific document can be bundled together into the value of a <key, value> tuple. HBase, is an open source implementation of BigTable on top of Hadoop.

Since we generate the layered representation within Hadoop, one can use the advantage of staying in the same environment. Order-preserving partitioner is supported with HBase and the set of values from each layer can be written into a column family for quick access. Cassandra also allows user-specified partitioner functions, and additionally supports range search on keys. Similar to BigTable, Cassandra also supports sorted column families. However, it is implemented out side of Hadoop.

One of these systems can be chosen according to application's requirements, and the layered representation of the data set can be directly loaded into the storage system for computation of online algorithms using real-time access. Also, as discussed in the next section, we will be using a non-relational storage system as the core of the scalable archival web graph repository.

8. CONCLUSION AND FUTURE WORK

Throughout this work, we showed an efficient way of building archival web graphs, and proposed an efficient representation for online and offline analysis. We also discussed a fast way to perform updates on the archival web graph. Our experiments show graph construction algorithm scales well with the increasing data size and the layered graph representation is efficiently compressible.

We are currently setting up a 50 nodes cluster and will be using this cluster for running the archival web graph construction framework on the full version of DELIS data set and also on the data set collected by 20th century project at Internet Archive. Next, we will be analyzing these large datasets to discover interesting patterns of archival web graphs. Our plan is to first run experiments for *link updates*. For this experiments, only the BV layer alone provides necessary information. Also, combination of OL and BV layers can be used to discover interesting *frequent patterns* in adjacency lists, such as local and global menu detection give us hints for design of better compression techniques.

We will also be building a scalable, fault tolerant, efficient repository specifically designed for the layered web graph representation. Rather than building such a repository from scratch, one of the state-of-art scalable repositories from our discussion in Section 7 will be used as the core component providing all fundamental specifications of a distributed system. To achieve better performance, additional layers will be developed on top of the core system according to characteristics of the layered graph representation; for example, integrating web graph specific compression techniques, designing specific index structure for two-way mapping queries.

Acknowledgements

We would like to thank Shuai Ding and Jinyang Li for helpful discussions. We also thank NYU NEWS Group for providing access to their Hadoop cluster, the Internet Archive for providing access to the Ireland data set, and LAW at University of Milano for providing access to the DELIS data set [5]. This research was supported by NSF Grant IIS-0803605, "Efficient and Effective Search Services over Archival Webs".

9. REFERENCES

[1] Luca Becchetti, Carlos Castillo, Debora Donato, Ricardo Baeza-Yates, and Stefano Leonardi. Link analysis for web spam detection. *ACM Trans. Web*, 2008.

[2] Klaus Berberich, Michalis Vazirgiannis, and Gerhard Weikum. T-rank: Time-aware authority ranking. In *In WAW*, pages 131–142, 2004.

[3] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The connectivity server: fast access to linkage information on the web. In *Proc. of the Seventh Int. Conf. on World Wide Web 7*.

[4] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proc. of the 13th Int. Conf. on World Wide Web*.

[5] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 2008.

[6] Paolo Boldi and Sebastiano Vigna. The webgraph framework ii: Codes for the world-wide web. In *Proc. of the Conf. on Data Compression*, 2004.

[7] Ilaria Bordino, Paolo Boldi, Debora Donato, Massimo Santini, and Sebastiano Vigna. Temporal evolution of the uk web. In *IEEE Int. Conf. on Data Mining*, 2008.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.

[9] Yen-Yu Chen, Qingqing Gan, and Torsten Suel. Local methods for estimating pagerank values. In *Proc. of the thirteenth ACM Int. Conf. on Information and Knowledge Management*.

[10] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. of the 26th Int. Conf. on Very Large Data Bases*.

[11] Anirban Dasgupta, Arpita Ghosh, Ravi Kumar, Christopher Olston, Sandeep Pandey, and Andrew Tomkins. The discoverability of the web. In *In Proc. of the 2007 Int. Conf. on World Wide Web*.

[12] Jeff Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, 2004.

[13] Jeffrey Dean and Monika R. Henzinger. Finding related pages in the world wide web. In *In Int. World Wide Web Conf.*, 1999.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 2007.

[15] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: an active distributed key-value store. In *Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation*, 2010.

[16] Daniel Gomes and Mário J. Silva. Modelling information persistence on the web. In *Proc. of the 6th Int. Conf. on Web Engineering*.

[17] Hadoop. http://hadoop.apache.org/core/.

[18] HBase. http://hadoop.apache.org/hbase/.

[19] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 2007.

[20] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proc. of the 2009 Ninth IEEE Int. Conf. on Data Mining*.

[21] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46, 1999.

[22] Wallace Koehler. Web page change and persistence—a four-year longitudinal study. *J. Am. Soc. Inf. Sci. Technol.*, 53, 2002.

[23] Reiner Kraft, Enes Hastor, and Raymie Stata. Timelinks: Exploring the link structure of the evolving web. In *In Second Workshop on Algorithms and Models for the Web-Graph*, 2003.

[24] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44, 2010.

[25] Jimmy Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *Proc. of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR'09) at SIGIR 2009*, Boston, Massachusetts.

[26] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the 2010 Int. Conf. on Management of Data*.

[27] Marc Najork. The scalable hyperlink store. In *Proc. of the 20th ACM Conf. on Hypertext and Hypermedia*, 2009.

[28] Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the web?: the evolution of the web from a search engine perspective. In *Proc. of the 13th Int. Conf. on World Wide Web*.

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[30] Sriram Raghavan and Hector Garcia-Molina. Complex queries over web repositories. In *Proc. of the 29th Int. Conf. on Very Large Databases - Volume 29*.

[31] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. In *Int. Conf. on Data Engineering*, 2003.

[32] Keith Randall Raymie, Keith H. Randall, Raymie Stata, Rajiv G. Wickremesinghe, and Janet L. Wiener. The link database: Fast access to graphs of the web. In *Research Report 175, Compaq Systems Research*, 2001.

[33] Ellen Spertus and Lynn Andrea Stein. Squeal: a structured query language for the web. *Comput. Netw.*, 2000.

[34] Torsten Suel and Jun Yuan. Compressing the graph structure of the web. In *Proc. of the Data Compression Conf.*, DCC '01.

[35] Project Voldemort. http://project-voldemort.com/.