Efficient Indexing and Representation of Web Access Logs *

Francisco Claude[1], Roberto Konow[1,2], and Gonzalo Navarro[2]

[1] Escuela de Informática y Telecomunicaciones, Universidad Diego Portales
fclaude@recoded.cl
[2] Department of Computer Science, University of Chile
{rkonow, gnavarro}@dcc.uchile.cl

Abstract. We present a space-efficient data structure, based on the Burrows-Wheeler Transform, especially designed to handle web sequence logs, which are needed by web usage mining processes. Our index is able to process a set of operations efficiently, while at the same time maintains the original information in compressed form. Results show that web access logs can be represented using 0.85 to 1.03 times their original (plain) size, while executing most of the operations within a few tens of microseconds.

1 Introduction

Web Usage Mining (WUM) [14] is the process of extracting useful information from web server access logs, which allows web site administrators, designers and engineers to understand the users' interaction with their web site. This process is used to improve the layout of the web site to better suit their users, or to analyze the performance of their systems in order to apply smart prefetching techniques for faster response, among other applications.

One particular WUM task is to predict the path of web pages that the user is going to traverse within a website. Accurately predicting the web user access behavior can minimize the user perception of latency, which is an important measure of the website quality of service [5,6,28]. This is achieved by fetching the web page *before* the user requests it. Another application is as a recommendation technique [19, 29]: the prediction can be displayed to the user, giving an insight of what the user might be looking for, therefore improving the user's experience. Other relevant mining operations include determining how frequently the path has been followed, which users have followed the path, and so on.

The prediction problem can be formalized as follows. Access logs obtained from web servers are used to extract the user's web site visit path as an ordered sequence of web pages $S_u = \langle v_1, v_2, v_3, \ldots, v_m \rangle$ (several sessions of the same user u might be concatenated into S_u). Therefore the system records the set

* This work was partially supported by the Conicyt PhD Scholarship, by Fondecyt Iniciación Grant 11130104, and by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F.

$\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of the accesses of each user. Given a new visit sequence (or path) P that is currently being performed by a user, the predicting task has to predict which page will be visited next. One näive approach to this problem is to first return the k web pages that have been visited most commonly by users after following the same path P, that is, we consider each time P appears as a substring of some S_u, and pick the most common symbols following those occurrences of P. After this process is done, more complex recommendation or machine learning algorithms [16] can be employed to accurately predict the next web page that is going to be visited. One particular challenge is that this operation needs to be done in an *on-line* manner, that is, we have to efficiently update our results as new requests are appended at the end of P. The system will eventually add those requests S' to the corresponding S_u sequences in $\mathcal{S}$, via periodic updates. At query time, the set $\mathcal{S}$ can be taken as static. The other mining operations are defined analogously.

Another interesting operation coming from WUM and general data mining is to retrieve the top-k most frequent sequences [12, 22] of a certain length. These are commonly used in retailing, add-on sales, customer satisfaction and in many other fields.

A typical WUM system faces two challenges: On the one hand, it has to manage huge amounts of data, that comes directly from the web access logs that store the records of all the interactions between the web server and the users. With the increasing amount of users and content on the Internet, handling this amount of data is a non-trivial task. On the other hand it has to provide accurate results. This is usually performed via a two stage process [16]: The first stage is a fast and simple filtration procedure that returns few hundreds or thousands of candidates from possibly millions of alternatives. During the second stage, more complex data mining techniques are performed to reduce the preliminary results to just a few high-quality results. In this paper we focus on improving the space consumption and time to perform queries on the web access sequences obtained from the web server logs used during the first step, thus freeing resources for the second stage and therefore increasing the performance of the process.

We present a space-efficient data structure in the Word-RAM model for representing web access logs, based on the Burrows-Wheeler Transform (BWT) [2]. Our index is able to efficiently process various queries of interest, while representing the data in compressed form. In this paper we focus on the following key operations; others are described in the Conclusions.

- $\mathit{Access}(u, i)$: Access the i-th web page visited by user u.
- $\mathit{UserPath}(u)$: Return the complete path done by user u.
- $\mathit{Count}(P)$: Count how many times path P has been performed in the collection.
- $\mathit{MostCommonNextPage}(P, k)$: Return the k most common web pages visited after path P.
- $\mathit{ListUsers}(P, k)$: Return k distinct users that have followed path P.
- $\mathit{MostFrequentPath}(k, q)$: Return the k most frequent paths of length q done by the users.

Our experimental results show that our index is able to represent the web access logs using 0.85–1.03 of their plain representations, thereby *replacing* them by a representation that uses about the same space but efficiently answers various queries, within microseconds in most cases. Our index can be easily deployed in other types of applications that handle ordered sequences, such as GPS trajectories, stock price series, customer buying history, and so on.

To our knowledge, this is the first compressed representation of logs that answers queries specific of WUM applications.

2 Basic Concepts

2.1 Rank and Select

Two basic operations used as building blocks for space-efficient data structures are *rank* and *select*. Given a bitmap B of length n, $rank_b(i)$ computes the number of bits b up to position i. The operation $select_b(j)$ retrieves the position where the j-th bit b appears. Munro [17] and Clark [3] obtained constant-time solutions for both operations while using $o(n)$ bits of space on top of B. Raman et al. [23] managed to compress the space to $nH_0(B) + o(n)$ bits[3] while still supporting both operations in constant time.

The wavelet tree [11] of a sequence S of length n over an alphabet of size σ extends the results for *rank* and *select* to general sequences, by decomposing the sequence hierarchically alphabet-wise in the form of a balanced tree. Internal nodes T_v store a binary string B_v. The root contains n bits, one per symbol in the sequence, and they are set to 0 or 1, depending on whether the corresponding symbol of S belongs to the lower or higher half of the alphabet. The left/right subtree is built for the subsequence of elements that have a 0/1 on the root. This decomposition continues, halving the alphabet, until the leaves, which correspond to a single symbol. All bitvectors are processed to handle binary rank and select queries in $O(1)$ time. This data structure accesses any $S[x]$ and solves $rank_b(S, x)$ and $select_b(S, x)$ in $O(\lg \sigma)$ time, using $n \lg \sigma(1 + o(1))$ bits (which is close to the space a plain representation of S would require).

2.2 Range Minimum Queries

A range minimum query asks for the position of the minimum element in a given range (i, j) of an integer array A of length n, that is, $RMQ_A(i, j) = argmin_{i \le k \le j} A[k]$. This query can be solved in constant time [8], after building a Cartesian tree over the array, convert it into a general tree using the usual bijection, and representing the general tree with a compact tree representation [27] that answers *lowest common ancestor (lca)* and other queries in constant time. This data structure requires $2n + o(n)$ bits.

The Cartesian tree of array $A[1, n]$ is a binary tree whose root corresponds to the minimum position i in A, its left child is the Cartesian tree of $A[1, i-1]$, and its right child is the Cartesian tree of $A[i+1, n]$.

[3] $H_0(B)$ is the zero-order entropy of the bitmap B.

2.3 Burrows-Wheeler Transform & The SSA Index

The Succinct Suffix Array (SSA) [20] is a compressed index that builds on the Burrows-Wheeler transform (BWT) of a text [2]. The main idea is to represent the BWT of a text T in compressed space and support pattern matching.

Given a text T of length N, ending with a unique symbol \$ smaller than the rest, the BWT corresponds to a permutation of the symbols in T that is reversible. A simple way to describe the transformation is to imagine the $N \times N$ matrix of the N cyclic rotations of the text T, sort the rows lexicographically, and then keep the last column of the matrix, $L[1, N] = BWT(T)$. The first column, formed by all the characters of T in order, is called $F[1, N]$. Note that any $F[i]$ is preceded by $L[i]$ in T.

It has been shown [15] that local compressors tend to handle the BWT of a text much better than the text itself, since the transformation tends to cluster together occurrences of the same symbol. This is not surprising, since the symbols are actually arranged according to their context (symbols appearing after it). An interesting operation that allows to support the ones we are interested in is known as the LF-mapping. $LF(i)$ tells where does $L[i]$ appear in $F[i]$, this way allowing us to retrieve the text that precedes it in T in backward form: $L[i]$, $L[L[i]]$, and so on. The LF operation can be computed as $LF(i) = rank_c(BWT(T), i) + occ[c]$, where $occ[c]$ corresponds to the number of symbols lexicographically smaller than c in T. By representing $BWT(T)$ with a wavelet tree [11], the LF operation takes $O(\lg \sigma)$ time, the same as accessing any position in $BWT(T)$.

The *backward search* operation [7] returns in $O(m \lg \sigma)$ time the range $L[sp, ep]$ from where all the occurrences of a given pattern P of length m can be located. It is called backward search since its procedure seeks the pattern in reverse order. Backward search is sufficient to compute the number of occurrences of P, as $ep - sp + 1$; this procedure is called *count*.

A Succinct Suffix Array (SSA) enhances the BWT representation with a sampling of some suffix array entries. This sampling is used to locate the actual positions where P occurs in T from the range $L[sp, ep]$. Given a *sampling factor* s_a, which requires $O((N/s_a) \lg N)$ further bits of space, the SSA *locates* the position in T of any of the $ep - sp + 1$ occurrences of P in $O(s_a \lg \sigma)$ time. With a similar sampling, the SSA can also *extract* any desired substring $T[l, r]$ in $O((s_a + r - l) \lg \sigma)$ time. By choosing any $s_a = \omega(\lg_\sigma N)$, the SSA index can be represented using $NH_k(T) + o(N \lg \sigma)$ bits of space, where $NH_k(T)$ is the k-th order entropy of the text T.

3 Indexing Web Access Sequences

3.1 Construction

We start by concatenating all ordered web access sequences $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ from all users into a sequence $\mathcal{T}(\mathcal{S})$ of size $N = \sum_{i=1}^{n} |S_i|$ over an alphabet of size σ, where σ is the amount of distinct web pages visited by any user in the

log file. Instead of building the BWT to index $\mathcal{T}(S)$, we construct the index over $\mathcal{T}(\mathcal{S})^R$, that is, $\mathcal{T}(\mathcal{S})$ has each S_i reversed. This simple trick allows us to maintain a range of elements that match the sequence of requests so far, while performing backward search, and thus allowing us to add new arriving requests by just performing one more step. Having the BWT of $\mathcal{T}(\mathcal{S})^R$ is not enough to reconstruct the information from the log. We also need to store the user identifier associated with each position in the sequence. To do so without spending $N \lg n$ bits to associate a user id to each position in the BWT of $\mathcal{T}(\mathcal{S})^R$, we construct a bitmap B of length N and mark with a 1 the positions where each S_i ends in $\mathcal{T}(\mathcal{S})^R$. We later index B to solve *rank* and *select* queries in constant time using compressed space [23]. This is enough to obtain the user associated to a location in the BWT of the sequence, by locating its original position p in the sequence and then performing $rank_1(B, p)$.

For listing the distinct users (strings) where path P occurs, we implement Muthukrishnan's document listing algorithm [18], as compressed by Sadakane [26]. We construct a temporary array $U[i] = rank_1(B, i)$, for $1 \leq i \leq N$, that stores the user ids and then permute the values so that the ids are aligned to the $BWT(\mathcal{T}(\mathcal{S})^R)$ sequence. Another integer array C is constructed by setting $C[i] = select_{U[i]}(U, rank_{U[i]}(U, i) - 1)$[4] for all $0 \leq i \leq N$, and then build a RMQ data structure on C. We keep this structure and discard C and U.

The RMQ data structure requires $2N + o(N)$ bits. The SSA index requires $NH_k(\mathcal{T}(\mathcal{S})) + o(N \lg \sigma)$ bits. The representation of bitmap B takes $H_0(B) + o(N) \leq N + o(N)$ bits. Note that we do not store the users, nor the frequencies in an explicit way.

3.2 Queries

Access(u, i). To obtain the i-th web page visited by user u within the sequence, we need to locate the position in the original sequence $\mathcal{T}(\mathcal{S})^R$ where the user's session begins. We can do this by computing $p = select_1(B, u + 1) - 1 - i$ and then applying *extract* $\mathcal{T}(\mathcal{S})^R[p, p]$ on the SSA index, in $O(s_a \lg \sigma)$ time.

UserPath(u). To obtain the path done by user u, we compute $p_1 = select_1(B, u)$ and $p_2 = select_1(B, u + 1) - 1$ and then *extract* $\mathcal{T}(\mathcal{S})^R[p_1, p_2]$ using the SSA index. This takes $O((\ell + s_a) \lg \sigma)$ time, where $\ell = p_2 - p_1$ is the length of the extracted path.

Count(P). Given a path P of length m we can count its occurrences, by just performing $Count(P)$ on the SSA index in $O(m \lg \sigma)$ time.

MostCommonNextPage(P, k). We describe this operation incrementally. Assume we have already processed the sequence of requests $P = r_1, r_2, \ldots, r_{m-1}$. Our invariant is that we know the interval $[sp, ep]$ corresponding to path P, and

[4] To avoid corner cases, we define $select_{U[i]}(U, 0) = -1$.

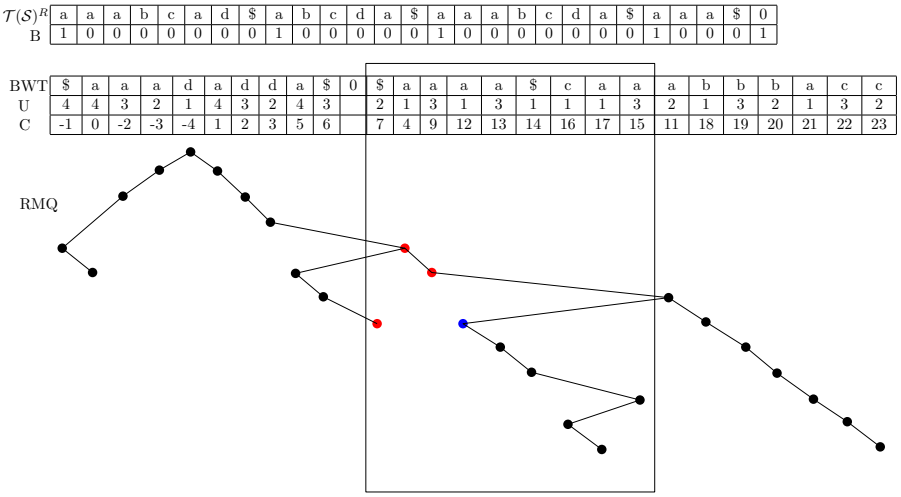

Fig. 1. Layout of our index organization using example sequences: $S_1 = dacbaaa, S_2 = adcba, S_3 = adcbaaa$ and $S_4 = aaa$. The dollar (\$) symbol is used to represent the end of each sequence and the zero symbol is used to mark the end of the concatenation of the sequences $\mathcal{T}(\mathcal{S})^R$. At the bottom we show the topology of the Cartesian tree built over the C array representing the RMQ data structure. Recall that arrays U and C are not represented and are only shown for guidance.

this is sufficient to answer query $MostCommonNextPage(P, k)$. Now a new request r_m arrives at the end of P. Then we proceed as follows:

1. Update the range $[sp, ep]$ in $BWT(\mathcal{T}(\mathcal{S}))$ using r_m, in $O(\lg \sigma)$ time [7].
2. Retrieve the k most frequent symbols in $BWT[sp, ep]$, which are precisely those preceding the occurrences of P^R in $\mathcal{T}(\mathcal{S})^R$, or following P in $\mathcal{T}(\mathcal{S})$.

The second step is done with the heuristic proposed by Culpepper et al. [4] to retrieve the k most frequent symbols in a range of a sequence represented with a wavelet tree. It is a greedy algorithm that starts at the wavelet tree root and maps the range (in constant time, using $rank$ on the bitmap of the wavelet tree node) to its children, until reaching the leaves. The traversal is prioritized by visiting the longest ranges first, and reporting the symbols corresponding to the first k leaves found in the process. The worst-case performance of this algorithm is bounded by the number of different symbols present in the string range. This is smaller than both σ and the size of the range. By using more sophisticated data structures (that nevertheless do not add much space) [13, 21], a worst case of $O(k + \text{polylog}\, n)$ time can be guaranteed. Note that, if we want to list *all* the request that have followed P, we can use an optimal algorithm based on depth-first traversal of the wavelet tree [9].

ListUsers(P, k). To list k (or all) distinct users that have followed path P we first locate the starting and ending points $[sp, ep]$ for the given path, using the

SSA index in $O(m \lg \sigma)$ time (we can also proceed incrementally as in operation *MostCommonNextPage*). Then we apply the optimal document listing algorihtm [18, 26]. Each value $C[p] < sp$, for $sp \leq p \leq ep$, signals a distinct value of $U[p]$ in $U[sp, ep]$. Recall that we do not have C or U anymore, but the procedure for extracting the list of users can be emulated with the RMQ data structure over array C and the bitmap B. The procedure for extracting the list of all users works as shown in Algorithm 1. Function *locate* takes a position in the BWT and maps it to the corresponding position in the original $\mathcal{T}(\mathcal{S})^R$, in $O(s_a \lg \sigma)$ time. Listing k distinct users for path P takes $O((k \cdot s_a + p) \lg \sigma)$ time.

Fig. 1 shows an example of listing users. The framed region represents the range sp, ep. Red nodes in the RMQ tree represent the position of the retrieved users, while the blue node represents the last visited node before returning.

Algorithm 1 – UserListing(sp, ep, users = ∅)

```
p ← RMQ_C(sp, ep)
if ep < sp then
   return users
end if
u ← rank_1(B, locate(p))
if u ∉ users then
   users ← users ∪ u
   UserListing(sp, p − 1, users)
   UserListing(p + 1, ep, users)
end if
```

MostFrequentPath(k, q). We want to retrieve the k most frequent paths of a certain length q done by the users in the system. We start by pushing into a priority queue all ranges obtained by performing a backward-search of paths of length 1 for each possible symbol (σ at most). Now, we extract the biggest range from the priority queue as well as the path that created that range. We execute the same procedure again, creating new paths by appending to the extracted path one further symbol (trying the σ possible ones in the worst case) and we push the ranges obtained by performing the backward search on these new paths into the priority queue. When we extract a path of length q, we report it and remove it from the priority queue. The procedure ends when k paths are reported or when the priority queue is empty. In the worst case, this operation can take $O(\sigma^q)$.

This method may perform poorly when the alphabet σ is large. An optimization is to avoid trying out all the σ characters to extend the current path, but just those symbols that do appear in the current range. Those are found by traversing the wavelet tree from the root towards all the leaves that contain some symbol in the current range [9].

Data Structure	Msnbc	Kosarak	Spanish
SSA	3,175,504	12,500,964	75,667,016
RMQ	1,770,690	2,807,570	39,538,238
Users Bitmap	608,044	777,804	9,413,732
Total	**5,554,246**	**16,086,346**	**124,618,994**
Plain	**5,782,495**	**18,884,286**	**120,613,202**
Ratio	**0,96**	**0,85**	**1,03**

Table 1. Space usage, in bytes, of the data structures used in our index. Plain corresponds to the sum of the space usage of plain representations of the sequence and users. Ratio corresponds to the total index size divided by the plain representation size.

4 Experiments and Results

Setup and implementations. We used dedicated server with 16 processors Intel Xeon E5-2609 at 2.4GHz, with 256 GB of RAM and 10 MB of cache. The operating system is Linux with kernel 3.11.0-15 64 bits. We used g++ compiler version 4.8.1 with full optimizations (-O3) flags.

We implemented the SSA index using the public available wavelet tree implementation obtained from libcds (`http://www.github.com/fclaude/libcds`) and developed the heuristic proposed by Culpepper et al. [4] on top of that implementation. The wavelet tree needed for the SSA index uses a RRR [23] compressed bitmap representation. The bitmap B needed to retrieve the users is used in plain form [10]. We implemented the RMQ data structure based on compact tree representations [1], which in practice requires $2.38n$ bits. Our implementation is available at `https://gitlab.com/fclaude/wum-index/`.

Experimental data. We used web access sequences from the public available Msnbc, Kosarak, and Spanish datasets. The Msnbc dataset comes from Internet Information Services log files of msnbc.com for a complete day of September, 28 of 1999. It contains web access sequences from $989,818$ users with an average of 5.7 web page categories visits per sequence, the alphabet size of this dataset is $\sigma = 17$. The Kosarak dataset contains the click-stream data obtained from a Hungarian on-line news portal. It contains sequences of $990,000$ users with an average of 8.1 web page visits per sequence and an alphabet size $\sigma = 41,270$. Finally, the Spanish dataset contains the visitors' click-stream obtained from a Spanish on-line news portal during September 2012. This dataset consists of $9,606,228$ sequences of news-categories that were visited, with an average of 12.3 categories visited per sequence and an alphabet size of $\sigma = 42$.

Space usage. Table 1 shows the space usage of each data structure for each dataset. Row "Plain" shows the space required to represent the original sequence $\mathcal{T}(\mathcal{S})^R$ using an array of $N \lg \sigma$ bits plus an array to represent the users using $n \lg N$ bits. The table shows that our index is able to compress the sequence

by up to 15%, on the Kosarak dataset, while requiring only 3% extra space at most, on the Spanish dataset. Within this space, we are able to support the aforementioned operations, while at the same time can reconstruct the original sequence and the users information.

We evaluated alternatives to the SSA, such as the *Compressed Suffix Array (CSA)* [24] and the *Compressed Suffix Tree (CST)* [25], using the ones provided by the sdsl-lite library (`https://github.com/simongog/sdsl-lite`). Table 2 compares their space usage to our SSA at representing the sequence. The SSA is a better choice in this case, using up to 33% less space than the others. The CST could be used to compute some of the operations, since it is naturally a faster alternative. In fact, we evaluated this alternative for counting the occurrences of a sequence, and it is in practice four to twenty times faster than the SSA. We discarded it since the space requirement makes it unpractical for massive datasets. In fact, the CST needs to be augmented in order to support all the operations presented in this paper, which would increase its memory usage.

Time performance. To evaluate the main operations using our proposed data structure, we generate query paths by choosing uniformly at random a position in the original sequences, and then extracting a path of the desired length.

Fig. 2 (left) shows the time to access positions chosen uniformly at random from users whose traversal log has length 1 to 100. The time does not depend on the length, but directly on $s_a \lg \sigma$. Fig. 2 (right) shows the time per symbol extracted, when we access the whole sequence associated with a user. We can see that for users with short interactions the SSA sampling has a greater effect, and this is amortized when accessing longer sequences, that is, the term $O(s_a \lg \sigma)$ is spread among more extracted symbols and the time converges to $\lg \sigma$.

We then measured the time to count the number of times a certain path appears in the access sequence. Fig. 3 (left) shows the time per query. As expected, it grows linearly with the length of the path being counted, and the $\lg \sigma$ term determines the slope of the line. On Fig. 3 (right) we show the results for the *MostCommonNextPage* operation. The x-axis and y-axis are in log scale. For datasets containing small alphabets such as Msnbc ($\sigma = 17$) and Spanish ($\sigma = 42$) this operation is performed in under 30 microseconds for all possible values of k (note it is impossible to obtain more than σ distinct symbols), and shows a logarithmic behavior. We also note that the slope of the logarithm

Data Structure	**Msnbc**	**Kosarak**	**Spanish**
SSA	1,08	0,77	0,85
CSA	1,61	0,87	1,13
CST	3,82	1,43	2,96

Table 2. Comparison of the space consumption of the SSA, CSA, and CST for representing sequence $\mathcal{T}(\mathcal{S})^R$. We show the ratio of each index space over the plain representation of the sequence without the user's information by using $N \lg \sigma$ bits.

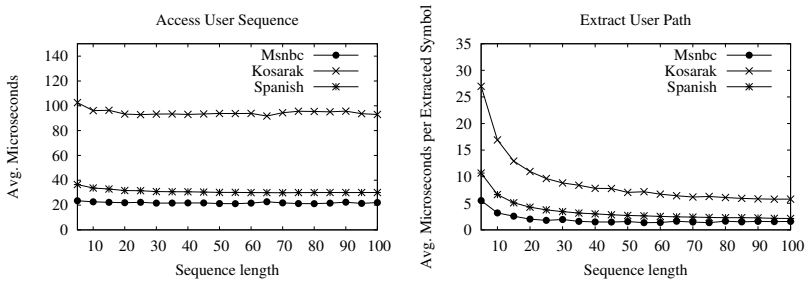

Fig. 2. On the left, average microseconds to perform $Access(u, i)$; on the right, average microseconds per extracted symbol for the operation $UserPath(u)$.

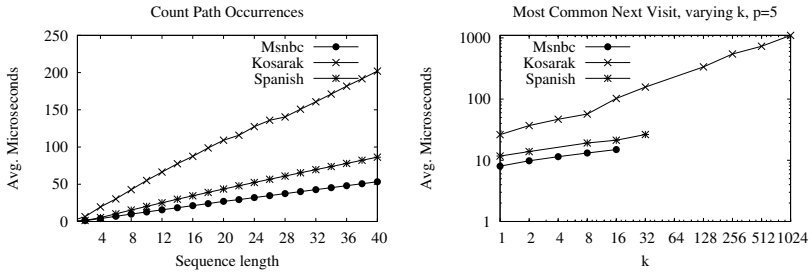

Fig. 3. On the left, average microseconds to perform $Count(P)$; on the right, average microseconds for the *MostFrequentPath* operation with varying k using patterns of fixed length $p = 5$.

depends on the value of σ, as shown in the Kosarak dataset. The operation, however, is still reasonably fast, taking less than 1 millisecond for $k = 1024$.

Fig. 4 (left) shows the time for retrieving the set of users that followed a given access pattern in the system. For shorter sequences, the index has to retrieve a bigger set of users, as these sequences are more likely to appear. As the sequences grow in length, the time decreases, since the resulting set is also smaller. The behavior for Kosarak, which after a certain point starts increasing in time per query, can be explained by the fact that determining the range $[sp, ep]$ grows linearly with the length of the pattern, and at some point it dominates the query time. This is also expected, at a later point, for Msnbc and Spanish.

Finally, Fig. 4 (right) shows the query time for operation $MostFrequentPath(k, q)$. Our first implementation tried following all symbols at every step of the algorithm. This worked quite well for small alphabets but had a very bad performance on the Kosarak dataset. This plot shows the implementation traversing the wavelet tree at each step to only follow symbols that do appear in the range. This gives a slightly worse performance for small alphabets, but a considerable speedup (1–2 orders of magnitude) for larger ones.

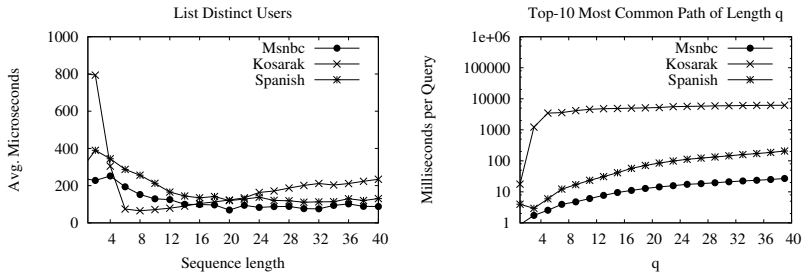

Fig. 4. On the left, average microseconds to list distinct users that traversed paths with varying lengths. On the right, the time required to perform top-10 most common path operation for varying pattern lengths.

5 Discussion and Future Work

We introduced a new data structure for handling web access sequences in compressed form that fully replaces the original dataset and also supports useful operations for typical WUM processes. Our experiments show that our index uses about the same size of the plain representation of the data, and within this space supports various relevant operations within tens of microseconds. This is competitive; consider that in most common scenarios the systems have to reply over a network, which have considerable latency and transfer time. Ours is the first compressed representation tailored for this scenario.

We have not yet fully explored other possible operations of interest in log mining that can be supported with our arrangements. For example, we can count the number of users that followed some path in constant time using $2n + o(n)$ bits using document counting [26], compute the k users that have followed a path most frequently using top-k document retrieval [13], and others.

Our index can be easily adapted to custom scenarios by adding satellite information, such as duration of the visit, actions (buy, login/logout, comment, etc.), browser information, location, and others, to each event in the log and include the information by mapping it to the BWT transform for later processing. This enables our index to be applied in other scenarios involving ordered sequences, such as GPS trajectories, stock price series, customer buying history, and so on.

References

1. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. 11th ALENEX. pp. 84–97 (2010)
2. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep., Digital Equipment Corporation (1994)
3. Clark, D.: Compact Pat Trees. Ph.D. thesis, Univ. of Waterloo, Canada (1996)
4. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k ranked document search in general text databases. In: Proc. 18th ESA. pp. 194–205 (2010)

5. Domènech, J., Gil, J.A., Sahuquillo, J., Pont, A.: Web prefetching performance metrics: a survey. Perform. Eval. 63(9), 988–1004 (2006)
6. Dongshan, X., Junyi, S.: A new markov model for web access prediction. Computing in Science and Eng. 4(6), 34–39 (2002)
7. Ferragina, P., Manzini, G.: Indexing compressed texts. J. ACM 52(4), 552–581 (2005)
8. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comp. 40(2), 465–492 (2011)
9. Gagie, T., Navarro, G., Puglisi, S.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comp. Sci. 426-427, 25–41 (2012)
10. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. Posters 4th WEA. pp. 27–38 (2005)
11. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th SODA. pp. 841–850 (2003)
12. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Mining Knowl. Disc. 15(1), 55–86 (2007)
13. Hon, W.K., Shah, R., Vitter, J.: Space-efficient framework for top-k string retrieval problems. In: Proc. 50th FOCS. pp. 713–722 (2009)
14. Hussain, T., Asghar, S., Masood, N.: Web usage mining: A survey on preprocessing of web log file. In: Proc. ICIET. pp. 1–6 (2010)
15. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
16. Mobasher, B.: Data mining for web personalization. In: The adaptive web, pp. 90–135. Springer (2007)
17. Munro, J.I.: Tables. In: Proc. 16th FSTTCS. vol. LNCS 1180, pp. 37–42 (1996)
18. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. 13th SODA. pp. 657–666 (2002)
19. Nadi, S., Saraee, M., Davarpanah-Jazi, M.: A fuzzy recommender system for dynamic prediction of user's behavior. In: Proc. ICITST. pp. 1–5 (2010)
20. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1) (2007)
21. Navarro, G., Valenzuela, D.: Space-efficient top-k document retrieval. In: Proc. 11th SEA. pp. 307–319. LNCS 7276 (2012)
22. Pei, J., Han, J., Mortazavi-Asl, B., Zhu, H.: Mining access patterns efficiently from web logs. In: Proc. 4th PAKDD, pp. 396–407. Springer (2000)
23. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Alg. 3(4), art. 43 (2007)
24. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Alg. 48(2), 294–313 (2003)
25. Sadakane, K.: Compressed suffix trees with full functionality. Theor. Comp. Sys. 41(4), 589–607 (2007)
26. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discr. Alg. 5(1), 12–22 (2007)
27. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. 21st SODA. pp. 134–149 (2010)
28. Su, Z., Yang, Q., Lu, Y., Zhang, H.: Whatnext: A prediction system for web requests using n-gram sequence models. In: Proc. 1st WISE. pp. 214–224 (2000)
29. Sumathi, C., Valli, R.P., Santhanam, T.: Automatic recommendation of web pages in web usage mining. Intl. J. Comp. Sci. Eng. 2, 3046–3052 (2010)