Scalable and Efficient Construction of Suffix Array with MapReduce and In-Memory Data Store System

Hsiang-Huang Wu*, Chien-Min Wang, Hsuan-Chi Kuo*, Wei-Chun Chung and Jan-Ming Ho
Institute of Information Science, Academia Sinica
Taipei, Taiwan
{cmwang,wcchung,hoho}@iis.sinica.edu.tw {virtuoso.wu,hsuanchiku}@gmail.com*

Abstract—Suffix Array (SA) is a cardinal data structure in many pattern matching applications, including data compression, plagiarism detection and sequence alignment. However, as the volumes of data increase abruptly, the construction of SA is not amenable to the current large-scale data processing frameworks anymore due to its intrinsic proliferation of suffixes during the construction. That is, ameliorating the performance by just adding the resources to the frameworks becomes less cost-effective, even having the severe diminishing returns. At issue now is whether we can permit SA construction to be more scalable and efficient for the everlasting accretion of data by creating a radical shift in perspective. Regarding TeraSort as our baseline, we first demonstrate the fragile scalability of TeraSort and investigate what causes it through the experiments on the sequence alignment of a grouper (i.e., the SA construction used in bioinformatics). As such, we propose a scheme that amalgamates the distributed key-value store system into MapReduce to leverage the in-memory queries about suffixes. Rather than handling the communication of suffixes, MapReduce is in charge of the communication of their indexes, which means better capacity for more data. It significantly abates the required disk space for constructing SA and better utilizes the memory, which in turn improves the scalability radically. We also examine the efficiency of our scheme in terms of memory and show it outperforms TeraSort. At last, our scheme can complete the pair-end sequencing and alignment with two input files without any degradation on scalability, and can accommodate the suffixes of nearly 6.7 TB in a small cluster composed of 16 nodes and Gigabit Ethernet without any compression.

Index Terms—suffix array; MapReduce; Redis; in-memory processing;

I. INTRODUCTION

Suffix Array (SA), proposed in [2] and enhanced in [3], is more widely used because of better locality of memory reference and consumes less space than suffix tree [4]. Table I illustrates how to construct the SA of SINICA$^*$ in lexicographically ascending order, where $\odot$ is a delimiter and lexicographically smaller than the other characters. All possible suffixes are listed in the rightmost column and we sort them by comparing the characters from left to right. On the other hand, SA[$i$] is a sorted array composed of the indexes that indicate the corresponding sorted suffixes. Suppose the length of a string is $n$ and the sorting algorithm is comparison-based. The total number of suffixes and the time complexity would be $n$ and $O(n^2\log n)$ respectively since we include the time spent on comparing the characters (i.e. $O(n\cdot \log n)$). Hence the demands for the fast construction of SA has led to the development of linear time algorithms through exploiting the characteristic of the suffixes [5]–[7]. Also, SA construction that needs only $O(n)$ working space is presented in [8].

| Index i | SA[$i$] | Sorted Suffix | Suffix |
|---------|---------|---------------|--------|
| 0       | 6       | $\odot$      | SINICA$^*$ |
| 1       | 5       | A$\odot$     | INICA$^*$ |
| 2       | 4       | CAS          | INICA$^*$ |
| 3       | 3       | ICA$\odot$   | ICAS   |
| 4       | 2       | INICA$\odot$ | CA$\odot$ |
| 5       | 1       | NICA$\odot$  | A$\odot$ |
| 6       | 0       | SINICA$\odot$| $\odot$ |

TABLE I: Suffix Array of SINICA$^*$.

In the era of ”Big Data”, SA construction confronting large volumes of data becomes very critical and needs to be handled by the distributed computing frameworks such as Hadoop [9]. This is because the resources of the single machine, especially the memory, cannot afford these copious suffixes derived from ”Big Data”. More concretely, libdivsufsort [10], state-of-the-art SA construction algorithm for single thread, claims $O(n\log n)$ worst-case time using only $5n + O(1)$ bytes of memory space. Say, there are $10^9$ strings and each of them consists of 20 characters (i.e. $n = 20$). If we want to perform in-memory sorting, the required memory would be 100 GB at least. It implies switching from scale-up to scale-out is inevitable, but how good is scale-out?

Many distributed systems that emerge from behind cloud computing improve speedup by the promise that cloud computing provisions unlimited resources. This naturally leads to the focus on how we fit the application into a distributed computing framework because the speedup is thought to be improved directly by scale-out: adding more resources to the systems. We argue that such plunge into scale-out, inherently associated with the mechanisms of the distributed computing frameworks, is not the panacea for speedup; instead, this desirable performance gain might belie its poor capacity for the coming larger volumes of data.

We use scalability and efficiency as our criteria for how good the capacity of a distributed system for SA construction is. The outline is described in the following. Two directions of scalability—scalability$_1$ and scalability$_2$—defined in [11] are our conceptual and main considerations. To elaborate on scalability$_1$, we introduce three types of scalability from [12] to explore it in more detail aspects, and for scalability$_2$, we introduce the efficiency to understand whether the incorpo-
ation of the in-memory data store systems into MapReduce (MR) is cost-effective or not.

- **Scalability** is the ability to handle increased workload (without adding resources to a system).
  - Load scalability
  - Structure scalability
  - Space scalability
- **Scalability** is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system’s capacity.
  - Efficiency

Involving network, I/O, hierarchical storage, and computing units, a distributed system nowadays cannot ascribe its performance gain to one kind of resources. Thus, some measurement commensurate with time, for example latency or throughput, is necessary to reason about the performance while being analyzed to get insights into which factor the performance is bounded by. We identify data store footprint with time, where the data store refers to memory and disk in this paper. The performance analysis of the data store footprint is carried out by investigating which one of the four factors—CPU, memory, disk and network—is bound to be the bottleneck mostly.

It is because reading and writing the sheer size of data through I/O almost predominates in large-scale data processing that the extent of space required can reflect the extent of time consumed. We keep three observations concerned with data store footprint in perspective:

- **Fine-grained data movement**. Suffixes are tiny in comparison with the scale of the input.
- **Data access pattern**. Access of suffixes while sorting is irregular and very frequent.
- **Trading disk I/O for memory I/O**. Not only does local memory access outperforms local disk access but also remote memory access competes it in latency.

In [14], the SA construction purely by MR is proposed and the experimental results show non-linear speedup within a range of 30-, 60- and 120-core cluster while the problem size is fixed. According to the authors’ investigation, it is caused by writing the replications and not used values (unsorted suffixes) to Hadoop Distributed File System (HDFS). Furthermore, we assess the performance improvement in terms of efficiency and assume that the experimental result of the 30-core cluster is seen as the baseline. By \( \frac{\text{speedup}}{p} \), the efficiency of the 60-core cluster and the 120-core cluster are \( \frac{145}{120} = 72.5\% \) and \( \frac{138}{4} = 38.25\% \). Despite the rough calculation, we still can learn that scale-out without discerning what kind of resource matters might thwart the advancement in efficiency. Regarding scalability, [14] claims the SA construction scales linearly. Through the metrics of the experiments, we make deductions about its scalability and find it scales linearly as well. Nevertheless, we will prove that SA constructed only by MR overloads the disks, thereby easily causing a breakdown in the scalability. On the other hand, inspired by this analysis that MR can scale linearly, we delineate our scheme in a way that adopts MR as the foundation and exhibits the keen insights of scalability and efficiency with respect to scalability.

Through the analysis of data store footprint, we show how our scheme can surmount the fragile scalability.

It is always convincing to demonstrate how our scheme reaches the goals using a real application. Sequence alignment is a very important application in bioinformatics, and highly relies on two index structures—SA and Burrows-Wheeler Transform (BWT) [14]. The latter can be derived from the former. As an illustrative example of our scheme, we use the authentic sequencing data of grouper genome. The total input size is 64 GB (325,718,730 reads) and the length of each read is about 200 bp (i.e. 200 characters). Without getting rid of any suffix, the total suffixes including their indexes would be around 6.7 TB, hundred times the input size. Furthermore, we start a SA construction with TeraSort as our baseline for analysis. According to the deficiency identified in TeraSort, we develop our scheme that evolves into a system that makes available more scalable and efficient SA construction. More specifically, there are 6.7 TB suffixes during the construction in our cluster composed of 16 nodes and Gigabit Ethernet, and without sacrificing the speedup, our experiment takes 11 hours to generate the output that contains the suffixes and the indexes of the corresponding reads.

II. MapReduce

MR is the enabling technology for large-scale batch processing [15]. The dataflow Map-Sort-Shuffle-Merge-Reduce mainly constitutes MR. Figure 1 illustrates MR in a manner that considers MR as a programming model. For application modeling, the relation with Map () and Reduce () conforms with Map(, ) \( \rightarrow \) [(){}] and Reduce(, , ) \( \rightarrow \) [(), ()], where the parentheses (...) and brackets [...] denote a key-value pair and a list respectively. Furthermore, Map () deals with only one key-value pair (i.e. atomic) every time and then generates the intermediate key-value pairs, whereas Reduce () aggregates these intermediate key-value pairs associated with the same key. As the example shown in Figure 1, an application collects two different types of data, white and striped, and then, generates two corresponding collections. Suppose there are 100 units of data that are either white (50%) or striped (50%) and scattered over the input file.

Two steps are in this application: identifying every data and collecting the data of the same type. Giving every unit of data a unique number (e.g., 1 to 100), we regard as the serial number and as the corresponding unit of data. Whenever Map () reads the key-value pair (,, ), it recognizes and then emits the intermediate key-value pairs (, ) if is white (striped). As to the Reduce (), it is in charge of collecting according to the key of the intermediate key-value pairs and finally, outputs both the 50 units of white and striped data. In the application modeling, we think in the way that each key-value pair has its own Map () and the intermediate key-value pairs with the same key (one group) are processed by one Reduce ()

The execution framework of our example is assumed to have two nodes of which each can accommodate two slots for the
Fig. 1: The interpretation on application modeling and execution framework of MR.

map tasks (i.e. mappers) and one slot for the reduce tasks (i.e. reducers). This assumption is for Hadoop 1.2 for simplicity. For Hadoop 2 YARN, slots are replaced with containers which are not limited to mappers or reducers. In reality, the mapper deals with a batch of data called Input Split that contains more than one key-value pair. On the other hand, the reducer is allowed to handle more than one group under the constraint that all the intermediate key-value pairs associated with the same key must be processed by the same reducer. Once the application implemented in MR starts, InputFormat reads the input file from HDFS and divides it into several Input Splits. In our example, the number of Input Splits is 4 and each of them contains 25 key-value pairs (invoking Map() 25 times). The number of Input Splits also determines the number of mappers, whereas the number of reducers can be specified by the programmers through the system settings. Every mapper processes the 25 key-value pairs and generates the intermediate key-value pairs. Partitioner dispatches them to the reducers according to their corresponding partition numbers. Sorting is performed on the files spilled by a mapper according to the partition number and then, key in order. Merging is applied to the Map outputs (fetched from mappers) of a reducer and groups them into one single file before the reducer begins. It is possible that one reducer might have several different intermediate keys (invoking Reduce() several times). In the end, OutputFormat writes back the results to HDFS.

In the following, we experiment on Hadoop 2.7.2 with the cluster in size of 16 physical nodes. Table II illustrates the total hardware resources and the resources managed by YARN. There are two types of Intel(R) Xeon(R) CPU and each node is equipped with two CPUs of the same type. E5620 and E5-2620 are quad- and hex-core CPU that can provide 8 and 12 threads respectively. We assign 2 GB and 8 GB memory to a mapper and a reducer of which the heapsize is 1 GB and 7 GB respectively. The total number of VCore is 128 (the default value of VCore is 8 for each node). The replication factor is set to 1 to avoid excessive data writing to the disk. Note that we assign 1 GB memory to ApplicationMaster (AM) and ask all nodes to provide this extra 1 GB memory to prevent the memory slots for reducers from the occupancy of AM. For instance, we actually ask each node to donate 17 GB memory so that, at most, 8 mappers and 2 reducers can run concurrently. To make such 1 GB not distracting, we omit it in the following sections.

### Table II: Summary of the resources in a Hadoop cluster with 16 physical nodes. The number of VCore and memory is donated by each node evenly.

| Resources Managed by YARN | VCore | Memory | Disk |
|---------------------------|-------|--------|------|
|                           | 128   | 256 GB | 28.24 TB |

**Task III: Baseline: TeraSort for Suffix Array Construction**

Taking the ostensible advantages of sorting large volumes of data and even including the optimizations [14], we argue that SA construction with MR can only alleviate the intrinsic problems of scalability and efficiency in quantity but cannot ameliorate them in quality. It is because there exists an essential trait inherent in SA construction with MR that we cannot obviate—**keeping every suffix in place**. Without involving the optimizations for SA construction, TeraSort presents a simple method for analyzing the impact of such trait on scalability and efficiency. On the other hand, paired-end sequencing and alignment [15]—a popular technology in next-generation sequencing in bioinformatics—is adopted as our target application for analysis. Paired-end read means that a DNA fragment is read twice from one and the opposite directions. We prepare two input files for the paired-end sequencing and alignment of the group genome, where one input file contains those reads which are generated in one direction and the other input file contains those reads which are generated in reverse order. Each input file is about 32 GB in size and each read is about 200 bp. All the suffixes of one input file is generated first for TeraSort and they are about 3.4 TB in size, which is consistent with the self-expansion factor ($\frac{1 + 200}{2} \approx 100$). To make our baseline convincing and a fair comparison, except the settings specified in Section II we apply default settings to MR and distribute the input data in proportion to the sizes the disk space. Suppose there are six input files in the same size, we would distribute one to the nodes with 825 GB or 870 GB, two to the nodes with 1.61 TB, and three (instead of four) to the nodes with 3.22 TB to avoid too many mappers running on them. Furthermore, we choose 64 reducers at most for the reason that each physical node can afford the moderate amounts of mappers ad reducers without crashing the systems.
The execution time taken by a system, especially involving parallel computing, is subject to many factors intertwined with each other and hardly isolated completely. So, the performance analysis with respect to time cannot clearly reason about the requirements of the system for large-scale data processing. Say, a system starts over a new task to finish the application after a task fails. How do we decouple the effect of the failed task from the whole application running when such case is non-deterministic? Furthermore, what if the running application comprises several slow tasks? More specifically, what is the exact execution time that this system is supposed to take? In other words, we think execution time is still suitable for the judgement about the performance but seems unwieldy for the performance analysis when large-scale data processing is considered. Thus we decide to abandon the attempt to evaluate how much time MR takes to sort the suffixes. Instead we propose an invariant and analytical abstraction commensurate with the time that a system is supposed to take—tracking how much the effective data is read from or written in the storages. We call it data store footprint, and the data constituting the output is defined as the effective data. In the previous case, the data associated with a failed task doesn’t count as the effective data if the output is not produced by it. As such, data store footprint is deterministic and invariant. To make the data store footprint analytical for MR, we develop a model for data store footprint shown in Figure 2 based on the dataflow Map-Sort-Shuffle-Merge-Reduce. The effective data in this model is categorized as the shuffling data, HDFS Read/Write, and Local Read/Write.

Fig. 2: The model for data store footprint in TeraSort.

We are more focused on how Local Read/Write changes with the size of the input since TeraSort doesn’t change the sizes of the input, output and shuffling data. Table III shows the data store footprint of 5 different sized inputs. In every one of these 5 cases, we regard the size of the input (i.e. HDFS Read) as 1 unit to analyze how many units TeraSort needs to sort the suffixes as the input size increases. To be concise, we use XR and YW to represent X units for reading and Y units for writing respectively, where X and Y are arbitrary positive real numbers. For example, Map in Case 1 performs 2.07W for local writing and the actual size of the written data is 2.07 × 637.18 ≈ 1318.96 GB. Similarly, Reduce in Case 2 performs 1.37R for local reading and the actual size of the read data is 1.39 × 1.24 ≈ 1.72 TB. In addition, for the first 4 cases, we repeat each of them five times and produce two statistics, mean (µ) and standard deviation (σ) to depict the scalability of TeraSort and use Case 5 to point out its breakdown. We make two observations about the data store footprint:

1. On Map-side, the load of Local Write is twice as much as the load of Local Read.
2. The loads of Local Read and Write on Reduce-side are equivalent and increase as the input size increases.

We delve deeply into the data store footprint by mining the counters of the mappers and reducers to reason about what causes such amount of data read from and written in the local disks. As shown in Figure 3 the data in the buffer is spilled in the local disk once the buffer reaches the level of 80% (i.e. 80 MB). Because the size of the input split for every mapper is around 128 MB, every mapper spills the data to the local disk twice and two intermediate files are merged into one later on. For the sake of fault tolerance, the resulting file resides in the local disk once a reducer fails to complete and an initiated reducer can fetch the data. Thus, there are approximate 1R and 2W for local disks on the Map-side.

Fig. 3: Local I/O loading of TeraSort on the Map-side.

In Figure 4 the memory buffer is determined by 70% of the heapsize (0.7 × 7 = 4.9 GB) and a memory merger is triggered once the memory buffer reaches 66% full. In Case 1, a reducer receives the data of 20.56 GB and the number of spilled files is reckoned to be around 6 (20.56 ÷ 3.27 ≈ 6). Since the default value of io.sort.factor is 10, all files residing in the local disk are sent to the Reduce() without merging. That is why there are about 1R and 1W on the Reduce-side. Due to the subtle mechanism of merging, we estimate how much local disk I/O would be in Case 5 by the following steps:

1. There are around 35 (111.38 ÷ 3.27 ≈ 34.06) spilled files, where a reducer receives the data of 111.38 GB.
2. In the first round, we merge 28 spilled files into 3 groups so there are 3 merged files and 7 spilled files left. Thus, 28R/W is needed in this round.
3. In the second round, we merge all the files into one file. 1R/W is needed in this round.
4. The total units are (28 + 1) × 1.03 ≈ 1.88.

Figure 5 illustrates the elapsed time of those 5 cases in Table III to examine the scalability of TeraSort. It shows that TeraSort scales in a linear sense from Case 1 to Case 4 but no longer holds Case 5 in such linearity. The failure of Case 5 is mainly caused by the errors about the memory issues such as GC overhead limit or Java heap space, whereas we find that the increasing of the local disk I/O is endangering scalability as well. As to the memory issues, TeraSort picks the first 10 bytes as the key to group the
suffixes for sorting. However, it is very common that plenty of suffixes are grouped together for sorting because their first 10 characters are the same (e.g. ATATATATAT), thereby stressing the heap space and garbage collection (GC) out. In contrast, the lack of the enough disk space would compel TeraSort to start those reducers running on the nodes with less disk space over on the other nodes, thereby taking more time to complete SA construction. Suppose two reducers in Case 5 are running at the same pace on the same node, their temporary files and outputs would occupy about 644 GB (111.38 × 2 × 2.89) disk space which in turn is very likely to make the node unusable and reschedule the reducers to the other available nodes. Worse still, such the deficiency may cause non-deterministic elapsed time. In our environment, the resources of all the nodes are used only for our experiments to defer such the breakdown.

As illustrated in Table IV to reinforce the hypothesis above, we increase both the memory and input size to eliminate the memory issues and make the lack of the disk space more stressful for our cluster respectively. Although the ratios of Local Read/Write is smaller than those of Case 5 due to the larger heapsize, the size of the files generated by a reducer would be about 738 GB (129.02 × 2 × 2.86). We do find that all failed reducers are caused by the lack of the enough disk space, which in turn affects the completion time dramatically—the σ of the case in Table IV is much higher than those of Case 1 to 4. Besides the non-deterministic completion time, we conclude that on-disk merging is inevitable and would result in more extra local disk I/O on the Reduce-side when the input size becomes larger. Generally, as shown in Figure 4, this is because there would be many files spilled from the memory buffer and then, merged into one single file through more than one iteration if needed.

Note that there is no performance optimization by parameter tuning for TeraSort. Based on the analytical discussion above, there exits an essential trait inherent in TeraSort that we cannot obviate—keeping every suffix in place. Under the influence of this trait, we argue that, for MR, the innate capability to handle the block devices and the superior mechanism of message passing restrain TeraSort from scaling well, no matter how we tune the parameters for TeraSort. This is because the necessity for keeping every suffix in place easily makes the space requirement grow to the extent that the heavy I/O loads of the local disks and the strong demand for the space to store those processing suffixes degrade the performance severely. We believe that any parameter tuning for the performance optimization without considering this issue would not give the desired promise of the scalable SA construction.

IV. SCHEME FOR SCALABLE AND EFFICIENT SUFFIX ARRAY CONSTRUCTION

To resolve the issue of keeping every suffix in place, we set a goal of keeping only the raw data in place, which means a suffix is obtained via the query about it. We examine the data store footprint of this goal specifically through three criteria: data movement, data access pattern, and storage I/O. In our goal, the data movement of suffixes would be more fine-grained since the suffixes are generated, stored, and queried on the fly in tiny size. Besides, random and irregularly frequent access to the raw data not only destroys the locality, thereby making caching difficult, but also exacerbates the overhead of the storage I/O and the network communication. From the

| Input size       | Case 1 | Case 2 | Case 3 | Case 4 | Case 5* |
|------------------|--------|--------|--------|--------|---------|
|                  | Map    | Reduce | Map    | Reduce | Map     |
| Local Read       | 1.03   | 1.03   | 1.03   | 1.39   | 1.03    |
| Local Write      | 2.07   | 1.03   | 2.07   | 1.39   | 2.07    |
| HDFS Read        | 1.00   | 1.00   | 1.00   | 1.00   | 1.00    |
| HDFS Write       | 1.01   | 1.01   | 1.01   | 1.01   | 1.01    |
| Shuffle          | 1.03   | 1.03   | 1.03   | 1.03   | 1.03    |
| Time (min.)      | µ=61.8 | σ=1.30 | µ=143.4| σ=4.83 | µ=230.4 | σ=12.30 | µ=312.3 | σ=12.65 | µ=709.4 | σ=95.55 |

Note that four experiments of Case 5 don’t complete SA construction due to the failures of some reducers, which in turn takes longer time than one succeeded experiment. We use all of them as the metrics.
examination above, we find out that SA construction is possessed of the extreme scale of processing the data—generating each suffix (e.g. a few bytes) and sorting all the suffixes (e.g. several Terabytes). It indicates that relying on only one type of storage (e.g. disk) might improve the performance of one extreme by sacriﬁcing the performance of the other extreme.

Since MR has proven its great capability of sorting, we conceive of an abstraction that requires the space for only the raw data and is competent to access the sufﬁxes at speed by taking advantage of memory. Having the beneﬁts of memory, we can overcome the problem of the data access pattern and alleviate the I/O loads by trading disk I/O for memory I/O. Though the distributed in-memory ﬁle systems, Alluxio (formerly known as Tachyon) for example, are popular for accelerating large-scale data processing by exploiting the better speed of data access in memory. However, the structure behind the distributed in-memory ﬁle systems still emphasizes the management of ﬁles in a way that considers the underlying storages as the block devices. The drawback to the block devices is that, given an index of a sufﬁx, it takes time to seek the target block, retrieve the whole block, get the wanted sufﬁx, and discard the other data in that block. Moreover, we can infer from the extreme scale we just describe above that the distributed in-memory ﬁle systems can indeed help to enhance the performance of SA construction but the majority of access time would be made redundant which results in poor scalability.

Instead of the distributed in-memory ﬁle system, we adopt the distributed in-memory data store system like Redis [17] as the realization of our abstraction. As an in-memory key-value data store system for small chunks of arbitrary data, Redis is natural to be integrated into MR through the communication of key-value pairs and easy to scale in number and size. Figure 6(a) illustrates the data store footprint in our scheme. In the following sections, we introduce our scheme in detail and assess its possibilities from a standpoint of load scalability, structure scalability, space scalability and efﬁciency.

A. Load Scalability

If a system is possessed of load scalability, it can function gracefully at light, moderate, or heavy loads while making good use of available resources. We rephrase it as balancing the loads on the memories, disks, and network communication by the cooperation among them. In other words, our scheme offloads keeping every sufﬁx in place with disks onto the concept of keeping only the raw data in place with memories and network communication to achieve better load scalability. Figure 6(b) conveys this concept that the in-memory data store system (i.e. a bunch of Redis instances) takes charge of storing the raw data in memories and responding the queries about sufﬁxes via network communication, whereas MR only needs to manage the indexes of the sufﬁxes to abate the loading of local disk I/O since the size of the indexes is relatively smaller than the size of sufﬁxes. The same format as what we use for TeraSort, the ﬁrst and second columns in Input File in Figure 6(b) are full of the sequence numbers and reads respectively. To distribute the reads to the Redis instances evenly, we make every sequence number modulo the number of the Redis instances to determine which Redis instance the key-value pair (Sequence Number, Read) goes to. In addition to putting the raw data in the Redis instances, Map sends the indexes of the corresponding sufﬁxes to Redcue so that Reduce can acquire them from the Redis instances using the indexes. To make the number of sufﬁxes be dispatched to each reducer evenly, our scheme adopts the partitioning method similar to TeraSort and [13] by assuming the randomness of the sufﬁxes. Given the number of reducers (e.g. n), we sample N sufﬁxes and sort them to estimate the ranges of the sufﬁxes, where N = 10000 × n. In our case, after sampling 320000 sufﬁxes and sorting them, we pick the 10000th, 20000th, ..., and 310000th sufﬁxes to determine the boundaries of the ranges. Finer partition can be achieved by increasing the number of sampling points. Note that our scheme overcomes the self-expansion by shifting it from disks to memories and network communication. Through the speed of memory access and network communication, our scheme can enhance the load scalability which outweighs the decrease of available memories. We prove it in Section IV-D.

B. Structural Scalability

If a system is possessed of structural scalability, its implementation or standards do not impede the growth of the input size. We investigate structural scalability in our scheme with the following requirement: it is relatively insensitive to the growth of the input size with respect to TeraSort. By means of these Redis instances, our scheme encapsulates those sufﬁxes in the raw data and handled them on demand to reduce the self-expansion effect on the sufﬁxes in exchange for the self-expansion effect on the indexes of those sufﬁxes. Here comes the question: wouldn’t such self-expansion destroy the scalability like what just happened to TeraSort?

The index we mean here is the key-value pair communicating in MR and can be used to acquire the corresponding sufﬁx. As shown in Figure 6(b), once we know the Sequence Number, we look it up in the target Redis instance, ﬁnd the read, and extract the sufﬁx from the read by the offset. Rather that using String, we choose the numerical representation in long or int due to its better capability of accommodating more objects to address within a ﬁxed number of bytes. In addition, it is also ﬂexible to expand as the input size increases with very little overhead (e.g. a few bytes). The exact way to represent Sequence Number and offset numerically is Sequence Number × 1000 + offset since offset ranges from 0 to 200. The retrieval of Sequence Number and offset can be done by division and modulo respectively. On the other hand, there are only ﬁve possible characters in a read: A, C, G, T and $. With base 5, we use $ = 0, A = 1, C = 2, G = 3 and T = 4 to represent the key used in MR numerically. To ﬁt long or int, we encode the preﬁx of every sufﬁx in a ﬁxed number of characters (e.g. 10). So do the boundaries of the ranges for the partition. Moreover, there is every chance that a lot of the short sufﬁxes are grouped
together for sorting, which more likely results in the errors about GC overhead limit or Java heap space. We discover that if the length of a suffix is smaller than the length of the prefix we defined, the prefix is actually the suffix itself. Say, the prefix in a length of 10 characters for a suffix AGT$ is AGT$ itself. Reaping such the benefit, our scheme doesn’t have to sort those suffixes because they are the same, which provides the memory relief for the reducers and saves time. Figure 7 illustrates how our scheme partitions the sorting group and how the size of a sorting group changes according to the length of the prefix. Since Prefix1 is in a length of 3, the prefixes of those four suffixes are all ATG. Conforming to MR, these suffixes are grouped together and sorting at the same time. Applying Prefix2, we would have four sorting groups and each contains one suffix. The order among these sorting groups is maintained by the partitioner, thereby matching up with the order in the scenario of Prefix1. There is a rule of thumb: the longer length of the prefix, the smaller size of the sorting group which in turn requires less memory to sort the suffixes.

Fig. 7: How different length of the prefixes determines the number of the groups and the size of the group for sorting.

Acquiring the suffixes one by one through the network communication squanders the time we gain from the disk I/O and makes the memory I/O busy. To utilize the network communication and memory I/O efficiently, our scheme aggregates those indexes of the suffixes which are stored in the same Redis instance, and retrieves the suffixes from it at one time to save the communication cost. Since Redis doesn’t support the retrieval of the multiple partial contents, we add a new Redis command called "mget suffix" [18] in Redis and its corresponding function in Jedis [19] to retrieve the whole suffixes back instead of the whole reads. As such, our scheme almost saves half an amount of data communicating in the network while acquiring the suffixes. Because putting multiple reads at one time is permitted in Redis, our scheme lets the mappers aggregate those reads which are assigned to the same Redis instance and put them to it when the mappers finish reading the input file.

In summary, our scheme further relieves the self-expansion effect on the suffixes by aggregating as many suffixes as possible while storing the reads and acquiring the suffixes. As such, not only is the access of the memory and network I/O reduced, but the utilization of the network bandwidth is improved. Concerning the self-expansion effect on the indexes of the suffixes, our scheme has superior capability of restricting it in a constant factor unless there exists some sorting group that cannot fit into the memory for sorting. For example, int contains four byte so the threshold is 13 because the numerical value of TTTTTTTTTTTTTTTTTTTTTTTTTT is 1220703124, which is the largest number smaller than 2147483647. So the total bytes of a key-value pair used in MR is 12 bytes (i.e. int+long), which is smaller than 100 by a factor of 8 and has nothing to do with the length of reads. Only when some sorting group is too big to fit into the memory, do we need to partition the sorting groups in finer grain by lengthening the prefix. If we replace int with long to accommodate the longer prefix (the threshold would be 26), the total bytes are just 16 bytes, which is still smaller than 100 by a factor of 6.

C. Space Scalability

If a system is possessed of space scalability, its memory requirements do not grow to intolerable levels as the number of items it supports increases. Here it is referred to the economical usage of the heap and two factors are considered: the size of the sorting group and the type of the garbage collector. Involved in TeraSort as well, the former factor usually results in Java heap space while a reducer attempts to allocate the memory for the big sorting group, or GC overhead limit when little progress is made on the GC. On the other hand, if the sizes of the sorting groups are very small, a reducer would waste time on the overhead of switching from group to group for sorting only the small number of the suffixes. Furthermore, the amount of suffixes
acquired from the Redis instance would also be very small, which means low throughput. This dilemma comes from the fact that the size of a sorting group varies all the time, which in turn influences the sorting time and the throughput. To make the sizes vary within a narrow range, we accumulate the sorting groups without sorting until such accretion exceeds some threshold. That is, we prevent not only the heap from the shortage of available memory by shrinking the sorting group size but also the time of sorting from the switching overhead by collecting the sorting groups together. We choose $1.6 \times 10^6$ as the threshold value just because the experiments with this value are better than $3$. The performance issue that the throughput of the suffixes acquired from the Redis instance is decreased when stop-the-world GC starts from the Redis instance is verifiable in Table V. Since our scheme includes the generation of suffixes, it takes more time in Map (25 mins in average) than TeraSort does. Nevertheless, we can see the significant reduction of the local disk I/O in contrast to TeraSort. This is because MR handles only the indexes of the suffixes rather than the suffixes.

Handling the indexes of the suffixes also helps to decrease the amount of shuffling data. Some might argue that the network usage in our scheme is higher than TeraSort because there is extra network communication of the suffixes generated from those Redis instances. Actually, it is the procedure of generating the suffixes and necessary for TeraSort as well. The reason why we exclude the time of generating the suffixes from TeraSort is fairness—writing the suffixes into the disk takes a lot of time. Once we take the generation of the suffixes into account, our scheme really reduces the network usage for shuffling by comparison with TeraSort.

Benefiting from the smaller key-value pairs, the reducers don’t need to merge the spilled files more than one iteration. Taking Case 5 as an example, one reducer receives about 17 GB data $(45 \times 17 GB \approx 17 GB)$ and there are only 6 spilled files $(\frac{45}{50} \approx 6)$ which we can merge in one iteration. That’s why the Local Read/Write of Reduce is the same as the Shuffle. The throughput of the suffixes acquired by a reducer is about 20 MB/sec and doesn’t last for the whole time. We think the network bandwidth (1 Gigabit in our scheme) is not fully utilized and could be improved by increasing the number of the reducers or the size of the sorting group. We roughly classify the computation time into three categories—getting suffixes, sorting, and others—where their percentages are about 60%, 13%, and 27% respectively. Since the latency of acquiring the suffixes is the dominant factor, the high-end network like InfiniBand would be very helpful to our scheme. Nevertheless, as shown in Figure 8 our scheme (blue) outperforms TeraSort (green) in terms of time and space as the input size increases regardless of the inclusion of generating suffixes. We believe that the access of the suffixes through the memory and network with the in-memory data store system is much better and more compact than through the block devices with the conventional filesystem.

While the better scalability of our scheme is qualitatively proved and exhibited in quantity, some might argue our scheme is supposed to be better because of the extra memory, and claim the scalability of TeraSort can also be improved using the same way. As discussed in Section III the memory issues and deficiency of disk space could be appeased by including more memory in MR but the question is—how efficient is the way you add the extra memory? There are

D. Analysis and Efficiency of Our Scheme

We let our scheme execute on the same MR environment presented in Section III for the reason that the enhancement of the scalability and performance can be clearly reflected in data store footprint and time respectively. As such, we let every node donate the extra memory for its Redis instance to accommodate the input files and set the length of the prefix $23$. The overhead of storing the input data in these 16 Redis instances is about 1.5 times as much space as the input size due to the metadata. For instance, those 16 Redis instances need 48 GB memory to store the input data of 32 GB in size (i.e., each node has to donate the extra 4 GB memory for that). In Table IV we normalize the metrics by regarding the size of output as 1.01 unit for intuitively comparing the workload on the disk I/O. This is because the outputs of TeraSort and our scheme must be the same so we use the output size as the reference point. Note that the input data of Case 1 to 5 is exactly the same as those in Table III.
Table V: Data store footprint of our scheme with 32 reducers and the elapsed time includes the generation of suffixes. Note that Case 6 is the SA construction for the pair-end sequencing and alignment with two input files.

Table VI: Data store footprint of mem_heap with 32 reducers of which each uses 16 GB physical memory and 15 GB heap. We show only Local Read/Write on the Reduce-side because the other parts are the same as Table III.

We claim that the scalability_{1,2} of our scheme is radically different from those of TeraSort. Compared with mem_reducer, our scheme apparently presents smaller value of a, almost the same value of b, and—the most vital—bigger value of breakdown. On the other hand, with such a and b, the speedup of our scheme is getting more significant while the input size is getting larger. Furthermore, our scheme is more efficient than

\[
f(x) = \begin{cases} 
  ax + b & \text{if } x < \text{breakdown} \\
  N/A & \text{otherwise}
\end{cases}
\]
TeraSort with respect to the memory usage. In table VIII we list the efficiency of Case 1 to 4 but ignore Case 5 due to its unstable metrics, where the $\mu$ is adopted in the calculation of speedup. Neither mem\_heap nor mem\_reducer can achieve more than 75% efficiency. In contrast, our scheme performs amazing efficiency, even greater than 100%. This is because the memory used for storing the input data is relatively small whereby the mem\_ratio is close to 1. Although it is strange that the efficiency could be greater than 100%, it is another evidence that the scalability of our scheme is essentially different from that of TeraSort. Say, if our scheme with 32 reducers is considered as the baseline, the efficiency of our scheme with 64 reducers would not be greater than 100% since their scalability is structurally the same.

| Input size | Case 1 | Case 2 | Case 3 | Case 4 | Case 5* |
|------------|--------|--------|--------|--------|--------|
| Time (min.) | 1.03   | 1.03   | 1.03   | 1.03   | 1.03   |
| mem\_ratio | $\mu=46.8$ | $\mu=100$ | $\mu=156.6$ | $\mu=242.8$ | $\mu=365.8$ |
| mem\_reducer | $\sigma=3.56$ | $\sigma=0.7$ | $\sigma=2.41$ | $\sigma=7.53$ | $\sigma=13.83$ |

TABLE VII: Data store footprint of mem\_reducer with 64 reducers of which each uses 8 GB physical memory and 7 GB heap. The breakdown occurs in Case 5: two experiments succeed in SA construction, whereas the other three fail due to the oversize sorting group.

It is worth noting that our scheme could be faster by not writing the suffixes into HDFS. This is because the suffixes can be obtained through the Redis instances with their indexes. In other words, our scheme can also save the time in the following stages by exploiting the concept of keeping only the raw data in place. The reason why we write them out is for the fair comparison with TeraSort.

V. CONCLUSION

We have presented a quantitative analysis of SA construction in detail, and qualitatively pointed out the limitation of the scalability in TeraSort: excessive local disk I/O when the input gets enormous. Though our analysis is based on the application of bioinformatics which possess a characteristic of many small pieces of data, we have deeply addressed how the scalability of TeraSort collapses and found that keeping every suffix in place is the reason why TeraSort cannot scale well.

As such, keeping only the raw data in place has been proposed as a conceptual guideline for our scheme to trade off memory I/O and network against disk I/O. Based on this guideline, we have proposed a scheme for scalable and efficient SA construction built on MR and distributed in-memory data storage system. Our scheme allows the suffixes to be generated from distributed in-memory data storage system and alleviates the self-expansion effect by taking the advantages of memory I/O and network. Instead of passing the suffixes, MR plays the role of communicating the indexes of suffixes to abate disk I/O during SA construction. In addition, we have developed our scheme with respect to load scalability, structure scalability, and space scalability whereby our scheme can exhibit better scalability. Through the efficiency, we have shown that the scalability of our scheme is better than TeraSort too. All the experiments in the paper are conducted using the authentic grouper genome and demonstrated to convince us that our scheme can perform scalable and efficient SA construction.

REFERENCES

[1] O. O’Malley, “TeraByte Sort on Apache Hadoop,” http://sortbenchmark.org/YahooHadoop.pdf, May 2008.
[2] U. Manber and G. Myers, “Suffix Arrays: A New Method for On-Line String Searches,” in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, ser. SODA ’90. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1990, pp. 319–327.
[3] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing Suffix Trees with Enhanced Suffix Arrays,” Journal of Discrete Algorithms, vol. 2, no. 1, pp. 53 – 86, 2004.
[4] E. Ohlebusch and S. Gog, “Efficient Algorithms for The All-Pairs Suffix-Prefix Problem and The All-Pairs Substring-Prefix Problem,” Inf. Process. Lett., vol. 110, no. 3, pp. 123–128, Jan. 2010.
[5] J. Kärkkäinen, P. Sanders, and S. Burkhardt, “Simple Linear Work Suffix Array Construction.” Springer, 2003, pp. 943–955.
[6] D. K. Kim, J. S. Sim, H. Park, and K. Park, Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003 Morelia, Michoacán, Mexico, June 25–27, 2003 Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ch. Linear-Time Construction of Suffix Arrays, pp. 186–199.
[7] P. Ko and S. Aluru, “Space Efficient Linear Time Construction of Suffix Arrays,” in Journal of Discrete Algorithms. Springer, 2003, pp. 200–210.
[8] W.-K. Hon, K. Sadakane, and W.-K. Sung, “Breaking a Time-and-Space Barrier in Constructing Full-Text Indices,” in Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on, Oct 2003, pp. 251–260.
[9] http://hadoop.apache.org/
[10] Y. Morii, https://github.com/y-256/libdivisufsort
[11] C. B. Weinstock and J. B. Goodenough, “On Systems Scalability,” in Distributed Systems, “ in Stanford CS295 class lecture, ser. Spring, 2007.
[12] A. B. Bondi, “Characteristics of Scalability and Their Impact on Performance,” in Proceedings of the 2Nd International Workshop on Software and Performance, ser. WOSP ’00. New York, NY, USA: ACM, 2000, pp. 195–203.
[13] J. Dean, “Software Engineering Advice from Building Large-Scale Distributed Systems,” in Stanford CS295 class lecture, ser. Spring, 2007.
[14] R. K. Menon, G. P. Bhat, and M. C. Schatz, “Rapid Parallel Genome Indexing with MapReduce,” in Proceedings of the Second International Workshop on MapReduce and Its Applications, ser. MapReduce ’11. New York, NY, USA: ACM, 2011, pp. 51–58.
[15] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6, ser. OSDI’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.
[16] “An introduction to next-generation sequencing technology,” http://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_2016.pdf
[17] http://redis.io/