Efficient Scalable Temporal Web Graph Store

Khoi Duy Vo[1], Sergej Zerr[2], Xiaofei Zhu[3], and Wolfgang Nejdl[1]
[1]*L3S Research Center, Leibniz University of Hanover, Hanover, Germany*
[2]*University of Bonn, Bonn, Germany*
[3]*College of Computer Science and Engineering, Chongqing University of Technology, Chongqing, China*
[1]{khoi, nejdl}@l3s.de, [2]szerr@uni-bonn.de, [3]zxf@cqut.edu.cn

***Abstract*—Temporal web graphs have been attracting much attention recently due to their important applications in web search, data mining, and social network analysis. Accumulated over long periods, those graphs have grown gigantic in size and rich in temporal evolution, which poses tough challenges for data storage and management. Though a few temporal graph management systems were previously proposed, none of them can simultaneously satisfy both essential requirements when retrieving on temporal web graphs: *very large data scalability* and *very low querying latency*.**

In this work, we address the above gap in existing works by developing a highly efficient temporal graph management system which is dedicated to web graphs. To this end, we greatly extend the most efficient framework for managing large static web graphs to handle temporal information using the property matrix while preserving most of the outstanding features of the base framework. Ultimately, our proposed system can achieve a nearly instant response for vertex-centric temporal retrieval while still being scalable to huge datasets. Experiments on a real-world dataset with more than 43B nodes and 317B links show that using a small non-dedicated cluster, our system can reach a reduction of data storage space up to 88% of raw data size and reduce the retrieval time by 20%, compared to the baselines. We also demonstrate that our system also yields a significant reduction of computational costs for many graph ranking algorithms.

***Index Terms*—temporal graph representation, graph index, archival search, compression, distributed system**

I. Introduction

Graph storage and analysis have been core subjects in many research areas, including machine learning, web search, and social network analysis. Efficient systems for managing and data retrieval with large graphs are therefore highly demanded. Developing such a system has been attracting much research in recent years, e.g. [1]–[11].

Among different types of graphs, the ones resulted from the temporal evolution of the web pose even more challenges to manage due to their large size and constant evolution. In such graphs, meaningful analysis is only possible when utilizing the graph at the states as it was found at the interesting time points. For example, in Fig. 1, we show a temporal web graph which is much more complicated than its static version. In addition to *scalability*, this complication further requires managing systems to be able to provide temporal retrieval functionalities with very low querying *latency* [7], [12]–[14].

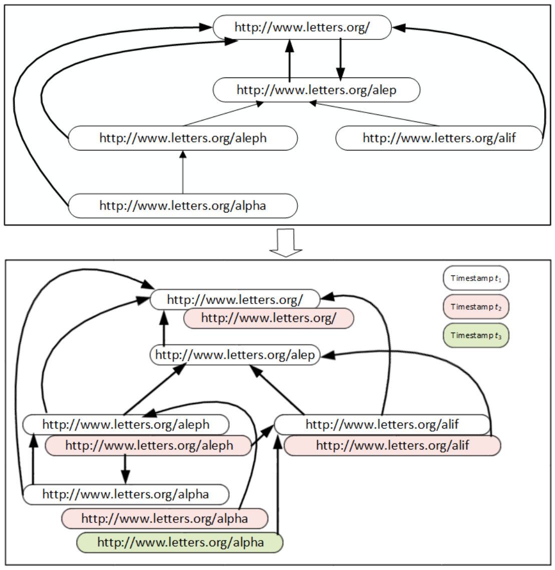

http://www.letters.org/
http://www.letters.org/alep
http://www.letters.org/aleph
http://www.letters.org/alif
http://www.letters.org/alpha
Timestamp t_1
Timestamp t_2
Timestamp t_3
http://www.letters.org/
http://www.letters.org/
http://www.letters.org/alep
http://www.letters.org/aleph
http://www.letters.org/aleph
http://www.letters.org/alif
http://www.letters.org/alif
http://www.letters.org/alpha
http://www.letters.org/alpha
http://www.letters.org/alpha

Fig. 1: An example of a web graph (upper) and its complication over time (lower)

Let take Internet Archive[*](IA) as an example. As the vast storage of past knowledge, IA is among the biggest temporal datasets and has also been attracting attention in various research fields. Unfortunately, even advanced models and engines are incapable of handling data sizes at that scale with low latency. As a consequence, extracting temporal information from IA remains a challenging and time-consuming task. Several works suggest overcoming the scalability problem by applying algorithms such as HITS and PageRank on temporal web graphs extracted from archives to improve the ranking of the search results without knowledge about the content by employing such graph information [15], [16]. However, these works did not tackle the temporal dimension and have to resign in front of the graph evolution problem.

Deal with the above graph evolution problems, there exists a trade-off between *scalability* and *latency*, since high scalability requires compact data in memory and storage, whilst low latency prefers less compressed data to reduce computation. From one side, many attempts [3], [7], [17]–[22] were concerned with the scalability of the graph and focus on compacting graph representations that fit well in storage and main memory, on the cost of latency. On the other side, graph-based retrieval approaches typically focus on building efficient data retrieval structures, but lack scalability [10], [11], [23]–[25] showed in their experiment. There are very few existing

[*] https://archive.org/

works that address both *scalability* and *latency* in working with temporal graphs [11], [25]. These works, unfortunately, require specific information and context, e.g., user profile graph or map graph, thus can not be directly applied for temporal web graphs.

In this paper, we, therefore, would like to address this gap in existing works. Specifically, we propose a highly efficient storage management system with enhanced data structure and build-in methods that are optimized and balanced for both *scalability* and *latency* and can simultaneously master the time dimension. Our approach is based on the existing Scalable Hyperlink Store [2], which is the most efficient distribution system that inherits the partitioning, mapping, encoding, compacting, compressing features to deal with *scalability* and *latency* requirements. Our work can also be directly applicable for *scatter-gather* algorithms such as HITS, SALSA, WCC, SCC, and similar algorithms [3], [5], [26]–[28]. Experiments on diversified real-world temporal web graphs prove the efficiency of our approach on large temporal web graphs. In summary, the main contributions of this work are as follow:

- **Formulation**: We present a temporal graph storage model represented by adjacent lists and adjacent pairs, which is capable to store temporal information.
- **Functionality**: We provide a solution to optimize index/retrieval operation on both *scalability* and *latency* of large-scale temporal web graphs. The performance of the system shows improvement over the ad-hoc temporal enhancement of state-of-the-art *vertex-centric* static graph management systems.
- **Praxis**: Our approach is built and evaluated on a large scale temporal web graph extracted from a German Web Archive with more than 43 billion nodes and with a total of 317 billion revision edges. Since there is a lack of similar system in temporal web literature, we empirically analyze the performance of the system using the following two methods: **(1)** Measure the system with respect to different characteristics such as reliability, durability, and robustness. **(2)** Measure the system in a real context using some stressful algorithm applications: HITS and SALSA.

The paper is organized as follows: We briefly review closely related work Section II. Next, we give an overview about state-of-the-art studies in static web graphs in Section III, which is the base of our work. We then present our work in Section IV, followed by the algorithm complexity is analyzed in Section V. Section VI presents the evaluation methodology and experimental results on the *functionality* and *practicability* of our work. Finally, in Section VII, we discuss the strengths and drawbacks of the proposed system based on evaluation results and identify future research questions.

II. Related Work

In graph data management, optimization of both *scalability* and *latency* simultaneously is a non-trivial task since improving one may hurt the other. In this section, we will identify the priorities of recent studies in this context and then analyze the approaches and procedures to optimize these problems by considering their objectives and data domains.

A. Static graph

To address the *latency* requirement, Khandelwal et al. [10] suggested a flat representation and additional operations (e.g., to build skip pointer lists and distribute the graph across servers) to index graphs by combining vertices and edges. This representation enables the compression of the input graph to be fitted into the main memory, and the retrieval engine can directly manipulate it. This work is reported to achieve state-of-the-art latency on graphs of moderate size. However, its performance on large graphs is not examined. Later Claude et al. [23] proposed to split the graph into primary and secondary memory. As a result, their system can be distributed and is scalable to graphs with 5B edges.

For dealing with the *scalability* requirement, Labouseur et al. [8], Malewicz et al. [3] proposed a distributed framework to support large-scale graph processing. Later, Mai et al. [20] and Zou et al. [21] proposed different frameworks based on GrapLab [4]. These frameworks, however, do not deal with the latency requirement. Other existing works address the scalability in the context of some specific applications. They are mostly designed to optimize particular operations on index data. For example, Kabiljo et al. [18] focused on partitioning the indexed data of large hyper-webgraphs without considering the real-time latency. Fang et al. [17] proposed to cluster graph for faster communities retrieval. In another different attempt, Kruse et al. [28] built a graph engine to support exploration and analysis on user networks. Since tailored to specific applications, these methods are inapplicable to our problem and data domain.

From the view of data priority, most of the work in web graph falls into one of the two categories during the indexing phase as the study of Roy et al. [5]: either **(1)** *Vertex-centric*: the graph is indexed by vertices to support vertices' and their adjacent vertices' retrieval, or **(2)** *Edge-centric*: the graph is indexed by links to supporting links target retrieval. A few works tried to combine several data such as vertices, edges, and their properties in indexing [6], [10] on a powerful single server thus cannot handle at such scale graphs as in our work.

Another approach in scaling up the graph management system is to improve internal operations during the indexing phase to reduce expensive computation or storage costs. For instance, Buehrer et al. [29] propose a compression method dedicated to graph indexing. Again, this study can significantly improve the compression rate, but ignorance of on-the-fly latency in its experiments.

B. Temporal graph

The latest studies in temporal graph developed by Nelson et al. [22] compressed full graph capture at a particular timestamp using 3-Dimension matrices to annotate the presence of a node in that capture. Though the work can have good compression ratios at small scales, the 3-D matrices can face an issue when the graph becomes gigantic and spans a very long duration. Moreover, this work can not deal with sparse 3-D matrices which hurt the compression ratio when the captures are collections of small subsets of the whole graph at a particular timestamp.

Among the most recent studies, there is the work of Ma et al. [11], which modeled the input temporal graph as a collection of snapshots at different timestamps and indexed the graph nodes based on their connectivity to other nodes. This work, however, did not handle sparse temporal graph and required an expensive preprocessing step for measuring nodes' connectivity, hence can not be scalable to the large graph.

Another system named Chronos [6] is handling temporal graphs through partitioning edges based on their *time-locality*. The study also aimed to support *scatter-gather* model algorithms. However, experiments on the very large graph were reported as unsuccessful tasks. In the same approach, ImmortalGraph [7] had multiple aspect analyses on *time* and *structure-locality*, and then applied into different datasets. However, on-the-fly latency was not the target of the author in this research.

The last one is from Moffitt et al. [30] to provide a similar solution to distribute a large web graph to many servers. The proposed solution, however, did also not focus on on-the-fly latency, thus can not be applied in our research context.

C. Web graph processing

There are very few previous works that are tailored for web graphs. Brisaboa et al. [24] first proposed a K^2 tree structure for a compressed representation of the graphs. This method, however, can handle various graphs up to moderate size. Najork [2] applied multiple techniques to achieve both *latency* and *scalability* and gained significant improvements on large graphs. This work can also well balance scalability and latency to achieve the state-of-the-art performance on 6B nodes graph. It motivates us to build enhancement.

In fact, web graphs have the unique characteristic that is not well explored. For example, each vertex is actual an URL with specific properties (e.g., string, web domain, path, and name of the page), which are not used by existing general graph approaches. It is worth notice that there are several effective works on the web graph domain, but none of them cover the time dimension of the web graph except the work from Boldi et al. [12], which just aimed to crawl a temporal web graph dataset for analysis.

In comparison to previous related works, we are the first to tackle *scalability* and *latency* problems for graph storage simultaneously and provide a solution for the temporal web graph retrieval problems that can handle gigantic graphs while maintaining state-of-the-art latency.

III. Scalable Hyperlink Store - SHS

In this section, we briefly describe the Scalable Hyperlink Store (SHS) [2], which is the base for our contribution. **Graph Data Structure**. SHS employs a *vertex-centric* data structure for web graphs. Specifically, a web graph in SHS is a directed graph with $G = \langle V, E\rangle$ where V is the set of URLs, each identifies a web page, and sorted by alphabet order on their domain name, and E is the set of (hyper) links between these pages. The system maintains a data store U that contains indices of all vertices $v \in V$ based on a hashing function on its domain name. In addition, for each vertex $v \in V$, the system maintains two stores of adjacency lists: out-link (also called *forward link*) and in-link (also called *backward link*) list. Each i-th element in each list is an identifier of a vertex having links from and to v, represented by d_i^v, respectively, as illustrated by example in Fig. 2. The web graph hence is represented as a triple of sets $G = \langle U, F, B\rangle$ where F and B are the set of vertices' forward and backward links, respectively.

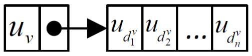


Fig. 2: Example of forward/backward link list: u_v is *id* of vertex v, $u_{d_i^v}$ is *id* of $i-$th vertex in the list.

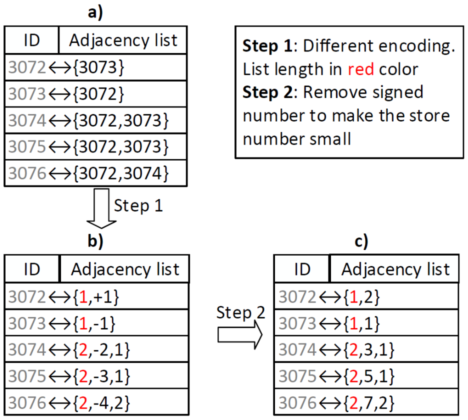
Graph Indexing Operations. These operations consist of partitioning, distributing at indexer client, then encoding, negative number removing, compacting, compressing, and storing on to persistent storage at each responsible server. For detail, the graph $G = \langle U, F, B\rangle$ is partitioned by the hash function that works on the vertices' index in U. Each partition is then distributed to a responsible server p. The value p and vertex's status(e.g. *normal* or *deleted*), together with index of v are embedded into an 64 bits number as a vertex's identifier u_v. Precisely, vertex v is assigned to server p as follows:

$$p = hash(v) \ mod \ N \tag{1}$$

In Eq. 1, N is the number of servers involved, and also the number of partitions. The p-th server is now responsible for the partition consisting of all vertices assigned to p and their forward and backward link list. We use $G_p = \langle U_p, F_p, B_p\rangle$ to denote the partition assigned to server p.

At each server p, each element d_i^v in forward and backward link list of vertex v are replaced by $u_{d_i^v}$. This list is then encoded by gap-encoding (delta-encoding), negative number (as result of gap-encoding operation) removing, then compacting by removing unused bits in number [13]. Finally, all three sets U_p, F_p, B_p are compressed by Variable Nybble Compression method [2], [13] and stored on disk. Fig. 3 illustrates these steps, in which figure a) shows mapped ($v \Rightarrow u_v$) adjacent list; figure b) shows gap-encoded of figure a); and figure c) show negative removed adjacent list of figure b).

Graph Retrieval Operations. Current SHS provides operations to retrieve adjacent list(s) of given vertices. Each vertex v in a query is assigned to its responsible server by recomputing partitioning function (Eq. 1) to retrieve the attached links. At the responsible server, all compressed indexed data of $G_p = \langle U_p, F_p, B_p\rangle$ are loaded into memory. It is worth notice that the loaded data may be very large in memory. Though data traversal on memory is fast, it still hurts the latency when the data size is too large. To reduce this cost, skip pointer lists [13] is built for all store cells U_p, F_p, and B_p with k elements in each block to skip inattention blocks during traversal phase.



a)

ID	Adjacency list
3072	↔{3073}
3073	↔{3072}
3074	↔{3072,3073}
3075	↔{3072,3073}
3076	↔{3072,3074}

Step 1: Different encoding. List length in red color
Step 2: Remove signed number to make the store number small

Step 1

b)

ID	Adjacency list
3072	↔{1,+1}
3073	↔{1,-1}
3074	↔{2,-2,1}
3075	↔{2,-3,1}
3076	↔{2,-4,2}

Step 2

c)

ID	Adjacency list
3072	↔{1,2}
3073	↔{1,1}
3074	↔{2,3,1}
3075	↔{2,5,1}
3076	↔{2,7,2}

Fig. 3: Operations on adjacent lists: a) Mapping u_v to URLs. b) Gap-encoding. c) Remove negative numbers.

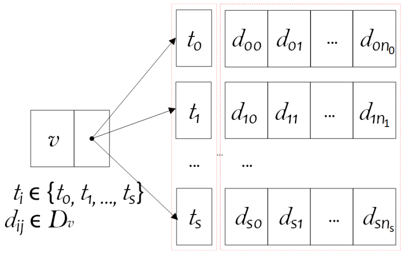

v
t_o
t_1
...
t_s
d_{00} d_{01} ... d_{0n_0}
d_{10} d_{11} ... d_{1n_1}
...
d_{s0} d_{s1} ... d_{sn_s}
$t_i \in \{t_o, t_1, ..., t_s\}$
$d_{ij} \in D_v$

Fig. 4: Captures of source vertex v and destination set at given captured timestamps.

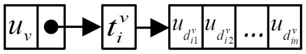


Fig. 5: A linked list from Fig. 2 enhanced with temporal information.

IV. Approach

We now describe our proposed extension for SHS [2] to handle temporal dimension of web graphs. The enhancement is basically done by embedding temporal dimension into the graph representation (Fig. 4) and enabling supporting operations. Our contribution is based on a novel bitwise-based data structure and operations, which were proved the efficiency in [34] to maintain the small latency of original SHS highly.

Specifically, we propose a novel temporal model, which employs a matrix to store the temporal evolution data. We explain how to implement and optimize it in SHS using the bitwise approach of the new model. The new temporal model has two implementation data structures capable of handling vertices at different density levels of link degree. We discovered that the majority of the processes in the temporal dimension of web graphs are the two fundamental operations. Hence we implement these two operations using bitwise-based calculations as the basic functionality of our system. To this end, we modify some individual workflow tasks in the indexing operations that drastically reduce the operation time compared to the original workflow.

Our solution elaborates on temporal dimension while keeping original high optimizations in both *scalability* and *latency* as the most crucial factors for efficient information retrieval in large and evolving graphs. Based on our extension, the system can rapidly extract adjacent nodes that appeared within two timestamps in large graphs to execute many fundamental graph algorithms (e.g., HITS and SALSA) efficiently at that scale while highly maintaining state-of-the-art latency from original work.

A. Temporal model as bit matrix

There are different types of temporal web graphs that need to be considered separately, depending on the initial crawling strategy. Some graphs are tracked and updated by the link evolution, while others are collections of captures of the whole graph at different timestamps. All aforementioned crawling strategies have drawbacks when crawling large-scale graphs due to delays in processing and thus the inability to capture the whole graph at a particular timestamp. Instead, typically the crawling of smaller sub-graphs at different timestamps is applied. Each vertex in such a temporal graph is a URL of a web page with a timestamp, called a capture. This type of graphs is also our primary concern dataset in this work. In this subsection, we aim to handle the temporal data of the vertices, which contain a few changes in all snapshots of different timestamps.

Formally, a temporal web graph is a directed graph $G^{\mathcal{T}} = \langle V, E, T \rangle$ where $V = \{v_0, v_1, ..., v_{|V|-1}\}$ is the set of URLs, and have corresponding identifier $v_i = i$; T is the set of all captured timestamps of the pages and $E \subseteq (\ V \times T \times V\ )$ is the set of links to or from the captured vertices. For each vertex v in the graph, we use the set $T_v = \{t_0, t_1, ..., t_{|T_v|-1}\}$ to denote the timestamps when v is captured. Fig. 4 also illustrates all captures of vertex v with $s = |T_v| - 1$, and out-link list of each capture, while Fig. 5 represents a capture at a given timestamps t_i of vertex v, which has out-link vertices set $D_{v^i} = \{x | \langle v, t, x \rangle \in E, t = t_i\}$. Therefore the out-link set of vertex v over time is $D_v = \bigcup D_{v^i}$. All related data of a vertex v can be represented as an Eq. 2 in which M_v is a similar representation as a presence matrix [22] or a property matrix [31] of the only node v in binary-value form M_v as in Eq. 3. We then call M_v as bit matrix.

$$a_v^{\mathcal{T}} = \langle v, T_v, D_v, M_v \rangle \tag{2}$$

$$M_v = \begin{bmatrix} e_{0,0} & e_{0,1} & \cdots & e_{0,n_0} \\ e_{1,0} & e_{1,1} & \cdots & e_{1,n_1} \\ \cdots & & & \\ e_{|T_v|-1,0} & e_{|T_v|-1,1} & \cdots & e_{|T_v|-1,|D_v|-1} \end{bmatrix} \tag{3}$$

In which $e_{i,j}$ is indicated by:

$$e_{i,j} = \begin{cases} 1 \text{ if } \langle v, t_i, d_v^j \rangle \in E \\ 0 \text{ if } \langle v, t_i, d_v^j \rangle \notin E \end{cases} \tag{4}$$

Analogously, the temporal web graph model can be represented as follows

$$G^{\mathcal{T}} = \langle U, F^{\mathcal{T}}, B^{\mathcal{T}} \rangle \tag{5}$$

where U are still URL set, and $F^{\mathcal{T}}$, $B^{\mathcal{T}}$ are forward and backward link stores of temporal adjacent lists, respectively.

Consider an example with a given vertex v having 3 captures at three timestamps $T_v = \{t_0, t_1, t_2\}$. These captures contain a set of out-links vertex $D_v = \{d_0, d_1, ...d_9\}$. Each capture contains a subset of $D_t \subseteq D_v$, with $t \in T_v$. Fig. 6 illustrates the matrix M_v of this example. The presence of vertex $d_v \in D_v$ of any given capture is indicated by bit 1.

Fig. 6 shows that the scarce of change in temporal snapshots of a vertex can lead to a high-density bit matrix M_v because most of the different rows in bit matrix can contain the same number of value 1 at the same columns.

B. Temporal model as adjacent pairs

Although the link evolution of a vertex is small in most web domains, some of them, such as news or blogs site, show the opposite and quite volatile behavior. The large difference values make the matrix M_v sparse. We, therefore, employ a specific sparse matrix representation for such cases. To calculate the dense ratio of matrix M_v, given $E_v = \{e_{i,j}\}$ is the set of elements in M_v, we calculate the ratio of the number of existent links over the number of elements in matrix M_v:

$$p_v = \frac{\{e_{i,j} \in E_v | e_{i,j} = 1\}}{|E_v|} \tag{6}$$

Our analysis on a temporal web graph extracted from Internet Archive [15], [32], [35] shows that it contains around 94% of domain having matrix density $p_v > 80\%$. On the other hand, only 0.6% of vertices have density $p_v < 40\%$. We discover that most of these vertices are very well-known German News, e.g. "http://bild.de/", "http://spiegel.de/", where the contents as well as the incoming and outgoing links are changing rapidly during their lifetime.

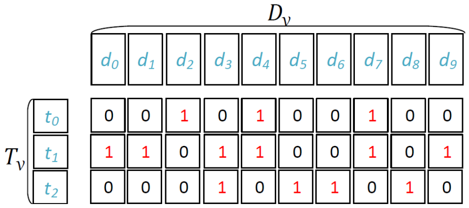
Our experiments show that the size of array A_{M_v} can make the latency longer while querying these domains. Therefore, we set the $threshold = 40\%$. In case the value of $p_v < threshold$, we employ a set of adjacent pairs i, j such that $M_v = \{\langle i, j\rangle | e_{i,j} = 1\}$ to indicate the existence of the corresponding links at the interested timestamp. The efficiency of this implementation will be analyzed in Sec. VI.

C. Temporal web graph store data structure

It was evident that stores compressed two dimensions array can hurt decompression operations, thus also nfluences latency. We solve this issue by flattening the matrix into 1-D dimension. Since the adjacent pairs are flatted, we just flatten M_v by concatenating the rows into a bit string. The length of the bit string is left padded to be multiples of 32. Then the bit string is splitted into multiple 32-bit elements to obtain a 1-D 32-bit number array representation by A_{M_v}. We then replace M_v by A_{M_v} in Eq. 2. Finally, we obtain a data structure of $a_v^{\mathcal{T}}$ as a one-dimension list of numbers of Eq. 7.

$$a_v^{\mathcal{T}} = \langle v, T_v, D_v, A_{M_v}\rangle \tag{7}$$

Fig. 7 shows adjacent elements of the example in Fig. 1 with matrix M_v flattened and embedded in each element as illustrated in Fig. 8. In Fig. 9 step 1) also shows one adjacent list in link store, in which M_v is flattened to a bit string. For example bit matrix M_v in Fig. 8, is implemented by a 32-bit elements, thus returns an 1-element array $A_{M_v} = \{401\}$ as the last row of Fig. 9 step 2) result.



		D_v									
		d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
T_v	t_0	0	0	1	0	1	0	0	1	0	0
	t_1	1	1	0	1	1	0	0	1	0	1
	t_2	0	0	0	1	0	1	1	0	1	0

Fig. 6: Example data in Eq. 3. It is depicting a node with 3 captures and 10 total links.

D. Skip pointer list

The SHS's skip pointer list also needs to be enhanced to adapt to the new graph data structure. Each pointer points to the starting position of a compressed block of k continuously temporal elements. As a result, the traversal and decompression will only happen within a block, hence can save much computer instructions.

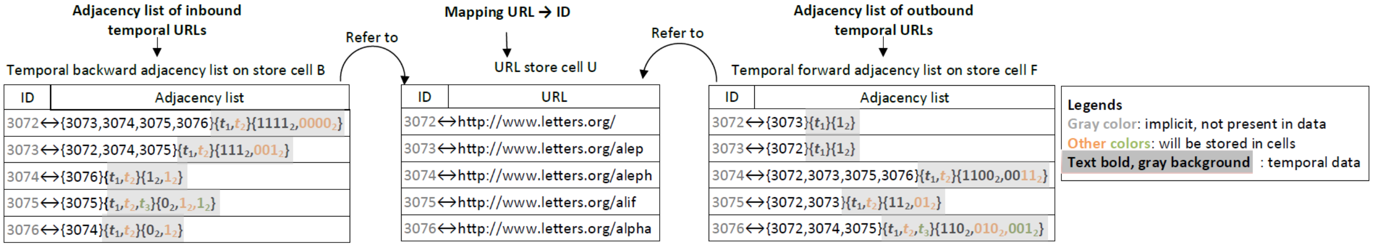
As in the previous section, each element $a_v^{\mathcal{T}}$ (Eq. 7) will be completely flattened and represented by a numeral array. However, the array's size can be very large and would not be optimally compressed. We then also employ gap-encoding to make these numbers smaller. Moreover, it is worth notice that negative decimal numbers actually utilize all bits of the defined bit length in computer memory by using 2's complement number method, thus also hurts the compression. We then repeat the steps in SHS to remove negative decimal numbers, but with a number array A_{M_v}. In Fig. 9 step 1) shows how to apply gap-encoding method to minimize the numbers in the temporally adjacent lists, and step 2) shows how to remove the negative numbers to get small values in 2's complement binary numbers.

The drawback of gap-encoding is in cases where an element at the later part of the list has to be accessed. In this case, we have to traverse the list from the beginning to the necessary position. In the context of large data, applying gap-encoding for whole data can have reversed effects and slow down query execution times.

To address this problem, we enhanced the skip pointer list in SHS to our new data structure with k elements of $a_v^{\mathcal{T}}$. Fig. 10 shows an example encoded data with block size $k = 4$. The arrays represented for elements $a_v^{\mathcal{T}}$ in a block then merge as a bigger number array, which is the best for compression. The value of k is very sensitive since if k is too small, the skip pointer list may be very large, whilst if k is too large, the retrieval computation will be costly. The value for k is set to 32 as Najork suggested in [2].

E. Temporal neighbors retrieval model

There are several operations proposed in [33], [35] to address the interest research on temporal graphs. We study


Adjacency list of inbound temporal URLs
Refer to
Mapping URL → ID
Refer to
Adjacency list of outbound temporal URLs

Temporal backward adjacency list on store cell B

ID	Adjacency list
$3072\leftrightarrow\{3073,3074,3075,3076\}\{t_1,t_2\}\{1111_2,0000_2\}$	
$3073\leftrightarrow\{3072,3074,3075\}\{t_1,t_2\}\{111_2,001_2\}$	
$3074\leftrightarrow\{3076\}\{t_1,t_2\}\{1_2,1_2\}$	
$3075\leftrightarrow\{3075\}\{t_1,t_2,t_3\}\{0_2,1_2,1_2\}$	
$3076\leftrightarrow\{3074\}\{t_1,t_2\}\{0_2,1_2\}$	

URL store cell U

ID	URL
3072↔http://www.letters.org/	
3073↔http://www.letters.org/alep	
3074↔http://www.letters.org/aleph	
3075↔http://www.letters.org/alif	
3076↔http://www.letters.org/alpha	

Temporal forward adjacency list on store cell F

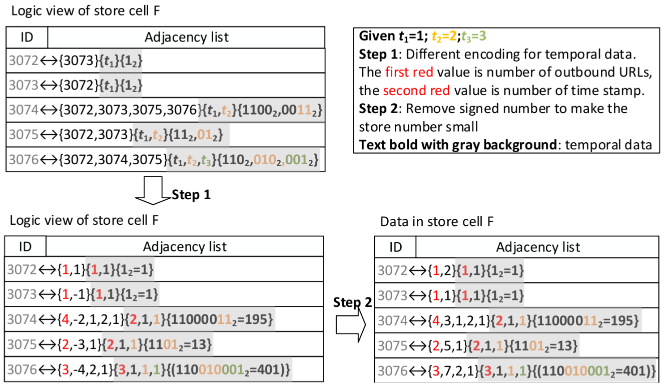
ID	Adjacency list
$3072\leftrightarrow\{3073\}\{t_1\}\{1_2\}$	
$3073\leftrightarrow\{3072\}\{t_1\}\{1_2\}$	
$3074\leftrightarrow\{3072,3073,3075,3076\}\{t_1,t_2\}\{1100_2,0011_2\}$	
$3075\leftrightarrow\{3072,3073\}\{t_1,t_2\}\{11_2,01_2\}$	
$3076\leftrightarrow\{3072,3074,3075\}\{t_1,t_2,t_3\}\{110_2,010_2,001_2\}$	

Legends
Gray color: implicit, not present in data
Other colors: will be stored in cells
Text bold, gray background : temporal data

Fig. 7: A logical view of the of temporal graph store cells.

$$\underbrace{3076}_{v}\leftrightarrow\underbrace{\{3072,3074,3075\}}_{D_v}\underbrace{\{t_1,t_2,t_3\}}_{T_v}\underbrace{\{110_2,010_2,001_2\}}_{M_v}$$

Fig. 8: A data structure representation of Eq. 7 in forward store cell from Fig. 7.



Logic view of store cell F

ID	Adjacency list
$3072\leftrightarrow\{3073\}\{t_1\}\{1_2\}$	
$3073\leftrightarrow\{3072\}\{t_1\}\{1_2\}$	
$3074\leftrightarrow\{3072,3073,3075,3076\}\{t_1,t_2\}\{1100_2,0011_2\}$	
$3075\leftrightarrow\{3072,3073\}\{t_1,t_2\}\{11_2,01_2\}$	
$3076\leftrightarrow\{3072,3074,3075\}\{t_1,t_2,t_3\}\{110_2,010_2,001_2\}$	

Given t_1=1; t_2=2;t_3=3
Step 1: Different encoding for temporal data. The first red value is number of outbound URLs, the second red value is number of time stamp.
Step 2: Remove signed number to make the store number small
Text bold with gray background: temporal data

Logic view of store cell F

ID	Adjacency list
$3072\leftrightarrow\{1,1\}\{1,1\}\{1_2=1\}$	
$3073\leftrightarrow\{1,-1\}\{1,1\}\{1_2=1\}$	
$3074\leftrightarrow\{4,-2,1,2,1\}\{2,1,1\}\{11000011_2=195\}$	
$3075\leftrightarrow\{2,-3,1\}\{2,1,1\}\{1101_2=13\}$	
$3076\leftrightarrow\{3,-4,2,1\}\{3,1,1,1\}\{(110010001_2=401)\}$	

Data in store cell F

ID	Adjacency list
$3072\leftrightarrow\{1,2\}\{1,1\}\{1_2=1\}$	
$3073\leftrightarrow\{1,1\}\{1,1\}\{1_2=1\}$	
$3074\leftrightarrow\{4,3,1,2,1\}\{2,1,1\}\{11000011_2=195\}$	
$3075\leftrightarrow\{2,5,1\}\{2,1,1\}\{1101_2=13\}$	
$3076\leftrightarrow\{3,7,2,1\}\{3,1,1,1\}\{(110010001_2=401)\}$	

Fig. 9: Example of temporal enhancement of Fig. 3.

these kinds of operations and found that they constitute from two fundamental query types:

Merge query to request a mergence of connected (backward/forward) vertices appeared within the a given time interval $[t_a, t_b]$ from all captures of a given vertex v.

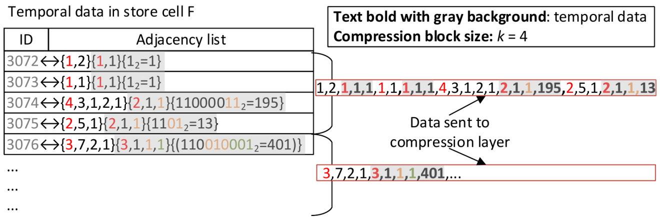
$$I^{Dir}_{v[t_a,t_b]} = \left\{x \in D_v | t \in T_{v[t_a,t_b]} \wedge M_v\left[t,x\right] = 1\right\} \tag{8}$$

Temporal snapshot query to request the connected (backward/forward) vertices of the latest capture of a given vertex v within the given time interval $[t_a, t_b]$.

$$S^{Dir}_{v[t_a,t_b]} = \left\{ I^{Dir}_{v[t_i,t_i]} | t_i \in T_{v[t_a,t_b]} \wedge t_i = max\left(T_{[t_a,t_b]} \setminus \{t_b\}\right)\right\} \tag{9}$$

Using these two queries above, we can build any composition variations to support the most complicated demands. For example, to request the vertices appear within the given time interval, a combination of above queries is depicted in Eq. 10. It is worth notice that $Dir \in \{Forward, Backward\}$ is the direction of request, corresponding with $\{F^{\mathcal{T}}, B^{\mathcal{T}}\}$, which identifies the link store that the query aims to seek in.

It is worth noting that the queries above consider the ID of vertices in query content. Therefore, one supporting function to find the mapping from vertex to ID and vice versa is implemented. This function is also important and affects



Temporal data in store cell F

ID	Adjacency list
$3072\leftrightarrow\{1,2\}\{1,1\}\{1_2=1\}$	
$3073\leftrightarrow\{1,1\}\{1,1\}\{1_2=1\}$	
$3074\leftrightarrow\{4,3,1,2,1\}\{2,1,1\}\{11000011_2=195\}$	
$3075\leftrightarrow\{2,5,1\}\{2,1,1\}\{1101_2=13\}$	
$3076\leftrightarrow\{3,7,2,1\}\{3,1,1,1\}\{(110010001_2=401)\}$	
...	
...	
...	

Text bold with gray background: temporal data
Compression block size: k = 4

1,2,1,1,1,1,1,1,1,1,1,4,3,1,2,1,2,1,1,195,2,5,1,2,1,1,13

3,7,2,1,3,1,1,1,401,...

Fig. 10: Temporal store cells blocks with $k = 4$.

latency. As a consequence, we also evaluate this function latency to verify our proposed data structure.

$$E^{Dir}_{v[t_a,t_b]} = I^{Dir}_{v[t_a,t_b]} \setminus \left\{ I^{Dir}_{v[t_0,t_a]}\right\} \tag{10}$$

F. Implementation

Architecture. We reuse the architecture built for SHS, and enhance the internal implementation in each server. The enhancement consists of partitioning method, data structure, and network communication. In detail, we implement our extension in two processes:

Indexing process. We also improve the indexing steps. More detail, instead of partitioning on-the-fly as SHS, we partition the graph on Hadoop and then simultaneously send the partitioned datasets to corresponding responsible server. Experiments show that this helps to reduces the indexing time duration of a large graph from 14 days down to 23 hours, which is about 15 times faster, in comparison with the original partition on-the-fly workflow of SHS.

Retrieval process. The retrieval process is executing queries executions as described in section IV-E with temporal data structure. In both basic queries (Sec. IV-E), for a given vertex v, time period $[t_a, t_b]$, the direction of the request is also provided.

We also extend more than 50 network communication protocol messages among the servers in both indexing and retrieving processes to make them to be capable to transport temporal data, e.g. extend current protocol to carry the pair of an edge $\langle v_{src}, d_{dest}\rangle$ to carry the triple $\langle v_{src}, t_i, d_{dest}\rangle$.

V. Complexity Analysis

In this section, we analyze the complexity of employed algorithms mentioned in section IV-E with a special focus on the retrieval as a *high-performance* feature of the system. Given a temporal query as format: $\langle v, t_a, t_b\rangle$, there are two main operations to compute the result. Tab. I shows these operations and their corresponding proportional times required for processing.

TABLE I: Operations complexity at partition p

Operations	**Complexity**
Seeking ID of a URL	$\mathcal{O}\left(\log n_p + C\right)$
Seeking URL from an ID	$\mathcal{O}\left(\log n_p + C\right)$
Retrieve an adjacent list of v	$\mathcal{O}\left(\log n_p + \sum\limits_{v=b}^{b+K} \lvert a_v^{\mathcal{T}} \rvert + \lvert M_v \rvert\right)$

1) Find mapping URL to ID and vice versa. Given URL store cell $U_p = \left\{u_{v \in V_p}\right\} = \{0, 1, 2, ...n_p\}$ where all vertices $v \in V_p$ belong to the partition p, and $k = K$ is the block size in storage cells, there are two fundamental operations to exchange ID to URL and vice versa. Both operation complexities match $\mathcal{O}\left(\log n_p + C\right)$ as the original operation in SHS.

2) Retrieve adjacent list from temporal link store cells. Similarly, this operation needs to examine the skip pointer list to find the block pointer of interest, then examine the block to find the requested adjacent list. In the worse case, the algorithm has to search from beginning to the end of the interested block and read all data of the adjacent list found; thus proportional time duration needed to process the operation matches $\mathcal{O}\left(\log n_p + \sum\limits_{v=b}^{b+K} \lvert a_v^{\mathcal{T}} \rvert + \lvert M_v \rvert\right)$.

VI. Experimental Evaluation

A. Evaluation method

To our best knowledge, there has not been other work on *temporal web graph* that simultaneously deals with both *scalability* and *latency*. Therefore, we implement two systems based on well-known frameworks and use them as baselines to evaluate our system. In particular, we build and then index 4 diversified graphs on our system at different scales on both data size and system architecture. The resource consumption in different phases is tracked to evaluate system performances. We then evaluate the systems on the following aspects:

- **Functionality evaluation**. We use tracked data of the indexing process and measure the latency by requesting many different query types over these diversified datasets to prove: **1)** *Scalability* of our system. **2)** The two implementations of the matrix densities support very small *latency*. **3)** The effectiveness of the new link store cell's data structure is independent of temporal data extension. **4)** Finally, we show that our system's latency outperforms the implemented baselines on requesting different batch queries' sizes from $1000 - 5000$ vertices(URLs) over selected indexed datasets. From our perspective, these activities are fair because the data size and queries are in the scope of the baselines.
- **Practicability Evaluation**. In this evaluation, we first employ naive temporal HITS and SALSA [26], [27] to simulate intensive application usage. Then for the practical application experiment, we install the system on our organization cluster for daily research projects use in 6 months and collect user feedback for evaluation.

TABLE II: Detailed properties of graph datasets

	Wiki	**Bing**	**Small_GA**	**German**
#Nodes	2,166,669	2,098,796	1,276,554,018	43,061,274,770
#Links	86,337,879	486,259,835	9,440,581,074	317,783,143,013
Duration	2002-2011	1998-2014	1-3/2013	1998-2014
Length	10 years	16 years	3 months	16 years
#Captures	1 - 3240	2	1 - 91	1 - 5840
MaxDegree	394,371	2	2,492,801	66,428,064
Size(GB)	3.7	38.7	1070	21200

B. Dataset

It is worth noticing that the link degree and number of the snapshot of the graph can significantly affect the retrieval activities. Therefore, we intentionally choose 4 diversified datasets with different characteristics, as detailed in Tab. II. We then used them to evaluate the functionality, including the efficiency and usability of our approach over a wide range of characteristics of datasets. The details of each dataset are as follows:

Temporal German Wikipedia Graph. This is a graph evolution of (hyper)links among German language pages of Wikipedia [†] within 10 years. It is small and low average link degree. For short, we call it $Wiki$.

Bing Result in German Archive Graph. This is a graph evolution of (hyper)links among German domain pages (*.de) within 16 years, which is a 1-hop extension of Bing search results of 1.7 million Wikipedia entities. We intentionally process the extended graph to make its maximum link-degree of each vertex at 2, which means consisting of 2 captures at the starting and the ending timestamps of each (hyper)link. Its size is moderate. We name it $Bing$.

Internet Archive. As the biggest dataset collection about the past, which also embeds the biggest temporal web graph among almost web pages within 16 years from 1996-2014. The evolution of (hyper)links, which connect among the captures over time, is extracted to form the temporal web graph. In our work, we use a subset in the Internet Archive collection, filtered from "*.de" domains and named German Archive. From German Archive, we extract two different graphs: a short duration graph within **1-Jan-2013** and **31-Mar-2013**, named $Small\_GA$, at large size; and a full duration graph, named $German$, at gigantic size; to compare the latency with baselines and evaluate the practicability.

C. Baselines

1) Leveraging SHS [2] and time filtering with MySQL. In the absence of appropriate baselines, we exploit a static web graph store and an external temporal data management system to build a temporal web graph management system. In this experiment, we leverage SHS as a state-of-the-art static web graph store in our context and MySQL as an external

[†] http://konect.uni-koblenz.de/networks/link-dynamic-dewiki

database system to store corresponding temporal data. We choose the dataset $Small\_GA$ to index in this baseline due to MySQL limitation on scale dataset. The indexed dataset for this experiment is processed in 3 steps: **1)** convert the dataset $Small\_GA$ into a static web graph by merging all links in revisions into one revision. **2)** Index that static graph obtains in step 1 with SHS. **3)** We store the original temporal data into the MySQL database management system using URL mapping results from SHS. To process results from a time concerned query, we retrieve all links from SHS and then use timestamps information to filter out the irrelevant results respecting the time constraints in the query by requesting time information from MySQL. Since MySQL is an external system, in this experiment, we do not take into account resource consumption measurement except latency.

2) Employ HBase. In this baseline, we make use an HBase system to store triple $\langle v, d_v, t_{v[i]} \rangle$ and index data by source v and timestamps d_v to support similar retrieval operations. HBase is installed on the same 29 multi-tasking shared servers together with our system. For this baseline, we also measure only the latency of the operations.

D. Experiment setup

We installed our system on different numbers of multi-tasking shared servers depended on each experiment. Each server holds 2 CPUs of various models from Intel(R) Xeon(R) E5-2620 (v2 - v3 series), which are from $12-24$ cores having $128-256$GB RAM installed. The network communication among the servers has high bandwidth at 58 Gbps. We also develop a tracking system to record execution data during the experiment phases.

To evaluate the system's *functionality*, we implement and measures indexing and retrieving operations using graph $Wiki$ and $Bing$ datasets onto $1, 2, 4, 8, 16$, and 29(limited by infrastructure). For indexing operations, we also experiment on these two graphs to evaluate the stability at different numbers of servers. For the retrieval operations, we randomly select either one operation in section IV-E for each query.

To experiment on *practicability*, we index the $German$ graph on 29 servers and evaluate the index performance through resource consumption. The system then acts like a graph search engine for naive temporal HITS and SALSA to evaluate system performance and to measure the responsibility of the system. Please note that we leverage the temporal version of these algorithms by naive approach, in which we simply apply **Merge** operation (section IV-E) to discard temporal information in given duration $[t_1, t_2]$ as input of HITS or SALSA algorithm. It is worth notice that since this dataset is gigantic, using it to do *functionality* experiments on a small number of servers is impossible.

E. Functional experiment results

Scalability. Fig. 11, chart a) and b) show time duration and the resource consumption respectively of the indexing phase using $Wiki$ and $Bing$ datasets on different numbers of servers. It is worth noting that $Small\_GA$ and $German$ datasets can not be indexed on a small number of servers such as $1, 2$, and 4 servers due to their huge size, thus we dismiss them in this experiment. RAM and disk usage are tracked at two phases: **(1)** mapping URL to ID and **(2)** the rest operations. The figures clearly show that our system is scalable since *a)* increment of the number of the servers may reduce indexing time but at diminishing returns due to the trade-off between computational and network communication time. *b)* Consumption slightly increased for both RAM and storage. Compared with resource consumption during indexing on 2 servers we found that this increment is negligible. Tab. III also shows further indexing data of 4 datasets on 29 servers and also proves the *scalability* due to moderate resource consumption while keeping a high compression rate.

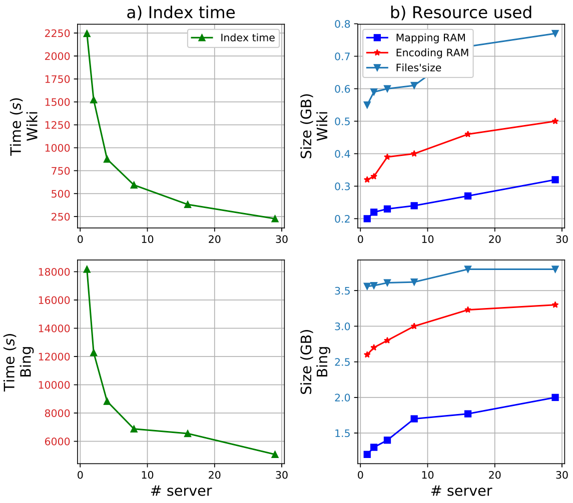


Fig. 11: Resource consumption to index $Wiki$ and $Bing$ graphs on different numbers of server.

TABLE III: Indexing statistics on different graphs

	Wiki	**Bing**	**Small_GA**	**German**
Original size (GB)	3.8	38.7	1070	21200
Indexed size (GB)	0.572	5.7	136	3781
% of original	15	14.7	12.8	17.8
Index time (s)	227	6381	52873	1351094

Latency. Because the data stores $F^{\mathcal{T}}$ and $B^{\mathcal{T}}$ have structural similarity (Eq. 7), we evaluate the latency on **Merge** and **Temporal Snapshot** (Sec. IV-E) requests. The performance is assessed in three concerning purposes:

- *Baselines comparison*. We define latency as Eq. 11:

$$latency = \frac{duration}{\#links} = \frac{1}{throughput} \tag{11}$$

to measure the average time to retrieve a link, which is the inverse of *throughput* [36]. The latency of our system is compared with two baselines using the exact query requests on each system, which is handling the indexed data of $Small\_GA$ dataset. Since $German$ is too large and the others are too small to run on the baselines, $Small\_GA$ is the best. Fig. 12 a) shows the latency on dataset $Small\_GA$ of identifier-to-identifier (UID-2-UID) retrieval operation, which does not include time to map an URL to a number and vice versa. Fig. 12 b) shows the string-to-string experiment results in which the input and output are URLs.

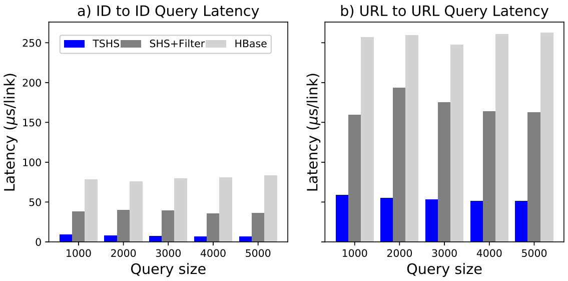


Fig. 12: Latency of different batch query sizes using the indexed dataset $Small\_GA$.

- *Matrix density independence*. We present that the matrix density does not affect both *latency* and *scalability* by using our approach. For details, we choose to experiment with a particular query batch Q (Eq. 12) to show the stability of the latency, regardless of the temporal matrix densities (Sec. IV-C). Query batch Q is built by selecting URLs from Q_d (with $d = \{0, 0.1, ..., 0.9\}$) with more than 30M captures.

$$Q_d = \{c | d \leq p_c \leq d + 0.1\} \qquad (12)$$

Fig. 13 shows that the latency of our system depends less on matrix density than on the total number of links in the result. The independence is explainable since in the case the result has a large number of links, its size is proportionally large. Moreover, sending a large size of data at once may reduce network overhead among servers, which also leads to reducing the latency.

- *Temporal data independence of enhanced data structure on $F^{\mathcal{T}}$ and $B^{\mathcal{T}}$*. We prove that the latency of our approach only depends on the number of servers and number of out-links, while the mapping location of an URL in huge indexed data does not affect the latency. In detail, we use two cases of two URLs: **1)**. Different locations in the indexed map but have approximate numbers of out-links. **2)**. Closed locations in the indexed map but have different numbers of out-links. Fig. 15 a) shows that two vertices at two ends of the mapping table do not affect latency, while b) shows the proportional execution time with the number of out-links.

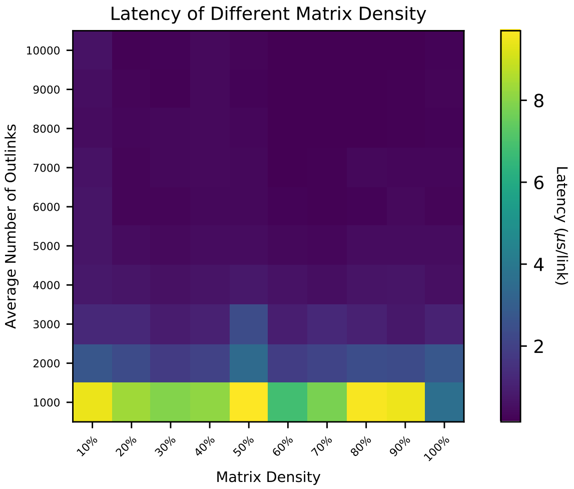


Fig. 13: Latency of system using index from dataset $German$ to respond batch queries Q.

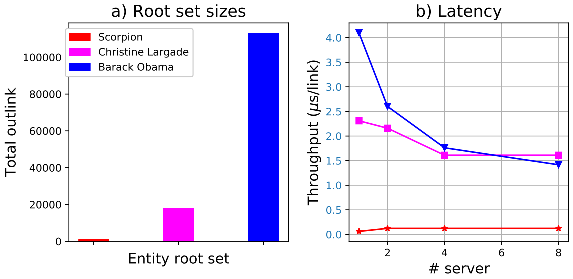


Fig. 14: Correlation between root set sizes acquired by entity search from $Bing$ as query and throughput of the system using the index of dataset G.

Algorithm 1 Retrieve base set subgraph within given duration from a root set

```
function GET-BASE-SET($R, t_a, t_b$)
    R is root set of nodes, $t_a$ and $t_b$ are timestamps
    Given temporal graph store $G^{\mathcal{T}} = \langle U, F^{\mathcal{T}}, B^{\mathcal{T}} \rangle$
    for all $v \in R$ do
        $B \leftarrow B \cup \left\{ \langle v, x \rangle | \forall x \in I^{F^{\mathcal{T}}}_{v[t_a, t_b]} \right\}$
        $B \leftarrow B \cup \left\{ \langle x, v \rangle | \forall x \in I^{B^{\mathcal{T}}}_{v[t_a, t_b]} \right\}$
    for all $v \in nodes\,(B)$ do
        $B \leftarrow B \cup \{ \langle v, x \rangle | \forall x \in nodes\,(B) \wedge x \neq n \}$
    return $B$
```

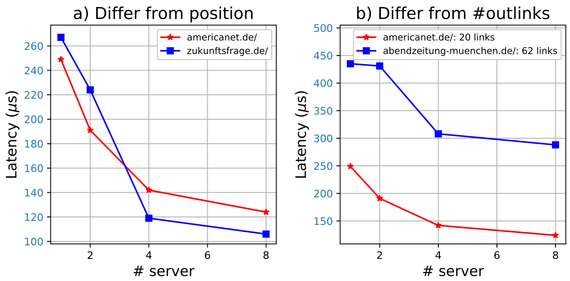


Fig. 15: Effectiveness of mapping locations and result size: a) Different locations but approximate numbers of out-links. b) Approximate locations and the approximate number of out-links.

Algorithm 2 HITS on Temporal web graph

```
function HITS-SCORES($R, t_a, t_b$)
    R is root set of nodes, $t_a$ and $t_b$ are timestamps
    $B \leftarrow$ Get-Base-Set $(R, t_a, t_b)$
    for all $v \in nodes\,(B)$ do
        $H\,[v] \leftarrow \frac{1}{|nodes(B)|}$
        $A\,[v] \leftarrow \frac{1}{|nodes(B)|}$
    repeat
        for all $a \in nodes\,(B)$ do
            $A'\,(a) \leftarrow \sum_{(x,a) \in B} H\,(x)$
        for all $h \in nodes\,(B)$ do
            $H'\,(h) \leftarrow \sum_{(h,x) \in B} A\,(x)$
        for all $v \in nodes\,(B)$ do
            $H\,(v) \leftarrow \frac{1}{\|H'\|_2} H'\,(v)$
            $A\,(v) \leftarrow \frac{1}{\|A'\|_2} A'\,(v)$
    until $H$ and $A$ converge return $\{aut, hub\}$
```

Algorithm 3 SALSA Authority on Temporal web graph

```
function SALSA-AUTHORITY-SCORES($R, t_a, t_b$)
    R is root set of nodes, $t_a$ and $t_b$ are timestamps
    $B \leftarrow$ Get-Base-Set $(R, t_a, t_b)$
    $B^A \leftarrow$ set nodes $\in B$ which have in-links
    for all $v \in nodes\,(B)$ do
        $H\,[v] = \begin{cases} \frac{1}{|B^A|} \text{ if } v \in B^A \\ 0 \text{ otherwise} \end{cases}$
    repeat
        for all $a \in B^A$ do
            $A'\,(a) = \sum\limits_{(x,a) \in B} \sum\limits_{(x,w) \in B} \frac{A(w)}{out(x)in(w)}$
        for all $v \in B^A$ do
            $A\,(v) = A'\,(v)$
    until $A$ converge
    return $A$
```

F. Practicality experiment

Intensive simulation context. We use **1) Merge** queries (Sec. IV-E) to merge all revisions of a temporal subgraph of graph $German$ within duration $[t_a, t_b]$ into a static subgraph. The queries are 15 entity search results of the popular people, locations, and organizations on live Bing. **2)** These search results are then used as root sets of HITS and SALSA, varying from 201 - 311 URLs(vertices). **3)** We track the execution time of 50 iterations of both algorithms over the indexed $German$ dataset. We then repeat this experiment with SHS over the static version of $German$ dataset. It is worth notice that when using SHS, step **1)** is unnecessary. Fig. 16 shows two performances of SHS and our approach. The data show that the overhead for time dimension processing in our approach is slightly higher than that of SHS, but acceptable.

TABLE IV: Detailed statistics of the system using dataset G during the processing of temporal HITS and SALSA

Property	Values
Files' size (GB)	95 - 250
Memory consumption (GB)	34 - 105
Get-Base-Set duration (ms)	532 - 1530
HITS duration (ms)	130 - 407
SALSA-Authority-Score duration (ms)	92 – 430

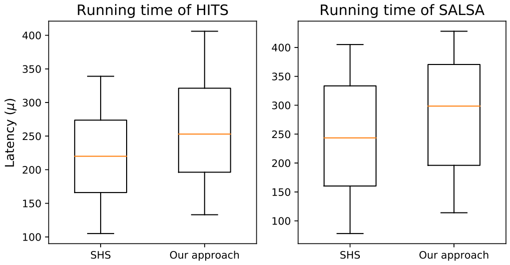


Fig. 16: Comparison of the results of our approach and SHS over dataset G to run HITS and SALSA.

The root sets are then sent to the Alg. 2 and Alg. 3 as input root sets. Fig. 14 shows the performance results of these algorithms using the input data, whilst Tab. IV shows the overall statistics of the experiments on dataset G. Obviously, the latency of a small result set slightly increases when the index data spans on more servers. However, it shows the contrary for the latency of the large result. This behavior is explainable that with the small result, the network communication can be overhead for throughput of each individual link in the result.

VII. Conclusion and Discussion

In this paper, we presented a novel approach to efficiently manage *temporal web graphs* in terms of *scalability* and *latency*. The measured latency of the query retrieval in our solution varies from a few to several hundred microseconds, which satisfy the most critical requirement for many on-the-fly temporal web graph algorithms. Our experiments showed that the latency of the retrieving phase outperforms the baselines by speeding up at least 5 times. Moreover, our enhancement on the SHS indexing workflow can significantly reduce the time cost for indexing compared to that of SHS.

The system has few technical limitations, which could be good motivation for future improvement. We currently handle a maximum of up to 32 servers, while there is room to expand the system to 128 servers. On the other hand, the partitioning process is currently running on Hadoop, thus making the system more complicated. Parallelizing and integrating the partitioning method with its client are also a possible future task.

References

[1] P. Boldi and S. Vigna. 2004. TheWebgraph Framework I: Compression Techniques. In Proc. 13th WWW (WWW '04). ACM, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[2] Marc Najork. 2009. The Scalable Hyperlink Store. In Proc. 20th HT (HT '09). ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/1557914.1557933

[3] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In Proc. 2010 ACM SIGMOD (SIGMOD '10). ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[4] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5, 8 (2012), 716–727. doi:10.14778/2212351.2212354

[5] A. Roy, I. Mihailovic, and W. Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In Proc. 24th SOSP (SOSP '13). ACM, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[6] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In Proc. 9th EuroSys (EuroSys '14). ACM, New York, NY, USA, Article 2, 14 pages. https://doi.org/10.1145/2592798.2592799.

[7] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. ACM Trans. Storage 11, 3, Article 14 (July 2015), 34 pages. DOI:https://doi.org/10.1145/2700302

[8] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, et al., "The g* graph database: Efficiently managing large distributed dynamic graphs", Distrib. Parallel Databases, vol. 33, no. 4, pp. 479-514, December 2015.

[9] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos. 2015. TimeCrunch: Interpretable Dynamic Graph Summarization. In Proc. 21th ACM SIGKDD (KDD '15). ACM, New York, NY, USA, 1055–1064. https://doi.org/10.1145/2783258.2783321

[10] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. 2017. ZipG: A Memory-efficient Graph Store for Interactive Queries. In Proc. 2017 ACM SIGMOD (SIGMOD '17). ACM, New York, NY, USA, 1149–1164. https://doi.org/10.1145/3035918.3064012

[11] S. Ma, R. Hu, L. Wang, X. Lin, and J. Huai. 2017. Fast Computation of Dense Temporal Subgraphs. In IEEE 33rd ICDE. 361–372. https://doi.org/10.1109/ICDE.2017.95

[12] Boldi, P., Santini, M., and Vigna, S. (2008). A large time-aware web graph. SIGIR Forum, 42, 33–38. URL: http://doi.acm.org/10.1145/1480506.1480511.

[13] Cambazoglu, B. B., and Baeza-Yates, R. (2015). Scalability Challenges in Web Search Engines. Morgan & Claypool Publishers.

[14] S. Sahu, S. S. J. L. M. T. O., A. Mhedhbi (2018). The ubiquity of large graphs and surprising challenges of graph processing. Proc. VLDB Endow., 10 ,1981–1984. URL: https://doi.org/10.14778/3137765.3137825. doi:10.14778/3137765.3137825.

[15] Vo, K. D., Tran, T., Nguyen, T. N., Zhu, X., and Nejdl, W. (2016). Can we find documents in web archives without knowing their contents? In Proc. 8th ACM WebSci WebSci '16 (pp. 173–182). New York, NY, USA:ACM. URL: http://doi.acm.org/10.1145/2908131.2908165. doi:10.1145/2908131.2908165.

[16] Kanhabua N., Kemkes P., Nejdl W., Nguyen T.N., Reis F., Tran N.K. (2016) How to Search the Internet Archive Without Indexing It. In: Fuhr N., Kovács L., Risse T., Nejdl W. (eds) Research and Advanced Technology for Digital Libraries. TPDL 2016. Lecture Notes in Computer Science, vol 9819. Springer, Cham. https://doi.org/10.1007/978-3-319-43997-6_12

[17] Fang, Y., Cheng, R., Li, X., Luo, S., and Hu, J. (2017). Effective community search over large spatial graphs. Proc. VLDB Endow., 10 , 709–720. URL: https://doi.org/10.14778/3055330.3055337. doi:10.14778/ 3055330.3055337.

[18] Kabiljo, I., Karrer, B., Pundir, M., Pupyrev, S., and Shalita, A. (2017). Social hash partitioner: A scalable distributed hypergraph partitioner. Proc. VLDB Endow., 10 , 1418–1429. URL: https://doi.org/10.14778/3137628.3137650. doi:10.14778/3137628.3137650.

[19] Lai, L., Qin, L., Lin, X., Zhang, Y., Chang, L., and Yang, S. (2016). Scalable distributed subgraph enumeration. Proc. VLDB Endow., 10 , 217–228. URL: https://doi.org/10.14778/3021924.3021937. doi:10.14778/ 3021924.3021937.

[20] Mai, S. T., Dieu, M. S., Assent, I., Jacobsen, J., Kristensen, J., and Birk, M.(2017). Scalable and interactive graph clustering algorithm on multicore CPUs. In IEEE 33rd ICDE (pp. 349–360). doi:10.1109/ICDE.2017.94.

[21] Zou, Z., Li, F., Li, J., and Li, Y. (2017). Scalable processing of massive uncertain graph data: A simultaneous processing approach. In IEEE 33rd ICDE (pp.183–186). doi:10.1109/ICDE.2017.70.

[22] M. Nelson, S. Radhakrishnan and C. N. Sekharan, "Algorithms on Compressed Time-Evolving Graphs," 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 227-232, doi: 10.1109/BigData47090.2019.9005704.

[23] Claude, F., and Navarro, G. (2010). Fast and compact web graph representations. ACM Trans. Web, 4 , 16:1–16:31. URL: http://doi.acm.org/10.1145/1841909.1841913.

[24] Brisaboa, N. R., Ladra, S., and Navarro, G. (2009). K2-trees for compact web graph representation. In Proc. 16th SPIRE SPIRE '09 (pp. 18–30). Berlin, Heidelberg: Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-03784-9_3.

[25] Li, L., Hua, W., Du, X., and Zhou, X. (2017). Minimal on-road time route scheduling on time-dependent graphs. Proc. VLDB Endow., 10, 1274–1285. URL: https://doi.org/10.14778/3137628.3137638.

[26] Najork, M. A. (2007). Comparing the effectiveness of hits and salsa. In Proc. 16th CIKM CIKM '07 (pp. 157–164). New York, NY, USA:ACM. URL: http://doi.acm.org/10.1145/1321440.1321465.

[27] Najork, M. A., Zaragoza, H., and Taylor, M. J. (2007). Hits on the web: How does it compare? In Proc. 30th SIGIR SIGIR '07 (pp. 471–478). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/1277741.1277823.

[28] Kruse, S., Hahn, D., Walter, M., and Naumann, F. (2017). Metacrate: Organize and analyze millions of data profiles. In Proc. ACM CIKM CIKM '17 (pp.2483–2486). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/3132847.3133180.

[29] Buehrer, G., and Chellapilla, K. (2008). A scalable pattern mining approach to web graph compression with communities. In Proc. 2008 WSDM (pp. 95–106). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/1341531.1341547.

[30] Moffitt, V. Z., and Stoyanovich, J. (2016). Towards a distributed infrastructure for evolving graph analytics. In Proc. 25th WWW WWW '16 Companion(pp. 843–848). Republic and Canton of Geneva, Switzerland. URL: https://doi.org/10.1145/2872518.2889290.

[31] Meimaris, M., Papastefanatos, G., Mamoulis, N., and Anagnostopoulos, I. (2017). Extended characteristic sets: Graph indexing for sparql query optimization. In IEEE 33rd ICDE (pp. 497–508). doi:10.1109/ICDE.2017.106.

[32] Berberich, K., Bedathur, S., Neumann, T., and Weikum, G. (2007). A time machine for text search. In Proc. 30th ACM SIGIR SIGIR '07 (pp. 519–526). New York, NY, USA: Association for Computing Machinery. URL: https://doi.org/10.1145/1277741.1277831.

[33] Silu Huang, James Cheng and Huanhuan Wu, "Temporal graph traversals: Definitions algorithms and applications", arXiv preprint arXiv:1401.1919, 2014.

[34] Huacheng Yu, "An improved combinatorial algorithm for boolean matrix multiplication", International Colloquium on Automata Languages and Programming, pp. 1094-1105, 2015.

[35] Holzmann, H., Nejdl, W., and Anand, A. (2017). Exploring web archives through temporal anchor texts. In Proc. 2017 WebSci WebSci '17 (pp. 289–298). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/3091478.3091500.

[36] Iosup, A., Hegeman, T., Ngai, W. L., Heldens, S., Prat-Pérez, A., Manhardto, T., Chafio, H., Capota, M., Sundaram, N., Anderson, M., Tanase, I. G., Xia, Y., Nai, L., and Boncz, P. (2016). Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. Proc. VLDB Endow., 9 , 1317–1328. URL: https://doi.org/10.14778/3007263.3007270.