Suffix Rank: a new scalable algorithm for indexing large string collections

Marina Barsky
Bard College at Simon's Rock
mbarsky@simons-rock.edu

Jonathan Gabor
Bard College at Simon's Rock
jgabor16@simons-rock.edu

Mariano P. Consens
University of Toronto
consens@cs.toronto.edu

Alex Thomo
University of Victoria
thomo@uvic.ca

ABSTRACT

We investigate the problem of building a suffix array substring index for inputs significantly larger than main memory. This problem is especially important in the context of biological sequence analysis, where biological polymers can be thought of as very large contiguous strings. The objective is to index every substring of these long strings to facilitate efficient queries. We propose a new simple, scalable, and inherently parallelizable algorithm for building a suffix array for out-of-core strings. Our new algorithm, *Suffix Rank*, scales to arbitrarily large inputs, using disk as a memory extension. It solves the problem in just $O(\log n)$ scans over the disk-resident data. We evaluate the practical performance of our new algorithm, and show that for inputs significantly larger than the available amount of RAM, it scales better than other state-of-the-art solutions, such as *eSAIS*, *SAscan*, and *eGSA*.

PVLDB Reference Format:
Marina Barsky, Jonathan Gabor, Mariano P. Consens, and Alex Thomo. Suffix Rank: a new scalable algorithm for indexing large string collections. *PVLDB*, 13(11): 2787–2800, 2020.
DOI: https://doi.org/10.14778/3407790.3407861

1. INTRODUCTION

The size of data collections increases with unprecedented speed. A large proportion of these data is sequential and can be modeled as strings. Some string collections are extremely large. For example, sequence datasets collected by the 1000 genomes project have reached 4.2TB in total size and are growing [33]. The International Cancer Genome Consortium has collected about 150,000 DNA sequences of at least 100GB each [35]. To perform efficient data analysis at this scale, we need to build a special-type index, a ***full-text substring index***. Major full-text indexes include *Suffix Trees* [31], *Suffix Arrays* [21], and *FM-indexes* [8]. These indexes expose the internal structure of monolithic strings and provide dramatic performance improvements for multiple complex string problems. Although full-text indexes are already widely used for biological sequence analysis [30], they do not typically scale to larger inputs.

Of all different index types, in this paper we concentrate on **suffix arrays**. The *Suffix Array* data structure was co-introduced in [21, 11] as a space-efficient alternative to the *Suffix Tree* [31]. The suffix array SA for a string X of length N is defined to be an array of N integers corresponding to the starting positions of suffixes of X in lexicographical order (see Figure 1).

The main query type supported by the suffix array is *pattern search*: given a query string Q (*pattern*) and a large string X (*text*), find all the occurrences of Q in X. Every substring of X is a prefix of some suffix, so if the suffixes are sorted, any pattern can be located in time $O(\log N)$ using binary search. This is a significant improvement over a naive $O(N)$ search, especially when N is very large. Moreover, sorted suffixes can be used to build an *FM-index* [8]. FM index can be compressed to occupy less space than the original string. Thus, the FM-index of a very large string could fit into RAM, even if the original string could not, and this facilitates even more efficient pattern search queries. The efficiency of the pattern search is crucial because it is used in many bioinformatics tasks, such as short-read mapping [12, 19], sequence alignment [16, 18], repeat detection [1], or genome assembly [6, 27].

Formally, in this paper we tackle the following problem:

PROBLEM 1. **External-memory suffix sorting**
Input*: String X of length N where $N >> M$, and M is the (limited/constant) amount of internal memory.*
Output*: A suffix array SA_X on disk.*

We propose a new external-memory algorithm for solving this specific problem. Our new algorithm, *Suffix Rank*, works in a constant amount of RAM and uses disk as a memory extension. The input string resides on disk and is never accessed at random positions. The input can be arbitrarily large, limited only by the disk space required to store intermediate data structures.

Our new algorithm has the following **properties**:

- For an input string N characters in length, the algorithm **runs in time** $O(N \log N)$. At most $\log N$ scans

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407861

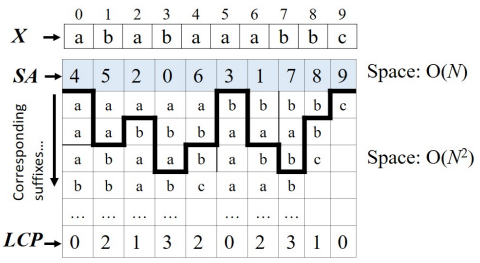


Figure 1: Suffix array SA for string X. Note that the SA contains only start positions, not the suffixes themselves. The last row contains the values of LCP - the length of the longest prefix shared by a suffix with the suffix preceding it in the suffix array.

of the input are required to produce the index. In practice, the number of scans is equal to $\log(max|LCP|)$, where LCP is the value of the longest common prefix shared by any two suffixes.

- The algorithm completely **avoids random access** to disk-resident data structures. It efficiently exploits the properties of real disks, including prefetching of large amount of data from a single local point of the input.
- The space complexity of the algorithm is **linear** in N.
- The algorithm is inherently **parallelizable**: it requires access to at most two small chunks of the input at a time, and performs shared-nothing computations for each chunk.
- The algorithm is unique in its **simplicity**; it can be easily implemented and used in production environment for indexing massive string collections.

Before presenting our solution, we review the existing methods for suffix array construction.

2. RELATED WORK

SUFFIX ARRAY FROM SUFFIX TREE

In general, the suffix array can be obtained by traversing an already built suffix tree. However, suffix tree construction involves significant overhead in terms of space, which makes it too expensive when only sorted suffixes are required. Linear-time suffix tree construction algorithms are limited to the main-memory settings [31, 29]. They exhibit poor locality of references and their performance degrades even when run in RAM due to multiple cache misses; this can be improved but not eliminated [28]. Suffix tree construction for strings which are much larger than RAM remains a challenge. Existing algorithms for out-of-core inputs are limited to *Wavefront* ($O(N^3)$ running time) [10], *BBST* ($O(N^2)$)[2], and *ERA* ($O(N^2 logN)$) [22]. These algorithms scale moderately with respect to the input-to-memory ratio. In this paper, we do not discuss the problem of constructing a suffix tree from which to derive a suffix array, but rather the problem of constructing a suffix array directly from the input string.

IN-MEMORY SUFFIX SORTING

A naive approach to the suffix array construction would be to sort suffixes as though they were separate strings. Such sorting would require comparison of N substrings of size $O(N)$ each, and would run in time $O(N^2 \log N)$. Much more efficient algorithms take advantage of the fact that suffixes are *overlapping* substrings of the same string. Already in [21], the authors proposed the first algorithm which runs in time $O(N \log N)$. The suffix array construction algorithms can be categorized into three groups [26]: *Prefix Doubling* algorithms such as *qsufsort* [17], *Recursive Sorting* such as *DC-3* [15], and *Induced Sorting* such as *SAIS* [24].

As noted in [26], a problem with all these sophisticated suffix sorting algorithms is that they are memory-latency bound – they access data at seemingly random positions and incur multiple cache misses even when run in RAM. Memory-latency is the reason why the linear-time *DC-3* algorithm can perform worse than the $O(N \log N)$ *qsufsort*, and both can be slower than $O(N^2 \log N)$ naive suffix sorting on some inputs [25]. This applies to all cases when the data needs to be transferred up the memory hierarchy: from slower to faster memory. When the data structures used in these algorithms outgrow the main memory and must be kept on disk, the computation becomes practically infeasible, because we constantly need to read new values from random disk locations discarding the values that are currently buffered in RAM. Furthermore, if the input is also kept on disk, brute-force comparison of substrings from random positions is not an option either. To work with such strings, algorithms must be redesigned to avoid random access to disk-resident data.

EXTERNAL-MEMORY CONSTRUCTION

To design a suffix sorting algorithm fully optimized for modern hardware and the properties of real disks, one might think of using the disk-friendly ***2-Phase Multi-way Merge Sort*** ($2PMMS$): partition the input into small chunks, sort suffixes in each chunk in RAM, and write these chunk suffix arrays to disk. The second phase would be to merge sorted suffixes from different partitions. However, directly applying the merge step of $2PMMS$ to suffixes is not possible: recall that sorted suffixes of each chunk are represented by their starting positions only, so there is not enough information to determine the relative order of suffixes from two different partitions. The only way to compare two such suffixes is to consult the input string itself, and when the input string is on disk, it cannot be read efficiently at random positions guided by the order in chunk suffix arrays.

Random access to the input string can be reduced by attaching a certain number of initial characters from each suffix to its position in the partitioned suffix arrays and comparing these prefixes during the merge. This technique of *prefix buffering* was employed by the *DiGeST* algorithm [3]. Prefix buffering was shown to eliminate about 90% of random accesses to the input string in experiments with genomic sequences [3]. However, the remaining 10% of random disk I/Os severely degrade the performance of the algorithm when the input string resides on disk. More sophisticated prefix buffering was proposed in [20]. The ***eGSA*** algorithm uses three techniques to accelerate the merge: prefix assembly, LCP comparison, and suffix induction. Despite these techniques, *eGSA* does not guarantee sequential disk I/Os for inputs containing long repetitive strings, such as a set of multiple human genomes [20]. In addition, there is a significant disk space overhead to store the prefix buffers.

Instead of prefix buffering, information about the relative order of suffixes from different partitions could be precomputed and used during merge. This technique was first used

in *BBST* [2]. The *BBST* algorithm runs in time $O(KN)$, where $K = N/M$ is the input-to-memory ratio, and M represents available amount of main memory. In essence, the algorithm scales quadratically in N, and is practical only for cases where K is small. The disk space required for temporary data structures is also $O(KN)$.

A more sophisticated partition-and-merge strategy was proposed in the ***SAscan*** algorithm [13]. As in *BBST*, *SAscan* breaks the input string X into K chunks ($K = N/M$). Then, it computes the Suffix Array and BWT for each chunk, and writes them to disk. Next, for each chunk, it constructs a gap array. The *gap array* of chunk k indicates how suffixes of X starting after chunk k are interleaved with suffixes starting in chunk k. After the gap arrays for each chunk are computed, the algorithm has enough information about the relative order of suffixes from different partitions to merge them into a single global suffix array. While computing the gap array for each chunk k, the *SAscan* algorithm needs to scan disk–resident information from all chunks following it – from $k + 1$ to K. Thus, disk data of order N is scanned K times. Since the number of chunks K depends on N/M, the algorithm is again quadratic in N, like *BBST*. However, unlike in *BBST*, the temporary disk space is linear in N. The *SAscan* algorithm is extremely fast when K is small. In addition, the algorithm is lightweight: it uses sophisticated compressed data structures, and thus its temporary disk footprint is quite small.

The algorithms that do not rely on multi-way merge sort extend in-memory techniques to external memory settings. In [7] the authors investigate the external-memory potential of the algorithms based on *prefix doubling* (the Manber-Myers algorithm) and based on *recursive sorting* (*DC-3*). They propose pipelined versions of both algorithms. The experiments in [7] show that proposed solutions do not scale to large inputs due to significant amount of random disk I/Os and multiple recursive steps.

An algorithm based on *induced sorting* was proposed in [5]. The authors modify the in-memory *SAIS* algorithm [24] for the external-memory settings. Their redesigned ***eSAIS*** algorithm has running time proportional to that of sorting in the Disk Access Model (DAM), an idealized model of computation. In this model, complexity is measured by the number of disk block transfers. However, in practice, transferring blocks from random disk locations is several orders of magnitude slower than when these blocks are read and written sequentially. This is due to the prefetching of large amounts of data in modern disk buffers: the need to move to a different disk location to read another block instead of using the prefetched data makes random-access algorithms infeasible for large inputs. The first step of *eSAIS* requires sorting a subset of suffixes as though they were separate substrings. For some inputs, these substrings can be quite large, and sorting them requires random access to more than one block of input. To sort such long substrings, the algorithm uses the doubling technique and an EM priority queue, which is not designed to hold variable-length substrings and does not guarantee that the required strings will be in memory when the algorithm needs it. The authors use heuristics in order to predict which part of the input will be accessed next, and pre-load this part into the main memory buffer. *eSAIS* does not perform exclusively sequential I/Os to the disk-based data structures.

Attempts to improve the performance of *eSAIS* were undertaken in [23, 32]. The new implementations called *DSA-IS* and *DSA-IS+* improve upon the substring sorting step, reducing peak disk usage by 30% and 50% respectively. Despite these disk space improvements, the experiments in [32] show that both enhanced algorithms are still outperformed by the original *e-SAIS* as the input grows. A new external-memory implementation of the *SAIS* algorithm called *fSAIS* was proposed in [14] and is under active development. It must be noted, however, that all the algorithms based on induced sorting are not easily parallelizable: they rely on the order of substrings which cross boundaries of the different input partitions. This separates them from our new algorithm which can process a single independent chunk of input at a time.

Next, we show how the suffixes of an out-of-core string can be sorted in total time $O(N \log N)$, using constant amount of RAM, linear temporary disk space, and performing exclusively sequential disk I/Os.

The rest of the paper is organized as follows. We begin with useful definitions in the Section 3. The new algorithm is described in Section 4. In Subsection 4.1 we prove the correctness of our approach. We discuss input partitioning in Subsection 4.2 and initialization in Subsection 4.3. Subsections 4.4 - 4.6 are dedicated to a detailed description of the *Suffix Rank* steps. We analyze the upper bounds for time and space complexity in 4.7, and conclude with Section 5, which presents an experimental evaluation of our new algorithm and its comparison with the state-of-the-art in the field.

3. PRELIMINARIES

Let the string $X = [x_1 \dots x_N]$ be a sequence of N characters over the finite alphabet Σ. The sequence of characters $X[i, j]$, where $i \leq j$, is called a *substring* of X. A *suffix* is defined to be a substring that starts at some position i and ends at position N. Each suffix is uniquely identified by its starting position. For example, the suffix S_4 of the string *banana* is *ana*, and S_1 is *banana*, the string itself. Similarly, the *prefix* P_j is a substring starting at position 1 and ending at some position j. Each prefix is uniquely identified by its ending position.

The goal is to sort the suffixes of X lexicographically and store their starting positions in an array – the *Suffix Array* of X. Formally, the *Suffix Array* of X SA_X is a permutation of the integers $[1 \dots N]$ such that $S_{SA_X[i]} < S_{SA_X[i+1]}$ for all $i < N$. Because every substring of X is a prefix of some suffix, it can be efficiently located in the array of sorted suffixes: the Suffix Array facilitates substring queries over input string X, and constitutes an *index* of X.

Note that each suffix of X occupies a unique position in the suffix array, since no two suffixes can be identical. We say that each suffix has its own unique final *rank*. For example, for input string $X = banana$, rank $R(S_6) = 1$, because this suffix - the substring 'a'- is the smallest of all suffixes of X. Similarly, ranks $R(S_4) = 2$, and $R(S_2) = 3$. Computing the final unique rank of each suffix is equivalent to finding its position in the suffix array.

The input is often a set of strings rather than a single string. Let Ω be a set of k strings with total length N. The *Generalized Suffix Array* of Ω is a permutation of the integers $[1, \dots N]$ that represents the lexicographic order of the suffixes of **all the strings** in the set. As with a single

string, the generalized suffix array can be efficiently searched for any query string using binary search in $O(\log N)$ time.

When the input contains multiple strings, it is possible for two different suffixes to be identical. However, if we append a unique character to the end of each string in Ω, then all the suffixes are guaranteed to be distinct. These characters, called *sentinels*, are not part of the input alphabet Σ and are considered to be lexicographically smaller than any symbol in Σ. When the number of strings in Ω is extremely large, it is difficult to represent numerous distinct sentinels. In practice, instead of using a unique character at the end of each string, we use a unique rank. During the initialization step, we add an additional position to the end of each string, and set the suffix rank at this position to the next available integer. Because we use 8-byte integers to store ranks, we practically cannot run out of unique sentinels. The smallest rank of the actual input characters then becomes the total number of sentinels+1. After appending a unique marker to the end of each string, we can reduce the construction of the generalized suffix array to the construction of a regular suffix array by concatenating all of the strings in Ω. Note that concatenating without this preprocessing would introduce some artificial suffixes which were not present in the original input. The unique sentinel at the end of each string prevents the characters past its position from being considered in sorting the suffixes of the corresponding string.

From now on, we only discuss suffix array construction for a single input string X of size N, keeping in mind that the same algorithm can be applied to a construction of a generalized suffix array after the above-mentioned preprocessing.

4. NEW ALGORITHM

Our algorithm combines ideas from the in-memory *qsufsort* [17] and disk-friendly $2PMMS$. The original *qsufsort* algorithm at each iteration i refines the order of suffix at position j by **looking** at the order at position $j+2^i$. It assumes constant-time access to each position of the partially built suffix array, and this only works if the array fits into RAM. As discussed in Section 2, we cannot apply the $2PMMS$ directly either: if we completely sort the suffixes in each partition, we would not have enough information to merge them. We propose a different strategy. Instead of completely sorting all suffixes in each partition, our algorithm proceeds in iterations. At each iteration, it doubles the length of the prefixes under consideration, produces tentative counts, and merges them using efficient buffered merge. After it employs at most $\log N$ disk-friendly merges, the suffixes of the entire input string are fully sorted.

Since the size of X significantly exceeds the amount of available main memory, we break it into separate partitions and work with one partition at a time. Each iteration of the algorithm consists of three steps: *refine*, *resolve* and *update*. In the *refine* and *update* steps, only a small part of the input is processed in main memory at a time. In the *resolve* step, we handle memory efficiently using buffering techniques from $2PMMS$.

We begin by proving the correctness of the doubling principle used for computing suffix ranks.

4.1 The doubling principle

Let the *final rank* of a suffix be its position in the suffix array. We start by assigning a very coarse rank to each suffix based on its first character. At this point, many suffixes will have the same rank. In the subsequent iterations, we refine these ranks using the doubling technique described below.

Let $P(i,h)$ be the prefix of length 2^h of suffix S_i, i.e. $X[i, \min(i+2^h-1, N)]$. Let h-rank(i) be the number of integers j such that $P(j,h) < P(i,h)$. Figure 2 shows the 1-ranks, i.e. ranks based on prefixes of length 2^1, for each position in a sample string.

The following theorem allows us to, given the h-rank of each suffix in a string, calculate their $(h+1)$-ranks.

THEOREM 1. *Let j-count(i,h) be the number of positions j such that h-rank$(i) = h$-rank(j) and h-rank$(j+2^h) < h$-rank$(i+2^h)$.*[1]

$$(h+1)\text{-rank}(i) = h\text{-rank}(i) + j\text{-count}(i,h)$$

PROOF. *Note that $P(j,h+1) < P(i,h+1)$ if and only if one of the following conditions is true:*

- $P(j,h) < P(i,h)$
- $P(j,h) = P(i,h)$ *and* $P(j+2^h,h) < P(i+2^h,h)$

There are h-rank(i) values of j such that $P(j,h) < P(i,h)$. Thus, all that's left to prove is that $P(j,h) < P(i,h)$ if and only if h-rank$(j) < h$-rank(i), and $P(j,h) = P(i,h)$ if and only if h-rank$(j) = h$-rank(i). We will prove the only if direction of each case; together these imply the if directions via contrapositives.

If $P(j,h) = P(i,h)$, then $P(k,h) < P(j,h)$ if and only if $P(k,h) < P(i,h)$. Therefore, h-rank$(j) = h$-rank(i).

Next, consider the case where $P(j,h) < P(i,h)$. Let $L_i = \{k | P(k,h) < P(i,h)\}$. Note that $L_j \subset L_i$, since if $P(k,h) < P(j,h)$, $P(k,h) < P(i,h)$. Furthermore, $i \in L_j$, but $i \notin L_i$. Therefore, $|L_j| < |L_i|$, so h-rank$(j) < h$-rank(i). □

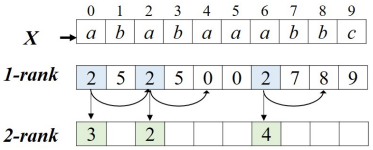


Figure 2: Illustration of Theorem 1: Refining ranks based on the doubling principle.

An example is given in Figure 2, in which the 1-ranks of certain suffixes are used to compute 2-ranks. The 1-ranks are obtained after iteration 1 and are based on $2^1 = 2$ first characters of each suffix. In order to calculate 2-rank(0), for example, we need to consider 1-ranks at three positions j: 0, 2 and 6, which are all equal to 1-rank(0). Of these values, 1-rank$(j+2^1)$ is equal to 2, 0, and 8 respectively. Only one of these values, 0, is less than 1-rank$(0+2)$=2. Thus, j-count$(0) = 1$. Therefore, 2-rank$(0) = 2+1 = 3$.

This theorem is at the heart of prefix doubling algorithms [21, 17]. We have adopted it to the concept of *h-ranks*. It follows that if the 0-*rank* of each suffix is known, we can find the 1-*rank* of each suffix after one iteration, the 2-*rank* of each suffix after two iterations, and the *i-rank* of each

[1]An astute reader may note that $j+2^h$ or $i+2^h$ may be greater than N, making the second condition undefined. However, this is unimportant, as the first condition, h-rank(i)=h-rank(j), will always be false in such cases.

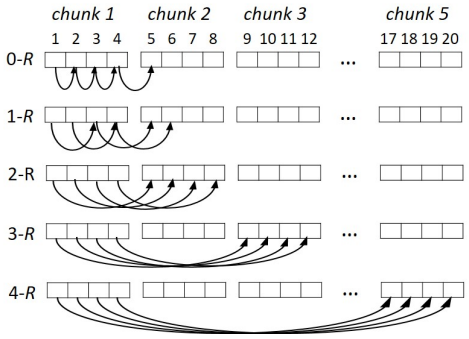


Figure 3: *Suffix Rank*: limiting working memory of the algorithm to at most two small chunks of the input.

suffix after i iterations. Because the i-*rank* corresponds to a prefix of length 2^i, the algorithm will find the final rank of each suffix in at most $\log N$ iterations.

From now on, for a suffix S_i on iteration h, we will refer to h-$rank(i)$ as its *coarse rank*, h-$rank(i+2^h)$ as its *fine rank*, and $(h+1)$-$rank(i)$ as its *updated rank*. The tuple consisting of the coarse and fine ranks of suffix S_i is called a *refined rank pair*. From Theorem 1, we can deduce the updated rank of S_i, which is the sum of its coarse rank and the number of refined rank pairs with the same coarse rank and a smaller fine rank.

This simple technique of summing up the counts of refined rank pairs in order to go from h-ranks to $(h+1)$-ranks allows one to determine the final rank of each suffix in $O(\log N)$ iterations. However the summation of refined rank pairs has to be performed over all coarse ranks of the entire input of size N – and in our problem setting, N is several times larger than the available main memory, so the summation cannot be performed in RAM. Next, we describe the steps of the disk-based *Suffix Rank* algorithm, which accomplishes this task effectively using disk as a memory extension.

4.2 Input partitioning

The input is divided into K chunks small enough for any two of them to be processed entirely in RAM. The chunk size must be a power of 2. The reason for that will become apparent shortly. For each chunk k, we maintain two arrays: h-R_k and h-SA_k. In h-R_k, we record the coarse h-rank of each suffix starting in chunk k. The second array, h-SA_k, stores the start positions of suffixes sorted by their coarse rank. Thus, suffixes with the same coarse rank will be adjacent to each other in h-SA_k.

To produce refined rank pairs in iteration h, we need to have access to both the coarse and fine rank of a suffix at the same time. These ranks are 2^h positions apart and may reside in different chunks. We load at most two contiguous pieces of data into main memory: one containing the coarse ranks and another containing the fine ranks. Because of the doubling nature of the prefix, and as long as the chunk length is set to a power of 2, the fine ranks 2^h positions away will always be located inside exactly one additional chunk. At each iteration h, we look at position i to collect the coarse rank of S_i, and at position $i+2^h$ to collect its corresponding fine rank. If we denote the length of each chunk by x, then all the fine ranks for chunk k which are not in chunk k itself are located in chunk $k+1+\lfloor \frac{2^h}{x+1} \rfloor$.

An illustration is given in Figure 3. Each chunk has length $x=4$. Consider chunk 1. In iterations $0, 1, 2$, we only need chunk 1 and chunk 2. In iteration 3, we need chunk 1 and chunk 3. And in iteration 4, the additional chunk of interest is chunk 5. Thus, access to only two small chunks at a time is required to generate the refined rank pairs in each iteration, and we make each chunk small enough for any two of them to fit entirely in main memory.

Since we collect refined rank pairs for each separate chunk of input in turn, it is impossible to immediately produce updated ranks by the method described in Theorem 1. While the refined rank pairs for the current chunk k are in main memory, we can compare any two of them; we know if their updated ranks will be less than, greater than, or equal to each other. However, we cannot know what those updated values will be without information from other chunks. Thus, while SA_k can be updated using only local information, updating R_k requires global information from all chunks. We get this information in the *resolve* step, and use it to calculate updated ranks. In the *update* step, these updated ranks are communicated back to their corresponding chunks.

After partitioning, the algorithm performs an **Initialization** of 0-*ranks* followed by at most $\log N$ iterations. Each iteration h consists of the same three steps:

1. Refine. We collect unique refined rank pairs and their counts from the corresponding pairs of chunks and generate refined rank pairs for each chunk. We sort the pairs, and write them to disk.

2. Resolve. We merge sorted pairs from all chunks using buffering techniques of the merge step of $2PMMS$. During the merge, we compute the total count of each unique refined pair across the entire input, and from these counts we deduce the updated rank of each suffix across the entire input string.

3. Update. We update ranks in each chunk.

From the memory point of view, the algorithm consists of sequential loads of constant-size chunks of input into the main memory, and a series of buffered merges.

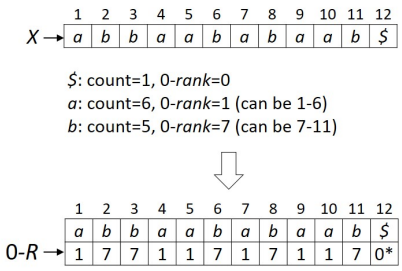
Next, we explain each step in detail. Our explanation is accompanied by the running example of ranking suffixes in the input string $X = abbaababaab$ with chunk size $x = 4$.

4.3 Initialization

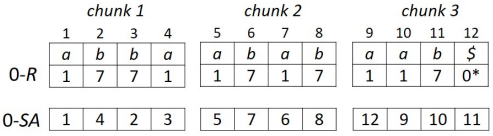
In this step, we set the initial 0-*rank* of each suffix. It can be computed efficiently by scanning the input string and counting the number of times each distinct character appears. The initial rank of suffix S_i is deduced from the total number of characters lexicographically preceding $X[i]$. After the character counting is completed, we never access the input string again – we work with the ranks in each chunk instead. Thus, the input string does not need to be randomly accessed and can reside on disk.

Before we continue, we introduce the concept of a *resolved rank*. Note that when a rank is unique across the entire input, it will not be updated again. We mark such ranks as *resolved* [2]. In the initialization step, only suffixes starting with a unique character are marked as resolved. Other suffixes are resolved during the *resolve* step, when we can tell whether an updated rank is unique across the entire input. In subsequent iterations, we skip over resolved ranks. The algorithm stops when all the ranks are resolved.

[2]We mark resolved suffixes with a $\star$ in our figures, and using a sign bit for each rank in our implementation.



(a) Counting total occurrences of each character to assign initial (coarse) ranks. Sentinel \$ is unique and thus suffix S_{12} is marked as *resolved* (*).



(b) For each chunk k of the input we produce 0-R_k rank array and 0-SA_k suffix array.

Figure 4: *Suffix Rank*: initialization.

An example of the initialization step is presented in Figure 4a. Character \$ is the smallest, so the corresponding suffix S_{12} gets its final rank 0. There are 6 suffixes starting with a. Thus, these suffixes will have final ranks $1-6$, but at this point (based on the first character) we assign a coarse rank of 1 to each of them. Suffixes starting with b will have ranks $7-11$, so their initial coarse rank becomes 7.

Based on the initial ranks, we create two arrays for each chunk k: rank array 0-R_k, and the suffix array 0-SA_k. The suffix array reflects the current order of suffixes within chunk k based on their first character. Guided by the order in h-SA_k, at each subsequent step h, the algorithm produces refined rank pairs and their counts for each chunk.

We show an example of these two initial arrays for each chunk in Figure 4b. Consider 0-SA for chunk 2. Two suffixes S_5 and S_7 occupy consecutive positions in this suffix array, and their coarse ranks are the same. In order to produce the refined rank pairs we only need to sort the fine ranks at positions $5+2^0=6$ and $7+2^0=8$ in the rank array 0-R. Using local suffix arrays eliminates the need to sort refined rank pairs across the entire chunk.

We can now perform $O(\log N)$ iterations, each consisting of *refine*, *resolve*, and *update* steps.

4.4 Step 1. Refine

This step collects refined rank pairs by reading coarse and fine ranks from at most two chunks. It generates triples *(coarse, fine, count)*, where *count* refers to the total local number of suffixes within the chunk k such that h-rank$(i) = coarse$ and h-rank$(i+2^h)=fine$. The values of coarse rank are collected in the order specified by the current h-SA. Resolved ranks are ignored. When a new unresolved coarse rank is reached at position i, we collect all suffixes with this rank along with the corresponding fine ranks. This collecting continues until the coarse rank changes. Then, using ternary quicksort [4], we rearrange the corresponding part of the chunk suffix array, using $fine$ ranks as the key. This part of SA now reflects the order based on the first 2^{h+1} characters of each suffix: at the end of *refine*, h-SA_k becomes $(h+1)$-SA_k. Any suffixes with the same refined rank pair will be next to each other in this updated SA. Triples (*coarse, fine, count*) can be created easily, because the refined rank pairs are now sorted. After each triple is created, it is added to an output buffer. Note that all the triples in this buffer are sorted by $(coarse, fine)$. Also note that the suffix array now contains the order of updated ranks for chunk k, even though the values of those updated ranks across the entire input are currently unknown.

The sorting of triples is performed largely like in *qsufsort* [17]. The difference is that instead of updating ranks, the sorted triples are written to disk. At the end of this step, we have on disk sorted runs of such triples for each chunk.

Figure 5 [1] demonstrates this step for iteration 0. After initialization, the ranks were based on a single character. The goal is to find a refined rank based on the two first characters of each suffix. For this, we look at the fine ranks 2^0 characters apart from the original rank. For example, in chunk 1 there is only one refined rank pair $(7,7)$ for suffix S_2, so the triple $(7,7,1)$ is generated and added to the disk run 1. Note that even though this refined rank pair is unique for chunk 1, we cannot conclude that the 1-rank of S_2 is resolved, before we aggregate the counts from all different chunks. In chunk 2, there are two identical refined rank pairs $(1,7)$ which correspond to suffixes S_5 and S_7. Thus the triple $(1,7,2)$ is generated and added to the disk run 2.

4.5 Step 2. Resolve

This step is the core of the *Suffix Rank* algorithm. We must determine which new rank each refined rank pair updates to. We calculate this using a buffered merge of sorted triples reminiscent to that of $2PMMS$.

For each chunk, an input and output buffer is created. From the previous step, we have a file of sorted triples (*coarse, fine, count*) from each chunk. Each input buffer is filled with the smallest triples from that chunk. A binary heap is initialized with the smallest triple from each buffer. The triple at the head of the heap is the smallest (or tied for the smallest) refined rank pair overall. We then process the triple at the head of the heap. After a triple is processed, the next triple from the same buffer is inserted into the heap. When we reach the end of any buffer, that buffer is refilled with the next smallest triples from the corresponding chunk. This way, we collect the total count for a given refined rank pair across the entire input.

We now describe how to deduce the updated rank of a given triple from this count. Recall that by Theorem 1, the updated rank of a suffix is equal to the sum of its coarse rank and the number of suffixes with the same coarse rank but lesser fine rank. There are the following cases to consider:

(1). If the coarse rank in a given triple is encountered for the first time, then the updated rank remains the same as the coarse rank for all identical refined pairs.

(2). If the coarse rank remains the same but the fine rank changes, then the updated rank is equal the sum of the coarse rank and the total count of refined rank pairs with the same coarse rank and lesser fine rank.

(3). If in any case, the overall count for a given refined pair is equal to 1, then the updated rank determined as described above becomes the final resolved rank of the suffix.

After an updated rank is computed, it is written to the output buffer of the corresponding chunk. When an output

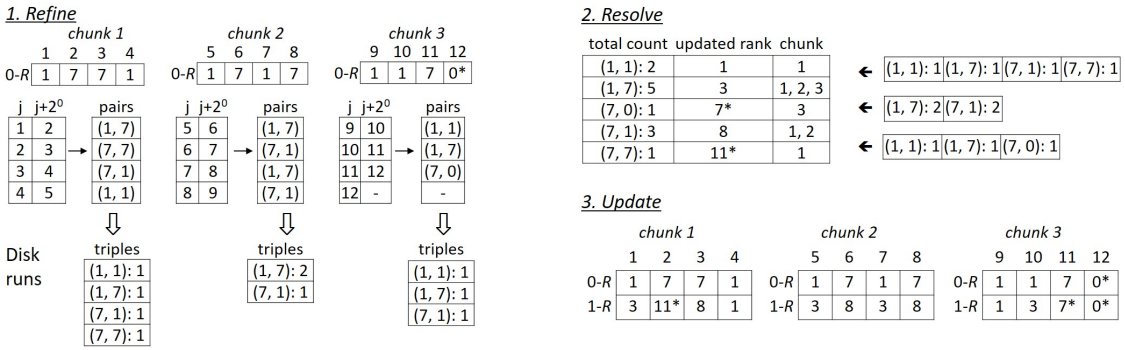


Figure 5: *Suffix Rank*: sample iteration. [1] Refined pairs for each chunk are generated in memory. Sorted pairs with their counts are written to disk. [2] The counts of unique pairs are summed up during the global merge. New ranks are deduced from these counts. Updates for each chunk are written to disk. [3] The updated ranks are delivered to each chunk.

buffer is full, its content is appended to the update file for this chunk. At the end of this step, we have the updated ranks for each chunk as a set of disk files.

Figure 5 [2] demonstrates the *resolve* step for iteration 0. We merge sorted pairs from the three chunks, and determine their global counts. Since there are two suffixes with refined rank pair $(1, 1)$, the final rank for these suffixes cannot yet be determined. Because for the coarse rank 1, there are no fine ranks smaller than 1, the updated rank remains 1. The rank of suffixes with refined rank pair $(1, 7)$ updates to 3, because from the count of $(1, 1)$ we deduce that there are precisely two suffixes with rank 1 which are smaller than $(1, 7)$. The global count for refined rank pair $(7, 0)$ is 1, meaning this rank is unique and is marked as resolved. The same is true for the pair $(7, 7)$. The updated ranks are reported back to their respective chunks in the form of small update files used in the next step.

4.6 Step 3. Update

For each chunk, we have a file containing its updated ranks in sorted order. Recall that in Step 1, SA was modified so that it reflects the order of suffixes in this chunk according to their updated rank. We scan the refined rank pairs in chunk k in the order specified by SA. We replace each previously unresolved rank with an updated rank. Once all ranks have been updated, we write the h-R_k array back to disk. If, at the end of this step, all ranks in all chunks are resolved, we are done. Otherwise, we perform another iteration of the algorithm, starting with the *refine* step.

The result of the *update* step for iteration 0 is shown in Figure 5 [3]. The ranks are now based on the two first characters of each suffix. In the remaining iterations, the order will be refined based on $4, 8, \ldots 2^{\log N}$ characters until all the ranks are resolved. Because at each new iteration h, the rank precision is based on a prefix twice the size of the previous iteration h-1, the algorithm is guaranteed to terminate in at most $\log N$ iterations.

After each suffix receives its final rank, we can construct the *Suffix Array* of string X by inverting the ranks array in a single sequential scan using buffered bucketing techniques.

4.7 Analysis

Suffix Rank performs at most $\log N$ iterations, and each iteration includes three steps described above.

Algorithm 1 *Suffix rank*: main loop. READ and WRITE refer to disk I/Os

```
1:  procedure SuffixRank(input string X of size N, M)
2:      READ X using buffer of size M
3:      count all distinct letters
4:      produce initial letter ranks
5:      break X into K chunks of size N/2M each

6:      for j ← 1, K do
7:          READ chunk k
8:          create array 0-R_k
9:          fill 0-R_k with initial ranks
10:         compute suffix array 0-SA_k
11:         WRITE 0-R_k to rank file R_k
12:         WRITE 0-SA_k to SA file SA_k
13:     end for
14:     h ← 0
15:     all_resolved ← False

16:     while all_resolved ≠ True do
17:         all_resolved ← True
```

Algorithms 1, 2, 3, and 4 show the pseudocode for the entire *Suffix rank* algorithm from the point of view of disk I/Os. The pseudocode uses notations in accordance with the definitions in Sections 4.1 and 4.2. Thus, h refers to the iteration number, which works with prefixes of size 2^h. The chunks are numbered $1, \ldots, K$. Notation i-R_k refers to the in-memory rank array for chunk k in iteration i, and R_k to the file of ranks stored on disk. Similarly, a suffix array for chunk k is represented by i-SA_k, and the suffix array file by SA_k. READ and WRITE commands represent disk I/Os.

The procedure SuffixRank on line 1 takes as an input string X and the size of the main memory M. The output is the final rank array of string X on disk. Lines 2–13 correspond to the *initialization* step. After initialization, the algorithm performs a single while loop (lines 16–47). The while loop continues as long as there is at least one unresolved rank.

Inside the while loop, three steps are repeated: *refine*, *resolve*, and *update*. The *refine* step is presented on lines 18–32 in Algorithm 2. In this step, for each chunk k of the input, we load into memory two rank arrays, R_k and R_{next}, and the suffix array SA_k (lines 19–22). Then, we use these three arrays to generate refined rank pairs and their counts for each chunk in turn. A more detailed view of this in-memory algo-

Algorithm 2 *Suffix rank*: refine

```
18:     for i ← 1, k do
19:         READ file R_k into h-R_k
20:         next ← k + 1 + ⌊2^h / (x+1)⌋
21:         READ file R_next into h-R_next
22:         READ file SA_k into h-SA_k
23:         T ← RefineFunc(h, h-R_k, h-R_next, h-SA_k)
24:         WRITE updated (h+1)-SA_k to file SA_k
25:         if len(T) > 0 then
26:             all_resolved ← False
27:         end if
28:         WRITE triples to sorted run k
29:     end for
30:     if all_resolved = True then
31:         return
32:     end if
```

Algorithm 3 *Suffix rank*: resolve

```
33:     do buffered multi-way merge of sorted runs
34:       during the merge:
35:       count totals for each (coarse, fine) pair
36:       deduce updated rank from these counts
37:       mark any rank with count total 1 as resolved
38:     WRITE updated to update files using buffers
```

rithm is presented in function RefineFunc (Algorithm 5). The function generates sorted triples $(coarse, fine, count)$ for a given chunk k.

The *resolve* step is described on lines 33–38 in Algorithm 3. This step is essentially a merge step of the well-known disk-friendly $2PMMS$. During the merge, we obtain the total count for each distinct refined rank pair and deduce the new updated rank of each suffix from these counts,following the logic described in Section 4.5. The updated ranks in sorted order are written to the update file for chunk k.

Finally, the *update* step is shown on lines 39–45 of Algorithm 4. For each chunk k, the contents of its current rank file, the current SA file, and the update file are loaded in main memory. The i-ranks are updated into $(i+1)$-ranks, guided by the order in the corresponding suffix array, and written back to disk to be consumed in the next iteration.

The loop terminates when all the ranks are resolved. This is guaranteed to happen after at most $\log N$ iterations of the loop.

In Algorithm 5, we present a more detailed pseudocode of the RefineFunc which is performed exclusively in RAM. It produces refined rank pairs for a given chunk. At each iteration, for a given chunk k, the order recorded in the suffix array SA is based on the current *coarse* rank of each suffix. We iterate over all the positions recorded in the suffix array, and find an interval $(start : end)$ corresponding to the same *coarse* rank. We then sort the refined rank pairs corresponding to this interval by $fine$ rank found in R_{next}. Next, the order in the corresponding section of the SA is updated accordingly. The h-SA becomes $(h+1)$-SA and is now ready for the next iteration. The list of sorted triples is returned to line 23 of Algorithm 4.4 and both the updated suffix array and the sorted triples are written to disk.

Based on the pseudocode in Algorithms 1 – 4 we can analyze the worst-case performance of *Suffix Rank* in the Disk Access Model of computation (DAM). In this model, B refers to the block size, and the complexity of the algorithm is expressed in the number of disk block I/Os.

Algorithm 4 *Suffix rank*: update

```
39:     for i ← 1, k do
40:         READ file R_k into h-R_k
41:         READ update file for chunk k
42:         READ file SA_k into (h+1)-SA_k
43:         update h-R_k to (h+1)-R_k
44:         WRITE (h+1)-R_k to rank file R_k
45:     end for
46:     h ← h + 1
47:   end while
48: end procedure
```

Algorithm 5 RefineFunc: refines ranks in a separate chunk (In RAM).

```
1:  function RefineFunc(h,R, R_next, SA)
2:      triples ← empty_buffer
3:      SA position j ← 1
4:      start ← j
5:      while j <= length(h-SA) do
6:          if R[SA[j]] ≠ resolved then
7:              while R[SA[j]] stays the same do
8:                  end ← j
9:                  j ← j + 1
10:             end while
11:             using R and R_next:
12:                 find fine ranks for SA[start : end]
13:             sort by fine ranks
14:             update order in SA[start : end]
15:             produce count for each (coarse, fine)
16:             append (coarse, fine, count) to triples
17:             start ← j
18:         else
19:             SA position j ← j + 1
20:         end if
21:     end while
22:     return triples
23: end function
```

In the initialization step, the entire input string of size N is read in one sequential scan (line 2), and the initial rank arrays for each chunk of total size $O(N)$ are written to disk (lines 11–12). That accounts to $O(N/B)$ block disk I/Os. Next, we count I/O operations inside the while loop. We have in total $K = O(\frac{N}{M})$ chunks. In the *refine* step, for each chunk k, three large files of total size $O(\frac{N}{K})$ are read from disk in a sequential scan and processed in memory. Then the local $(coarse, fine, count)$ triples are written to disk. The total number of such triples in the current chunk is at most $\frac{N}{K}$ (size of the chunk). All the reads and writes are sequential and this gives an upper bound of $O(\frac{N}{KB})$ disk block transfers for each chunk. Multiplying by the total number of chunks, K, gives $O(\frac{N}{B})$ total disk block I/Os. By the same logic, the total amount of work in the *update* step is also $O(\frac{N}{B})$ blocks.

The *resolve* step has the same I/O complexity as *2-Phase Multi-way Merge Sort*, which is $O(\frac{N}{B})$: the inputs are read in blocks, and the outputs are written in blocks using an in-memory input and output buffer for each chunk. This complexity is guaranteed as long as $N < \frac{M^2}{RB}$, where R is the size of a triple. As an example, the size of the triple in our implementation is 20 bytes (two 8-byte integers to represent a pair of ranks, and a 4-byte integer for local counts). With a block size $B = 4$ KB and memory as low as 1 GB, the assumption holds for input sizes of up to about 12 TB. Thus, with the above assumption, the total complexity of the algorithm in the DAM model is $O(\frac{N}{B})$ block I/Os per

iteration. With $O(\log N)$ iterations, this gives a total of $O(\frac{N}{B} * \log N)$ block I/Os.

Rather than just big-O analysis, let us now discuss the constants. From the pseudocode in Algorithms 2 – 4 it is easy to compute the total number of disk reads/writes per iteration. The algorithm requires us to read $3(\frac{N}{B})$ blocks in the *refine* step, and to write $2\frac{N}{B}$ blocks once. In the *update* step we read $3(\frac{N}{B})$ and write $\frac{N}{B}$ blocks. In the *resolve* step – which corresponds to the merge phase of $2PMMS$ – we have at most one read and one write of $\frac{N}{B}$ blocks, assuming that the input-to-memory ratio allows us to do it in a single pass (see e.g. [9]). This results in 7 full scans (reads) of N/B disk blocks, and 4 writes per iteration. The $11(N/B)\log N$ total I/O operations represent just an upper bound: the number of remaining unresolved ranks decreases in each iteration, and most of the reads and writes work with significantly less than $\frac{N}{B}$ blocks.

We now analyze the space complexity of our algorithm. The algorithm runs in a constant amount of memory M, and uses an overall linear disk space. The disk is used to store several intermediate data structures: rank arrays, suffix arrays, sorted triples runs, and the update files. The total size of rank arrays is always N, and the size of the rest is $O(N)$. The elements of a rank array are 8-byte integers, the elements of a suffix array are 4-byte integers, the elements of a sorted run are triples of size 20 bytes each (two longs and one int), and the update file contains updated ranks of size 8 bytes each. Thus, the upper bound on the temporary disk space is $40N$ bytes. In our settings, we treat cheap disk space as an unbounded memory extension. In practice, this upper bound on peak disk usage is never achieved, as per the experimental results presented in Figure 9b.

To summarize, our algorithm runs in linearithmic time in DAM model, and uses linear disk space. *Suffix Rank* is a truly external-memory algorithm because it reads and writes disk data sequentially. It must be noted that the actual number of iterations is bounded by the logarithm of the length of the maximum $|LCP|$. If the prefix shared by any two suffixes is smaller or equal than $max|LCP|$, then all the suffixes become resolved after $\log max|LCP|$ iterations.

Table 1: Datasets used in the experiments.

Dataset	Size N	Alph. Σ	# of strings	Max. string
WIKI [43]	76 GB	96	1	$7.5 * 10^{10}$
PROT [42]	59 GB	25	1	$5.9 * 10^{9}$
SKY [5]	30 GB	$\log N$	1	$3.0 * 10^{9}$
GUT [34]	20 GB	105	55,000	$1.5 * 10^{8}$
DNA [39]	20 GB	5	6	$7.5 * 10^{9}$

5. EXPERIMENTS

To assess the performance of *Suffix Rank* for solving Problem 1, we present experiments that compare it to alternative solutions (*SAscan* [13], *eSAIS* [5], and *eGSA* [20]) on a variety of datasets and across a range of hardware.

The prototype[3] of *Suffix Rank* is implemented as a suit of small modular *C*-**subprograms**. Each subprogram is responsible for a single step: *refine*, *resolve*, or *update*. The subprograms are invoked in turn during each iteration. Since the input does not fit into the RAM, each subprogram processes a single independent chunk of the input at a time. Once the current subprogram is done with the chunk assigned to it, it writes the output to disk. The next subprogram will consume the file on disk as its input. The subprograms are invoked in the main iteration loop implemented in a bash script. This simple modular program design is chosen with the future parallel version in mind: we want each single small subprogram to be able to process an independent chunk of input on a separate cluster node.

[3]The current version is available at `https://github.com/mgbarsky/SUFFIX_RANK`.

Setup. We use the following three hardware platforms to run the tests:

Platform D. Desktop computer equipped with a 4-core 3.40 GHz Intel i7-2600 CPU with 1 MiB L2 cache and 8 GiB of DDR3 RAM. The machine uses a single SATA hard drive with a total capacity of 5,589 GiB, 256 MB of on-board cache, and 7,200 RPM speed. The OS is Linux (Ubuntu 20.04).

Platform C. Cloud platform with 3.5 GHz Intel Xeon-Haswell Quadcore processor, 8 GB of DDR3 RAM, and a hard drive with a total capacity of 6 GB, speed of 7200RPM, and 256MB on-board cache. The OS is Linux (Ubuntu 18.04).

Platform S. Scientific platform which includes dedicated Lenovo SD530 servers with 40 Intel "Skylake" at 2.4 GHz processors and 188 GiB of RAM, with access to 10 TB of **fast, high performance shared file system made of SDDs**, running CentOS Linux 7.

Input datasets. The main characteristics of the datasets used in the experiments are given in Table 1.

WIKI. Single XML file containing a snapshot of all English wikipedia articles from 02/02/2020. Data type: natural language text.

PROT. Single file generated from the entire *TrEMBL* database of protein sequences after cleaning it from all additional entries, leaving only sequences of actual proteins. Data type: aminoacid sequences.

SKY. Artificially generated string (SKYLINE) which was introduced in [5] as an adversarial input for the *eSAIS* algorithm. SKYLINE string is recursively defined by the grammar $\{T_k \rightarrow T_{k-1}"k"T_{k-1}, T_1 \rightarrow 1\}$. For example, $T_1 =$ "1", $T_2 = T_1"2"T_1 =$ "121", and $T_3 = T_2"3"T_2 =$ "1213121". If you plot the numbers, it resembles a skyline. Sorting suffixes of a SKYLINE string is not a trivial task: there are identical prefixes at each level of recursion.

DNA. Short DNA raw reads from sequencing of six Human genomes. Data type: DNA sequences.

GUT. English book texts harvested from the Gutenberg project website. This natural language dataset is anomalous, because it contains several versions of certain books, and each file is prefixed with a long identical copyright notice.

Programs. We used the following implementations of competing algorithms:

SAscan. The current version *SAscan 0.1.1* [38] requires the installation of the *libdivsufsort* library [40].

eSAIS. The implementation [37] uses the Standard Template Library for Extra Large Data Sets (STXXL) [41]. The program performs asynchronous I/Os in parallel threads,

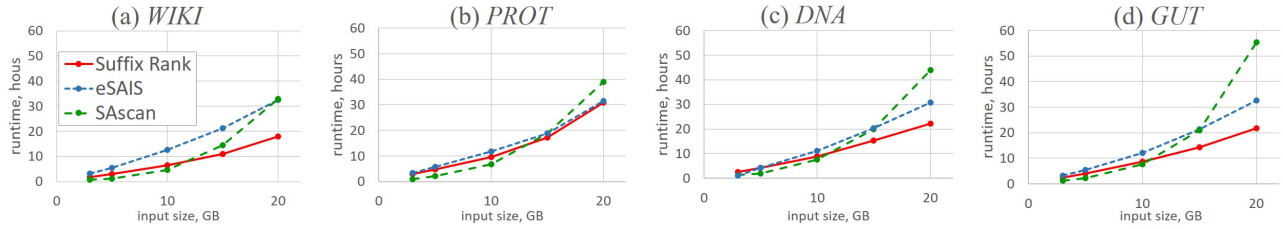
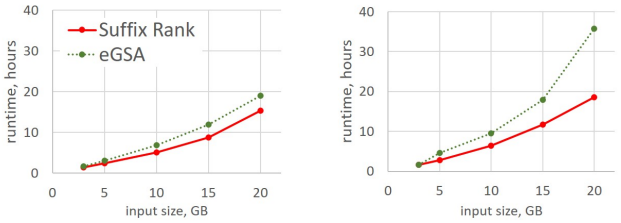


Figure 6: Comparative performance of *Suffix Rank* vs. *SAscan* and *SAIS* algorithms. Main memory used: 1 GB.



(a) *DNA-L* **(b)** *PROT-L*

Figure 7: Comparative performance of *Suffix Rank* vs. *eGSA* for building a Generalized Suffix Array.

implemented in STXXL. In addition, *eSAIS* uses a compressed representation of suffix start positions - 40 bits vs. 64 bits used by *Suffix Rank*. We run *eSAIS* in a mode which only generates a suffix array.

eGSA. The implementation [36] treats each line in the input file as a single string and builds a generalized suffix array for all such strings.

Note that our program is written in bare C, and as such it does not require any installation or setup. In contrast, both *SAscan* and *eSAIS* implementations rely on sophisticated external libraries.

Next we present the experimental results. We compare the performance and scalability of the *Suffix Rank* to other algorithms in Section 5.1, present large-scale experiments in Section 5.2, and conclude with the investigation of some properties of our algorithm in Section 5.3.

5.1 Comparative performance

In this section we show the performance results of our algorithm vs. all the other algorithms listed above. We run these experiments on Platform D.

From datasets in Table 2 we generate inputs of 5, 10, 15, and 20 GB in length by taking a prefix of the corresponding input string or a subset of the multi-file input. We constrain the memory available to each program to 1 GB. This corresponds to the increase in the input-to-memory ratios $K=\frac{N}{M}$ from 5 to 20. This technique of simulating out-of-core inputs is commonly used (see e.g. [22, 32, 13, 20]), and allows to explore much larger values of K than it would be possible would we use 16-32 GB of RAM limit. We are most interested in exploring the scalability of the algorithms for inputs much larger than the available main memory.

Suffix Arrays for a single string. Here we test the scalability of our algorithm against *SAscan* and *eSAIS*, which both treat the input as a single string. Note that in these experiments we constrained *eSAIS* to a single-thread, to make it comparable with *SAscan* and *Suffix Rank* which both run in a single thread.

The results are presented in Figure 6. When K is less than 10, *SAscan* algorithm shows an outstanding performance. It is also very lightweight, and uses the least temporary disk space. As K grows, however, the performance of *SAscan* significantly degrades, because the algorithm scales quadratically when both N and K are increasing. When K reaches 20, *SAscan* is significantly outperformed by *eSAIS* (even in its slower single-threaded version). Thus, for even larger K, *eSAIS* is the most promising out of the two.

Generalized Suffix Array. Neither *SAscan* nor *eSAIS* can build a generalized suffix array for a large set of small input strings. Out of three algorithms used in our experiments, the *eGSA* [20] is the only one which is designed to handle this task. Next, we compare the performance of our algorithm to *eGSA*.

The *eGSA* takes a single file as an input, but processes each line in this file as if it was a separate string. Thus, it is not applicable for the *WIKI* dataset, which we treat as a single input string. The *eGSA* also cannot handle cases when a single string in the collection is very large. We modified the *GUT* dataset, converting each file into a separate line, but we were unable to produce the results for *eGSA*: it took more than 18 hours on a 5 GB input, and at this point consumed twice the amount of the allocated memory.

By default, our implementation of *Suffix Rank* takes as an input a folder name, and each file in this folder is treated as a separate string. For this experiment, we wrote a special input handling module in which we treat each line in a file as a separate string. As described in 4.1, we assign initial unique resolved ranks in place of sentinel characters appended to the end of each line.

For this experiment, we treat the inputs from Table 1 differently:

DNA. Each file of this dataset contains multiple lines, and each line corresponds to a separate raw read obtained from a sequencing machine. In ***DNA-L*** we treat each such line as a separate input string.

PROT. Each line of this dataset contains a separate protein sequence. In ***PROT-L*** we treat each line as a separate input string.

The results in Figure 7 show that *Suffix Rank* builds a generalized suffix array faster than *eGSA* for all input sizes generated from these two multi-string collections. The difference in time is increasing with larger values of K. We therefore do not consider this algorithm for experiments on larger inputs.

From the results presented in this section we conclude that the only viable competitor to *Suffix Rank* for large-scale experiments is the *eSAIS* algorithm.

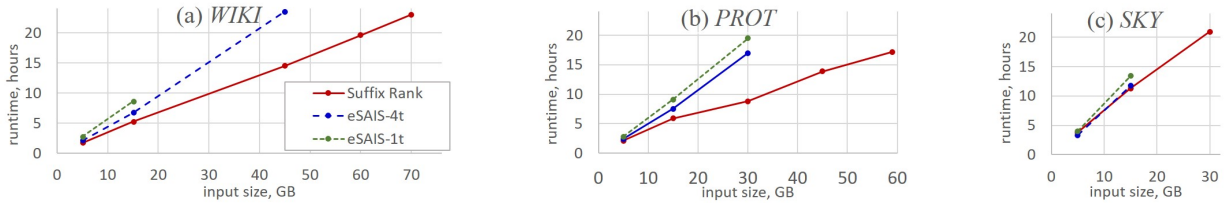


Figure 8: Large-scale experiments performed on Platform S (with fast SSD storage).

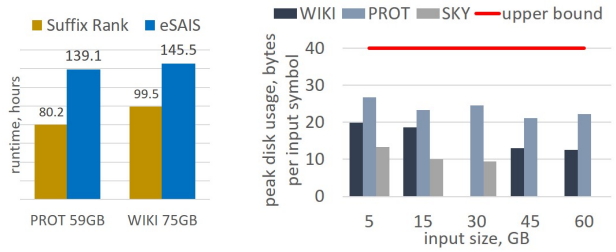


Figure 9: (a) Runtime (hours) of *Suffix Rank* vs. *eSAIS* on Platform C for two large inputs. (b) Peak disk usage measured in large-scale experiments compared to a theoretical upper bound $40N$.

5.2 Large-scale experiments

In this section, we present results of a performance evaluation for *Suffix Rank* vs. *eSAIS* (the most-scalable competitor identified in the previous section). We use three large datasets from Table 1: the full protein sequence database *PROT* (59 GB), snapshot of *WIKI* articles (75 GB), and the adversarial *SKY* dataset (30 GB). All the experiments are conducted using main memory constrained to 1 GB, with the largest input-to-memory ratio K being 75.

In Figure 9a we show the running time of our algorithm vs. the multi-threaded version of *eSAIS* on Platform C (which uses HDD, similarly to Platform D). Our algorithm has performance superior to *eSAIS* in these settings. It runs faster than *eSAIS* by more than 30% for the *WIKI*, and by 42% for the *PROT* dataset.

In Figure 8 we present the results of large-scale experiments on Platform S (equipped with high-performance SSD disks). We run two versions of *eSAIS*, using a single thread (*eSAIS-1T*), and using 4 threads (*eSAIS-4T*). Note that the experiments on this platform have a timeout of 24 hours. A missing point in a graph for a larger input size is an indication that the timeout was reached (it happens for *eSAIS-1T* and *eSAIS-4T*, never for *Suffix Rank*). Comparing the results for the same datasets on two different platforms, we see that *Suffix Rank* performance improves with faster memory type, where it completes indexing of a 59 GB string (PROT) in just 17 hours vs. 80 hours on platform C, and indexing of a 70 GB string (WIKI) in 23 hours vs. 100 hours on Platform C.

On Platform S, our algorithm performs significantly better than both *eSAIS-1T* and *eSAIS-4T* - even for smaller values of K. This can be explained by the fact that our algorithm has a higher I/O volume comparing to *eSAIS*, and benefits from fast secondary memory more than the latter. *Suffix Rank* is about 40% faster than *eSAIS-4T* for both 30 GB of PROT string and for 45 GB of WIKI string, and the gap between the running time of the two algorithms is increasing as we scale the input further.

The results in Figure 8(c) show that *SKY* dataset – designed as an adversarial input for *eSAIS* – is also adversarial for *Suffix Rank*. For *SKY* string of length 5 GB our algorithm runs slower than *eSAIS*. To process 15 GB *SKY* string it takes *Suffix Rank* 11 hours as compared to 6 hours with PROT string and 5 hours with WIKI string of the same length.

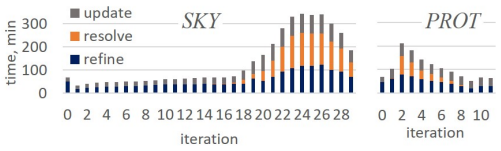
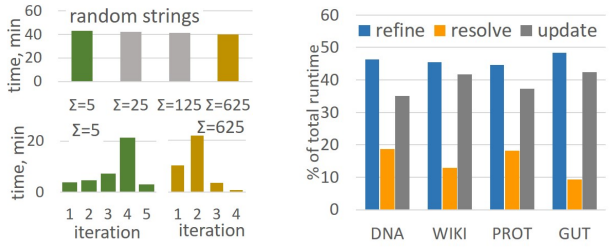


Figure 10: *Suffix Rank* performance profiles for *SKY* and *PROT*. X-axis: iteration number. Y-axis: time taken by different steps on this iteration (min). Input sizes: 20 GB. Platform: D.

In [5] the adversarial nature of *SKY* for *eSAIS* is explained by the need to perform multiple recursive steps. Our algorithm exhibits different dynamics on *SKY* inputs. To investigate this dynamics, we present the performance profiles for a real-life dataset *PROT* and the adversarial *SKY* dataset in Figure 10. *PROT* exemplifies a normal distribution of processing time for all real-life datasets used in the experiments. In early iterations, the number of distinct pairs to be merged is relatively small. This number grows exponentially in terms of alphabet size, so within a few iterations, there are many distinct pairs. However, most suffixes are resolved after iteration 7. Thus, later iterations only involve a handful of unresolved pairs. The *SKY* input completely changes this normal dynamics. For *SKY* string, there are very few unique distinct pairs at the low level of the recursion, none of them are resolved, and the number of distinct pairs grows until the very last iterations. All unresolved pairs are carried over to the next iteration, and this increases the amount of work in all three steps, especially during the *resolve* step. This explains the adversarial nature of *SKY* for *Suffix rank*.

The performance experiments presented in this section clearly show that for large out-of-core strings our algorithm outperforms *eSAIS* – the only one algorithm among all the alternatives considered in this paper for solving Problem 1 that continues to scale with input size.

Finally, in Figure 9b we present the values of peak disk usage during the experiments on Platform S. As analyzed in Section 4.7, the upper bound of temporary disk space required by the algorithm is $40N$. The results in Figure



(a) **(b)**

Figure 11: (a). [Top]. Running time on randomly generated strings (of length 2GB each) with increasing alphabet sizes. [Bottom]. Illustration of work per each iteration for $\Sigma = 5$ and $\Sigma = 625$. (b). Percentage of time spent by each subprogram for different datasets. The most expensive are *refine* and *update* steps.

9b show that in practice we never reach this upper bound, exhibiting factors ranging from $10N$ to $27N$.

5.3 Suffix Rank properties

Finally, we run several experiments to better understand the nature of our algorithm.

Alphabet size. First, we study how the performance depends on the alphabet size $|\Sigma|$ of the input. One might expect that the larger the alphabet, the lower the length and frequency of repetitive substrings, and this should decrease the running time. However, as the results in Figure 11a show, $|\Sigma|$ plays little role in determining the performance of our algorithm. For randomly generated strings, the running time marginally decreases as Σ increases. While the total number of iterations decreases as the alphabet size grows, the amount of work per iteration increases: even in the first iteration, when prefixes of length 2 are compared, the total number of different prefixes will be $|\Sigma|^2$, and for large $|\Sigma|$ this requires more rank pairs to be generated during the *refine* step and merged during the *resolve* step of the algorithm (See performance profiles in Figure 11a [Bottom]).

For real-life strings, there is no clear dependence on alphabet size. The algorithm runs slower on the *WIKI* dataset with alphabet size 96, than on the *DNA* dataset with alphabet size 5.

Table 2: Datasets and their statistics. Datasets are arranged in order of increasing processing time (measured on inputs of 3GB in length).

Dataset	**Max.** $\|LCP\|$	(a) log **Max.** $\|LCP\|$	(b) **Avg.** $\|LCP\|$	$p = a * b$	$\log p$	time, sec/MB
DNA	$5.1 * 10^2$	9	$3.7 * 10^1$	$3.3 * 10^2$	2.52	2.15
WIKI	$1.6 * 10^4$	14	$3.4 * 10^1$	$4.8 * 10^2$	2.67	3.68
PROT	$4.0 * 10^3$	12	$4.7 * 10^1$	$5.7 * 10^2$	2.76	3.98
GUT	$4.2 * 10^6$	22	$3.5 * 10^3$	$1.9 * 10^3$	5.30	5.23
SKY	$1.1 * 10^9$	30	$1.4 * 10^4$	$3.6 * 10^8$	10.02	7.45

Longest Common Prefix. We now examine which properties of the input (apart from the alphabet) influence the performance of *Suffix Rank*. We present some potentially relevant statistics for our datasets in Table 2.

The $max|LCP|$ is defined as the length of the longest prefix shared by a pair of suffixes. The total number of iterations equals $\log max|LCP|$. The $average|LCP|$, introduced in [13], is the sum of the lengths of all longest common prefixes between each pair of consecutive sorted suffixes, divided by the total number of suffixes (N). It represents a rough measure of the difficulty of sorting the suffixes: if the $average|LCP|$ is large, then we need – in principle – to examine more characters in order to resolve the relative order of two suffixes. For our algorithm, it means that such suffixes are carried over from one iteration to another, increasing the amount of work per iteration.

The datasets in Table 2 are arranged in the increasing order of time which we need to process them (sec/MB of input). Neither the $max|LCP|$ nor $avg|LCP|$ alone aligns with the increase in the running time. However, if we multiply the total number of iterations ($\log$ max $|LCP|$) by the approximate average work required at each iteration (average $|LCP|$), we obtain a combined statistic – we informally call is a *predictor* of the performance of *Suffix Rank*.

Subprograms. Finally, we study which of the three steps (subprograms) is most critical for the performance of *Suffix Rank*. The total time per subprogram on various datasets is presented in Figure 11b. The most time is spent in the *refine* and *update* steps. The least time is spent during the *resolve* step, since at this step we work with aggregated counts of refined rank pairs, not with all the ranks of each chunk. This makes the disk-resident inputs much smaller than in the *refine* and *update* steps. Thus, these two steps should be optimized first. Due to the independent nature of these two steps (each works with at most two small chunks of the input at a time), they are trivially parallelizable.

6. CONCLUSIONS AND FUTURE WORK

We presented a new method for suffix sorting, which scales to arbitrarily large inputs. Our *Suffix Rank* is a simple, fast, and inherently parallelizable algorithm which can be used to build Suffix Arrays and FM-indexes to structure and efficiently search large string collections. Our algorithm is designed for real disks and takes an advantage of prefetching by performing sequential I/Os of large chunks of the input. The algorithm is highly practical in its simplicity and can be used to create off-line indexes in the background of database systems intended for storing sequential data.

To improve the performance of the *Suffix Rank* we plan to parallelize the chunk processing. Our algorithm makes this parallelization straightforward because both *refine* and *update* steps work independently with at most 2 small chunks of the input at a time. The *resolve* step can also be parallelized, because the refining of ranks is performed independently for each coarse rank. We envision a fully parallelized algorithm on a Hadoop-like shared-nothing architecture, where short chunks of input and output are sent and processed in parallel by separate subprograms delivered directly to the cluster nodes.

7. ACKNOWLEDGEMENTS

We thank Shawn Shuang and YuXuan Liu for their work on earlier versions of the *Suffix Rank* implementation. Consens acknowledges the support provided by NSERC and Compute Canada. We also thank the anonymous reviewers for valuable comments and suggestions which lead to a substantial improvement of the quality of this paper.

8. REFERENCES

[1] N. Askitis and R. Sinha. Repmaestro: scalable repeat detection on disk-based genome sequences. *Bioinformatics*, 26:2368–2374, 2010.

[2] M. Barsky, U. Stege, and A. Thomo. Suffix trees for inputs larger than main memory. *Information Systems*, 36(3):644–654, 2011.

[3] M. Barsky, U. Stege, A. Thomo, and C. Upton. A new method for indexing genomes using on-disk suffix trees. In *CIKM 2008, Proceedings*, pages 649–658, 2008.

[4] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *ACM-SIAM SODA 1997, Proceedings*, pages 360–369, 1997.

[5] T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and lcp arrays in external memory. *ACM Journal of Experimental Algorithmics*, 21, 2016.

[6] T. Conway and A. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27:479–486, 2011.

[7] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12:3.4:1–3.4:24, 2008.

[8] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS 2000, Proceedings*, page 390, 2000.

[9] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson, 2 edition, 2008.

[10] A. Ghoting and K. Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. In *SIGMOD 2009, Proceedings*, pages 827–840, 2009.

[11] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: Pat trees and Pat arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82, 1992.

[12] S. Hoffmann, C. Otto, S. Kurtz, C. Sharma, P. Khaitovich, J. Vogel, P. Stadler, and J. Hackermller. Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS Computational Biology*, 5, 2009.

[13] J. Kärkkäinen and D. Kempa. Engineering a lightweight external memory suffix array construction algorithm. In *BDCA, Proceedings*, pages 53–60, 2014.

[14] J. Kärkkäinen, D. Kempa, S. Puglisi, and B. Zhukova. Engineering external memory induced suffix sorting. In *ALENEX 2017, Proceedings*, pages 98–108, 2017.

[15] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *The Journal of the ACM*, 53(6):918–936, 2006.

[16] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5, 2004.

[17] N. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[18] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler Transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[19] R. Li, C. Yu, Y. Li, T. Lam, S. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25:1966–1967, 2009.

[20] F. A. Louza, G. P. Telles, S. Hoffmann, and C. Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms for Molecular Biology*, 12(1):1748–7188, 2017.

[21] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *SODA 90, Proceedings*, pages 319–327, 1990.

[22] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. Era: Efficient serial and parallel suffix tree construction for very long strings. *PVLDB*, 5(1):49–60, 2011.

[23] G. Nong, W. H. Chan, S. Q. Hu, and Y. Wu. Induced sorting suffixes in external memory. *ACM Transactions on Information Systems*, 33(3), 2015.

[24] G. Nong, S. Zhang, and W. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[25] S. Puglisi, W. F. Smyth, and A. Turpin. The performance of linear time suffix sorting algorithms. In *DCC 05, Proceedings*, pages 358–367, 2005.

[26] S. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

[27] J. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26:367–373, 2010.

[28] D. Tsirogiannis and N. Koudas. Suffix tree construction algorithms on modern hardware. In *EDBT 2010, Proceedings*, page 263274, 2010.

[29] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 9(1):1432–0541, 1995.

[30] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. Prospects and limitations of full-text index structures in genome analysis. *Nucleic Acids Research*, 40(15):6993–7015, 2012.

[31] P. Weiner. Linear pattern matching algorithms. In *SWAT 1973, Proceedings*, pages 1–11, 1973.

[32] Y. Wu, B. Lao, X. Ma, and G. Nong. An improved algorithm for building suffix array in external memory. In *PAAP 2019, Proceedings*, pages 320–330, 2008.

[33] 1000 genomes data portal. `https://www.internationalgenome.org/data`. Accessed: 2019-08-15.

[34] Gutenberg corpus. `http://www.gutenberg.org/robot/harvest`. Accessed: 2019-08-15.

[35] ICGC data portal. `https://dcc.icgc.org/repositories`. Accessed: 2019-08-15.

[36] Implementation of the eGSA algorithm. `https://github.com/felipelouza/egsa`. Accessed: 2020-06-15.

[37] Implementation of the eSAIS algorithm. `https://github.com/bingmann/eSAIS`. Accessed: 2020-06-15.

[38] Implementation of the SAscan algorithm. `https://www.cs.helsinki.fi/group/pads/SAscan.html`. Accessed: 2020-06-15.

[39] Raw reads from the 1000 genomes project. ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/. Accessed: 2020-04-15.

[40] Software library of lightweight suffix array construction algorithms. https://github.com/y-256/libdivsufsort. Accessed: 2020-06-15.

[41] Stxxl: Standard template library for extra large data sets. http://stxxl.org/. Accessed: 2020-06-15.

[42] Uniprot protein datasets. https://www.uniprot.org/downloads. Accessed: 2020-04-15.

[43] Wikimedia dump web site. ftp://ftpmirror.your.org/pub/wikimedia/dumps. Accessed: 2020-04-15.