Correct and stable sorting for overflow streaming data with a limited storage size and a uniprocessor

Suluk Chaikhan, Suphakant Phimoltares and Chidchanok Lursinsap

Advanced Virtual and Intelligent Computing (AVIC) Research Center, Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University, Bangkok, Thailand

ABSTRACT

Tremendous quantities of numeric data have been generated as streams in various cyber ecosystems. Sorting is one of the most fundamental operations to gain knowledge from data. However, due to size restrictions of data storage which includes storage inside and outside CPU with respect to the massive streaming data sources, data can obviously overflow the storage. Consequently, all classic sorting algorithms of the past are incapable of obtaining a correct sorted sequence because data to be sorted cannot be totally stored in the data storage. This paper proposes a new sorting algorithm called *streaming data sort* for streaming data on a uniprocessor constrained by a limited storage size and the correctness of the sorted order. Data continuously flow into the storage as consecutive chunks with chunk sizes less than the storage size. A theoretical analysis of the space bound and the time complexity is provided. The sorting time complexity is $O(n)$, where n is the number of incoming data. The space complexity is $O(M)$, where M is the storage size. The experimental results show that *streaming data sort* can handle a million permuted data by using a storage whose size is set as low as 35% of the data size. This proposed concept can be practically applied to various applications in different fields where the data always overflow the working storage and sorting process is needed.

Subjects Algorithms and Analysis of Algorithms, Artificial Intelligence, Data Science
Keywords Algorithms, Sorting, Memory, Algorithm design and analysis, Computational intelligence

Submitted 26 October 2020
Accepted 17 December 2020
Published 12 February 2021

Corresponding authors
Suluk Chaikhan, suluk.c@student.chula.ac.th
Suphakant Phimoltares, suphakant.p@chula.ac.th

Academic editor
Marcin Woźniak

Additional Information and Declarations can be found on page 24

DOI 10.7717/peerj-cs.355

INTRODUCTIONS

Currently, the growth of data consumption by internet users has exponentially increased (*Laga et al., 2017*; *Bey Ahmed Khernache, Laga & Boukhobza, 2018*), and a massive storage size is required to store all incoming data to avoid any data loss in case of storage overflow (*Thusoo et al., 2010*; *Katal, Wazid & Goudar, 2013*; *Witayangkurn, Horanont & Shibasaki, 2012*; *Mehmood et al., 2016*). However, many applications such as data management, finance, sensor networks, security-relevant data, and web search possibly face this unexpected situation of a storage overload issue (*Lee et al., 2016*; *Babcock et al., 2002*; *Keim, Qu & Ma, 2013*; *Cardenas, Manadhata & Rajan, 2013*; *Dave & Gianey, 2016*). This issue induces the problem of representing big data with a limited storage size. Furthermore, some primitive operations such as the classic sorting algorithms (e.g., quick sort, heap sort) cannot be implemented due to the restrictive constraint of storing all sorted data inside the storage during the sorting process. A sorting algorithm is the first important step of many algorithms (*Cormen et al., 2009*; *Huang, Liu & Li, 2019*) such as searching and finding a closest pair (*Singh & Sarmah, 2015*; *Tambouratzis, 1999*).

Generally, when referring to data storage of a computer, it can be either primary storage (internal storage) or secondary storage (external storage). The size of primary storage is much smaller than that of the secondary storage. With reference to the size of storage, there are two types of sorting: internal sort and external sort. All data to be sorted by an internal sorting algorithm must be entirely stored inside the primary storage. Some of traditional internal sorting algorithms are bubble sort, insertion sort, quick sort, merge sort, and radix sort. However, if the data overflow the primary storage, the overflow must be stored in the secondary storage. In this case, external sort algorithms can be employed. Although these classic sorting algorithms are very efficient in terms of time and space complexities, the actual quantity of data generated yearly on the internet has grown tremendously faster than the growth rate of storage capacity based on the current fabrication technology (for both primary storage and secondary storage). This severe condition makes the classic sorting algorithms, where all data must be stored inside the computer, very inefficient because all overflowed data are lost.

In this study, both internal and external storage are viewed as one unit of storage with a limited size. This size is not gradually augmented during the sorting process of continuously incoming data. The challenging problem to be studied is how to sort the data under the constraints of limited storage capacity and storage overflow. The data are assumed to flow into the storage as a sequence of data chunks with various sizes less than or equal to the storage size.

Recently, many internal sorting algorithms have been remodeled by reducing comparison, swapping, and the time complexity to reduce the sorting time. *Farnoud, Yaakobi & Bruck (2016)* proposed an algorithm that sorts big data based on limited internal storage, but the result is a partial sort. Concoms sort (*Agrawal & Sriram, 2015*) is an algorithm that uses a swapping technique with no adjacent swapping. It reduces the execution time in some cases when compared to selection sort and outperforms bubble sort in every case. In particular, in the case that the input is a descending sequence, Concoms sort is more efficient than both traditional algorithms. Mapping sort (*Osama, Omar & Badr, 2016*) is a new algorithm that does not use comparisons and the swapping technique but it uses the mapping technique instead. This algorithm achieved the worst case time complexity of $O(n) + O(n \log n)$. *Vignesh & Pradhan (2016)* proposed a sorting algorithm by improving merge sort. It uses multiple pivots to sort data. The execution time of this algorithm is better than quick sort and merge sort in the best case and the average case, respectively. In addition, proximity merge sort (*Franceschini, 2004*) was proposed by improving the algorithm with an in-place property. *Faro, Marino & Scafiti (2020)* modified insertion sort to reduce the time complacency by inserting multiple elements for one iteration. The time complexity is $O(n^{1+\frac{1}{h}})$, where $h \in \mathbb{N}$. *Idrizi, Rustemi & Dalipi (2017)* modified the sorting algorithm by separating the data sequence into three parts, namely, negative numbers, zero numbers, and positive numbers. After the data in each part are sorted by printing the result, the algorithm can decrease the comparison by a separating process. Bidirectional conditional insertion sort algorithm (*Mohammed, Amrahov & Çelebi, 2017*) is a two-pivot insertion sort algorithm using the left comparator and right comparator. It is faster than insertion sort, and the time complexity is nearly close to $O(n^{1.5})$. Brownian motus and clustered binary insertion sort methods (*Goel & Kumar, 2018*) are algorithms that adapted insertion sort and binary insertion sort to reduce the comparison and the execution time. Both algorithms are suitable for sorting partial data. Internal sorting algorithms in the literature have focused on reducing the time for processing, but the storage issue for big data has been ignored.

Presently, accessing a large piece of information or big data is simple because of rapid technological advancements such as the cloud (*Al-Fuqaha et al., 2015*; *Kehoe et al., 2015*; *Vatrapu et al., 2016*) and network technology (*YiLiang & Zhenghong, 2016*; *Zhao, Chang & Liu, 2017*; *Zhai, Zhang & Hu, 2018*). One of the issues for sorting big data is the restricted internal storage, which is usually smaller than the size of big data. All big data cannot be stored in the internal storage. Therefore, the internal sorting algorithms cannot sort big data at one time. The external sorting algorithms are developed from the classic merge sorting algorithm to sort big data, which is separated into two phases: (1) the sorting phase sorts a small chunk of big data in the internal storage. After sorting, all sorted chunks are stored in the external storage and (2) the merging phase combines all sorted chunks from the sorting phase into a single sorted list.

Recently, TaraByte sort (*O'Malley, 2008*) has used three Hadoop applications, namely, TeraGen, TeraSort, and TeraValidate, to sort big data. This algorithm sorts 10 billion data in 209 s. This process is very fast, but it is expensive because it requires many processing units for sorting. *Kanza & Yaari (2016)* studied external sorting problems and designed multi-insertion sort and SCS-Merge V1 to V3. The objective of these algorithms is to decrease the write cost of intermediate results of sorting. Active sort (*Gantz & Reinsel, 2012*) is an algorithm that merges sorted chunks inside SSDs and is applied with Hadoop to reduce the number of reading and writing data. MONTRES (*Laga et al., 2017*), the algorithm designed for SSDs, can reduce the read and write cost of I/O in a linear function. External sorting algorithms in the literature have focused on reducing the read and write cost in terms of the execution time for processing, but the storage issue for keeping big data is still ignored. *Liang et al. (2020)* proposed a new algorithm, namely, B*-sort, which was designed on NVRAM and applied on a binary search tree structure.

In addition, *Farnoud, Yaakobi & Bruck (2016)* studied approximate sorting of streaming permuted data with limited storage; however, the result is not exactly sorted data, and only an approximate result is obtained when determining the data positions. Conversely, the approximate positions of ordered data can be provided when using the values as inputs. *Elder & Goh (2018)* studied permutation sorting by finite and infinite stacks. Although all possible permutations cannot be sorted, the exact order and values can be obtained. Let n be the total streaming numbers to be sorted and $M \ll n$ be the limited size of working

Table 1 **Comparison of sorting algorithms on streaming data. n is the total streaming numbers to be sorted and $M \ll n$ is the limited size of working storage.**

Sorting algorithms	Requiring extra storage	Preserving input appearance order	Time complexity	Working space complexity	Applicable to streaming data	Correct order	Correct value
Bubble sort	No	Yes	$O(n^2)$	$O(n)$	No	Yes	Yes
Selection sort	No	No	$O(n^2)$	$O(n)$	No	Yes	Yes
Insertion sort	No	Yes	$O(n^2)$	$O(n)$	Yes	Yes	Yes
Quick sort	No	No	$O(n^2)$	$O(n)$	No	Yes	Yes
Merge sort	Yes	Yes	$O(n \lg n)$	$O(n)$	No	Yes	Yes
Heap sort	No	No	$O(n \lg n)$	$O(n)$	No	Yes	Yes
Permutation sort (*Farnoud, Yaakobi & Bruck, 2016*)	No	No	$O(n/\omega(\log^2 n))$	$O(n)$	No	Yes	Yes
Permutation sort (*Elder & Goh, 2018*)	Yes	Yes	N/A	$O(n)$	Yes	No	No
External sorting	Yes	Yes	N/A	$O(n)$	Yes	Yes	Yes
Streaming data sort	No	Yes	$O(n)$	$O(M)$	Yes	Yes	Yes

storage. Table 1 summarizes the efficiency of various classic sorting methods and our proposed method (stream sort) in terms of seven characteristics: requiring extra storage, preserving the input appearance order, time complexity, space complexity, sorting streaming data, correct sorting order, and correct retrieved value by the sorted order.

This article proposes a new algorithm called *streaming data sort* for sorting streaming data with limited storage size by using only a single central processing unit. The proposed algorithm can correctly and stably handle a streaming data size of at least 2.857 times larger than the size of the working storage. The following concerns are emphasized in this study.

- All data must be in the exactly correct order after being sorted. No approximate and partial ordering is allowed in this study.
- The time complexity of *streaming data sort* of all iterations is $O(n)$.

CONSTRAINTS

In the stationary data environment, all classic sorting algorithms are based on the assumption that all numbers to be sorted must be entirely stored in the working storage of a computer during the sorting process. This implies that the whole data set cannot exceed the working storage size during the sorting process. Figure 1 illustrates storage constraint of the working storage in *streaming data sort*. However, in the streaming data environment, the data continuously flow into the computer one chunk at a time, and the number of incoming chunks is unknown in advance. If the size of the data chunk is larger than the working storage size, then the overflow will be permanently discarded from the computer. This makes the sorted result wrong. To make the study sufficiently feasible for analysis and practice, the following constraints are imposed.

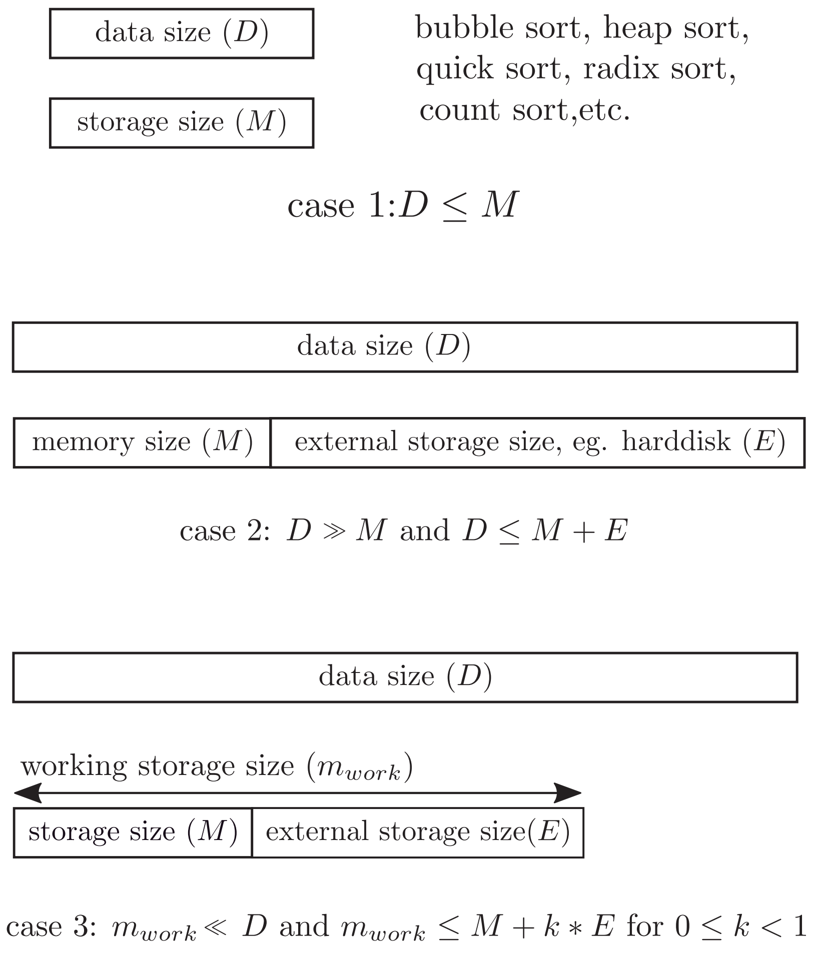
data size (D)
storage size (M)
bubble sort, heap sort, quick sort, radix sort, count sort,etc.
case 1:$D \leq M$
data size (D)
memory size (M)
external storage size, eg. harddisk (E)
case 2: $D \gg M$ and $D \leq M + E$
data size (D)
working storage size (m_{work})
storage size (M)
external storage size(E)
case 3: $m_{work} \ll D$ and $m_{work} \leq M + k * E$ for $0 \leq k < 1$

Figure 1 **Storage constraint. Case 1 for $D \leq M$ where all data must be in the storage. Case 2 for $D \gg M$ and $D \leq M + E$ where data overflow the storage. Case 3 for $m_{work} = M + E$, the constraint of this study.** Full-size DOI: 10.7717/peerj-cs.355/fig-1

1. The sorting process is performed by using only a fixed working storage of size M. This working storage is for storing the incoming data, previously sorted data, and other temporal data structures generated during the sorting process. No extra storage module is added during this period. The proposed sorting algorithm and the operating system are not stored in this working storage.
2. All numbers are integers. For floating numbers, they must first be transformed into integers.
3. At any time t, the sizes of previously sorted data in a compact form and the size of next incoming data chunk (h) must not exceed M.
4. The present incoming data chunk is completely discarded after being processed by the proposed sorting algorithm.
5. Only four types of relation between any two temporal consecutive numbers d_i and d_{i+1} are studied in this paper. The details and rationale of concentrating on these four types will be elaborated later.

The second constraint is the main concern of this study. After sorting the first incoming data chunk, all numbers are captured in a compact form and all sorted numbers are completely discarded. This compact form is used in conjunction with the next incoming data chunk for sorting. To avoid a storage overflow obstruction, the fourth constraint must be presented. The last constraint is derived from real-world data sets. From the observation of real-world streaming data sets from the UCI Repository (*Dua & Graff, 2019*) such as the census income, diabetes 130-US hospitals, incident management process event log, PM2.5 of five Chinese cities, KEGG metabolic relation network, Beijing multi-site air quality, and Buzz in social media, it is remarkable that most of the different values between two temporal consecutive numbers are between 0.38 and 2.98 on average. Hence, only four types of relations between any two temporal consecutive numbers are the focus. The definition of each type will be given in the next section.

DEFINITIONS AND NOTATIONS

Definition 1 *The window at time t, denoted by* $\mathcal{W}^{(t)} = (d_1, d_2, \ldots, d_h)$*, is a sequence of* $h \leq M$ *incoming numeric data at time t.*

Definition 2 *The sorted window of* $\mathcal{W}^{(t)}$ *at time t, denoted by* $W^{(t)} = (w_1, w_2, \ldots, w_h \mid w_i = d_j, w_{i+1} = d_k \text{ and } \forall w_i, w_{i+1} \in W^{(t)} : w_i < w_{i+1})$*, is a sequence of increasingly sorted numeric data of* $\mathcal{W}^{(t)}$.

Definition 3 Type-1 *subsequence* $T_1 = (w_i, \ldots, w_{i+l}) \subseteq W^{(t)}$ *is a sequence such that* $\forall\, w_i, w_{i+1} \in T_1$: $|w_i - w_{i+1}| = 1$.

An example of a Type-1 sequence is (1, 2, 3, 4, 5). The different value between any two adjacent numbers is equal to 1, namely, (|1 − 2|,|2 − 3|,|3 − 4|,|4 − 5|) = (1,1,1,1).

Definition 4 Type-2 *subsequence* $T_2 = (w_i, \ldots, w_{i+l}) \subseteq W^{(t)}$ *is a sequence such that* $\forall w_{i+a}, w_{i+a+1} \in T_2, 0 \leq a \leq l-1 : |w_{i+a} - w_{i+a+1}| = 1$ *when a is even and* $|w_{i+a} - w_{i+a+1}| = 2$ *when a is odd.*

An example of a Type-2 sequence is (4, 5, 7, 8, 10). The different value between any two adjacent numbers is equal to either 1 or 2, namely, (|4–5|, |5–7|, |7–8|, |8–10|) = (1, 2, 1, 2).

Definition 5 Type-3 *subsequence* $T_3 = (w_i, \ldots, w_{i+l}) \subseteq W^{(t)}$ *is a sequence such that* $\forall w_{i+a}, w_{i+a+1} \in T_3, 0 \leq a \leq l-1 : |w_{i+a} - w_{i+a+1}| = 2$ *when a is even and* $|w_{i+a} - w_{i+a+1}| = 1$ *when a is odd.*

An example of a Type-3 sequence is (5, 7, 8, 10, 11). The different value between any two adjacent numbers is equal to either 1 or 2, namely, (|5–7|, |7–8|, |8–10|, |10–11|) = (2, 1, 2, 1).

Definition 6 Type-4 *subsequence* $T_4 = (w_i, \ldots, w_{i+l}) \subseteq W^{(t)}$ *is a sequence such that* $\forall w_i, w_{i+1} \in T_4$: $|w_i - w_{i+1}| = 2$.

An example of a Type-4 sequence is (8, 10, 12, 14, 16). The different value between any two adjacent numbers is equal to either 1 or 2, namely, (|8–10|, |10–12|, |12–14|, |14–16|) = (2, 2, 2, 2).

During the sorting process by the proposed algorithm, it is necessary to identify the type of subsequence to be sorted first. Given a subsequence $(w_i, \ldots, w_{i+l}) \in W^{(t)}$, the type of this subsequence can be easily identified as type-p by setting

$$p = w_{i+2} + w_{i+1} - 2(w_i + 1) \tag{1}$$

Each already sorted subsequence $(w_i, \ldots, w_{i+l}) \in W^{(t)}$ is compactly written in a form of $(u, v)^{(p)}$ where $u = w_i$ and $v = w_{i+l}$ are used during the sorting process to minimize the storage use. $(u, v)^{(p)}$ is named *compact group* p. Any numeric data in between u and v are called *removed data*. These *removed data* are not considered and can be removed after the sorting process. For example, subsequence (1, 2, 3, 4, 5) is compacted as $(1, 5)^{(1)}$; (4, 5, 7, 8, 10) is compacted as $(4, 10)^{(2)}$; (5, 7, 8, 10) is compacted as $(5, 10)^{(3)}$; and (8, 10, 12, 14) is compacted as $(8, 14)^{(4)}$.

Note that a sequence $W^{(t)}$ may contain several compact groups and some single numbers. Suppose $W^{(t)}$ = (1, 2, 3, 4, 5, 7, 8, 10, 12, 14, 19). This sequence consists of the following subsequences $(1, 5)^{(1)}$, $(8, 14)^{(4)}$. Thus, $W^{(t)}$ can be rewritten in another form of compact groups and a set of single numbers as $W^{(t)} = ((1, 5)^{(1)}, 7, (8, 14)^{(4)}, 19)$. However, it is possible to find another set of compact groups from $W^{(t)}$ as $W^{(t)} = (((1,3)^{(1)}, (4,7)^{(2)}, (8,14)^{(4)}, 19)$. Obviously, different sets of compact groups for any $W^{(t)}$ use different storage sizes to store them.

To distinguish between $W^{(t)}$ written in the original sequence of numbers and $W^{(t)}$ written in a form of compact groups having a set of single numbers, the notation $Q^{(t)}$ is used instead of $W^{(t)}$ to denote a combination set of compact groups and single numbers. Each compact group i in $Q^{(t)}$ is denoted by q_i. In fact, either each compact group or a single number in $Q^{(t)}$ can be considered as an element of $Q^{(t)}$. For example, if $W^{(t)}$ = (1, 2, 3, 4, 5, 7, 8, 10, 12, 14, 19), then $Q^{(t)} = ((1, 5)^{(1)}, 7, (8, 14)^{(4)}, 19)$ such that $q_1 = (1, 5)^{(1)}$, $q_2 = (8, 14)^{(4)}$. All *removed data* of compact group $(u, v)^{(p)}$ will be occasionally retrieved to obtain a complete sorted subsequence in order to involve the new incoming subsequence in the sorting process. Hence, each retrieved number is denoted by r_i to make it different from each input number w_i during the sorting process. The retrieved sequence of $(u, v)^{(p)}$, denoted $R((u, v)^{(p)})$, can be obtained by using the following rules.

$$r_1 = u \tag{2}$$

$$r_{1+l} = v \tag{3}$$

$$r_{i+1} = \begin{cases} r_i + 1 & \text{for } p = 1 \\ r_i + (r_i - r_1 + p - 1) \bmod 3 & \text{for } p = 2, 3 \\ r_i + 2 & \text{for } p = 4 \end{cases} \tag{4}$$

To illustrate how to retrieve all numbers from a compact group, consider an example of sequence (5, 7, 8, 10) represented by the compact group $(5, 10)^{(3)}$. The retrieved numbers of $(5, 10)^{(3)}$ can be computed as follows: Since $p = 3$, $r_1 = 5$, $r_2 = (5) + ((5) - (5) + (3) - 1) \bmod 3 = 7$, $r_3 = (7) + ((7) - (5) + (3) - 1) \bmod 3 = 8$, $r_4 = (8) + ((8) - (5) + (3) - 1) \bmod 3 = 10$, $r_4 = 10 = v$. Accordingly, $R\left((5, 10)^{(3)}\right) = (5, 7, 8, 10)$.

CONCEPTS

The size of each incoming sequence is assumed to be at most the size of working storage. To make the working storage available for storing the next incoming data chunk after sorting the current chunk, it is required to represent some sorted subsequent numbers in a form of a *compact group*. However, not all subsequent sorted numbers can be compacted.

The properties and concept of a compact group representation will be discussed next. The sorting process consists of the following three main steps.

1. Transform the initial incoming sequence $W^{(1)}$ into a set of compact groups $q_i \in Q^{(1)}$.
2. At time t, obtain the next incoming sequence and insert each number $w_i \in W^{(t)}$ into the previous $Q^{(t-1)}$ at the appropriate position.
3. If there exist any adjacent compact groups $q_i = (a,b)^{(\alpha)}$ and $q_{i+1} = (c,d)^{(\beta)}$ such that the retrieved sequences $R((a,b)^{(\alpha)})$ and $R((c,d)^{(\beta)})$ satisfy one of the types of subsequences, then form a new compact group from the sequences of $R((a,b)^{(\alpha)})$ and $R((c,d)^{(\beta)})$.

Steps 2 and 3 are iterated until there are no more incoming sequences. The details of each step will be discussed next. Figure 2 shows an example of how the proposed approximate sorting works. The storage size $|m_{\text{tot}}|$ is 10. The first incoming 10-number sequence, that is, (18, 1, 10, 6, 2, 12, 9, 3, 16, 19), fills the whole storage. This sequence is sorted in an ascending order and forms a set $Q^{(1)} = ((1, 3)^{(1)}, 6, (9, 12)^{(2)}, (16, 19)^{(3)})$, as shown in Fig. 2A. The size of the storage used is decreased to 7. The second incoming sequence (14, 11, 17) is inserted into some compact groups in $Q^{(1)}$ to obtain $Q^{(2)} = ((1, 3)^{(1)}, 6, (9, 12)^{(1)}, 14, (16, 19)^{(1)})$, as shown in Fig. 2B. The size of the storage used after the second incoming sequence is increased to 8. The third incoming sequence (13, 20) is separately grouped with $(9, 12)^{(1)}$ and $(16, 19)^{(1)}$ from the previous $Q^{(2)}$ to make a new $Q^{(3)} = ((1, 3)^{(1)}, 6, (9, 13)^{(1)}, 14, (16, 20)^{(1)})$. Observe that the compact group $(9, 13)^{(1)}$ can be grouped with the single number 14 to make $(9, 14)^{(1)}$. Therefore, $Q^{(3)} = ((1, 3)^{(1)}, 6, (9, 14)^{(1)}, (16, 20)^{(1)})$. The fourth incoming sequence (8, 4, 15) is possibly and separately grouped with $(9, 14)^{(1)}$, $(1, 3)^{(1)}$, and $(16, 19)^{(1)}$ in $Q^{(3)}$ to obtain $Q^{(4)} = ((1, 4)^{(1)}, 6, (8, 20)^{(1)})$. The last incoming sequence (5, 7) is possibly and separately grouped with $(1, 4)^{(1)}$, 6, $(8, 20)^{(1)}$ in $Q^{(4)}$ to obtain $Q^{(5)} = ((1, 20)^{(1)})$.

PROPOSED ALGORITHM

The proposed sorting algorithm is composed of the following two major steps. These steps are based on the constraints previously imposed in Constraints Section.

1. Obtain the first input number sequence and sort the number in an ascending order. Then, create $Q^{(1)}$, a set of compact groups and a set of a single number.
2. At time t, obtain the next set of number sequences and insert the numbers into $Q^{(t-1)}$ to create the next $Q^{(t)}$.
3. Repeat step 2 until there are no more new incoming sequences.

The deils of steps 1 and 2 will be discussed in the following sections.

Creating compact groups

There are four types of compact groups. To identify the type of compact group from a number sequence, four counters c_1, c_2, c_3, and c_4 for type-1, type-2, type-3, and type-4, respectively, are employed. Let $s^{(i)}$ be the status condition of type-i. The value of $s^{(i)}$ is defined as follows.

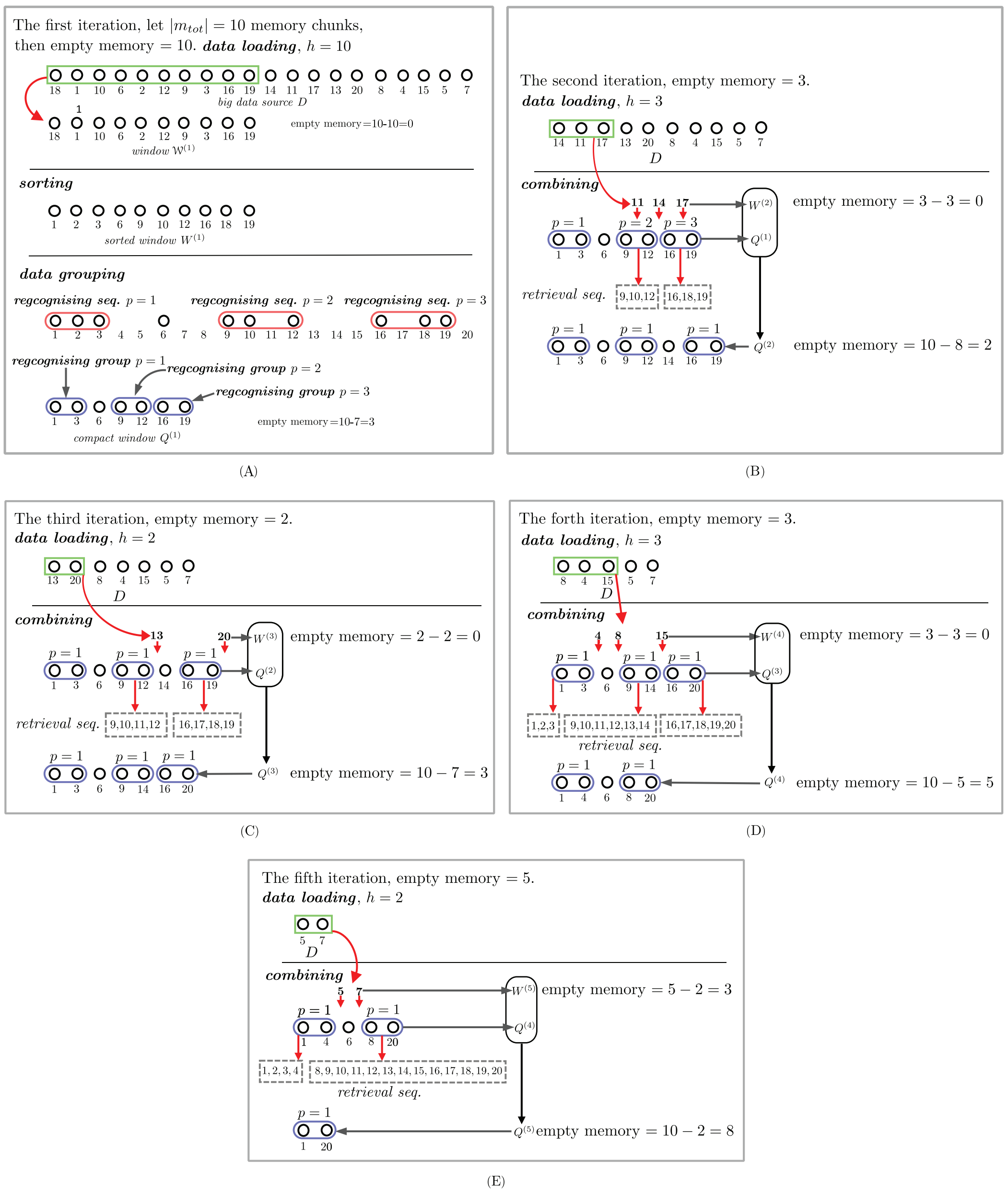

Figure 2 **An example of streaming data sort. The sorting steps are illustrated in subfigures (A), (B), (C), (D) and (E).** Full-size DOI: 10.7717/peerj-cs.355/fig-2

Definition 7 *Type-1 status condition $s^{(1)}$ of a datum w_{k+i} and its neighbors in a type-1 subsequence $w_k,\ldots,w_{k+i},\ldots,w_{k+m}$, where $m < h$ is a constant defined by:*

$$s^{(1)} = \begin{cases} 1 & w_{k+i} - w_{k+i-1} = 1 \quad \text{for} \quad 0 \le i \le m \\ 0 & \text{otherwise.} \end{cases}$$

Definition 8 *Type-2 status condition $s^{(2)}$ of a datum w_{k+i} and its neighbors in a type-2 subsequence $w_k,\ldots,w_{k+i},\ldots,w_{k+m}$, where $m < h$ is a constant defined by:*

$$s^{(2)} = \begin{cases} 1 & (w_{k+i-1} - w_k) \bmod 3 + 1 = w_{k+i} - w_{k+i-1} \quad \text{for} \quad 0 \le i \le m \\ 0 & \text{otherwise.} \end{cases}$$

Table 2 **Notations in streaming data sort.**

Notations	Short definitions	Examples
d_i	The i^{th} incoming datum	−3, 0, 10
$(d_1,d_2,d_3,\ldots)$	Sequence of streaming data	(−3, 0, 10,…)
h	Window size at iteration t	5, 0, 4, 1
w_i	The i^{th} member in a window	18, 1, 10
$\mathcal{W}^{(t)}$	Unsorted window at iteration t	(18, 1, 10, 6,…)
$\mathcal{W}^{(t)}$	Sorted window at iteration t	(1, 6, 10, 18,…)
p	Type of sub-sequence	1, 2, 3, 4
T_p	Type-p sub-sequence	(2, 4, 6, 8, 10, 12)
$(u, v)^{(p)}$	Type-p compact group	$(2, 12)^{(3)}$
r_i	The i^{th} retrieved number	2, 4, 6, 8, 10, 12
$R((u, v)^{(p)})$	Retrieved sequence of $(u, v)^{(p)}$	(2, 4, 6, 8, 10, 12)
$s^{(i)}$	Status condition of type-i	0, 1
q_i	The i^{th} compact group	$(1, 5)^{(4)}$, $(9, 12)^{(1)}$
S	Set of single numbers	{6, 7}
C	Set of compact groups	$\{(1, 5)^{(4)}, (9, 12)^{(1)}\}$
$Q^{(t)}$	Combining set of C and S at iteration t	$\{(1, 5)^{(4)}, 6, 7, (9, 12)^{(1)}\}$

Definition 9 *Type-3 status condition $s^{(3)}$ of a datum w_{k+i} and its neighbors in a type-3 subsequence $w_k,\ldots,w_{k+i},\ldots,w_{k+m}$, where $m<h$ is a constant defined by:*

$$s^{(3)} = \begin{cases} 1 & (w_{k+i-1} - w_k + 2) \bmod 3 = w_{k+i} - w_{k+i-1} \quad \text{for} \quad 0 \le i \le m \\ 0 & \text{otherwise.} \end{cases}$$

Definition 10 *Type-4 status condition $s^{(4)}$ of a datum w_{k+i} and its neighbors in a type-4 subsequence $w_k,\ldots,w_{k+i},\ldots,w_{k+m}$, where $m < h$ is a constant defined by:*

$$s^{(4)} = \begin{cases} 1 & w_{k+i} - w_{k+i-1} = 2 \quad \text{for} \quad 0 \le i \le m \\ 0 & \text{otherwise.} \end{cases}$$

The notations in this paper are given in Table 2.

$Q^{(t)}||S||\, C$ denotes orderly concatenating $Q(t)$, S, C according to the sorted order of all elements in $W^{(t)}$. The quantity of removed data of type-1 is greater than those of the other types. The difference of the first and the last data of a type-4 compact group is larger than the differences in the other types. To greatly reduce and control the storage size, the sequences of types 1 and 4 are detected before the sequences of types 2 and 3. Suppose the following sequence (1, 3, 4, 5, 6) is given. If types 1 and 4 are considered before types 2 and 3, then the given sequence is compacted as 1, $(3, 6)^{(1)}$, which requires 4 units of storage to store numbers 1, 3, 6, 1. However, if types 2 and 3 are considered before types 1 and 4, then the given sequence is compacted as $(1, 4)^{(2)}$, 5, 6, which requires 5 units of storage to store numbers 1, 4, 2, 5, 6.

Theorem 1 *If $p = \arg \max_{1 \le i \le 4} (c_i)$, then p denotes the correct type of the compact group.*

Proof: Suppose the sorted sequence is $W^{(t)} = (w_1, w_2, \ldots, w_h)$. We consider each type of compact group. Let $s_t^{(i)}$ be the status condition of type-i at time t and $T^{(i)} = (s_1^{(i)}, s_2^{(i)}, \ldots, s_h^{(i)})$ be the sequence of $s_t^{(i)}$. There are four cases to be investigated.

Case 1 (type-1): Suppose the sorted sequence $W^{(t)} = (w_1, w_2, \ldots, w_h)$ is in type-1. Then, we have the following four sequences of the status condition.

$$T^{(1)} = \left(s_1^{(1)} = 0, s_2^{(1)} = 0, s_3^{(1)} = 1, \ldots, s_h^{(1)} = 1\right) \text{ or } (0, 0, 1, \ldots, 1)$$
$$T^{(2)} = \left(s_1^{(2)} = 0, s_2^{(2)} = 0, s_3^{(2)} = 0, \ldots, s_h^{(2)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(3)} = \left(s_1^{(3)} = 0, s_2^{(3)} = 0, s_3^{(3)} = 0, \ldots, s_h^{(3)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(4)} = \left(s_1^{(4)} = 0, s_2^{(4)} = 0, s_3^{(4)} = 0, \ldots, s_h^{(4)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$

Obviously, the value of $c_1 = \sum_{t=1}^{h} s_t^{(1)}$ is larger than that of c_2, c_3, and c_4.

Case 2 (type-2): Suppose the sorted sequence $W^{(t)} = (w_1, w_2, \ldots, w_h)$ is in type-2. Then, we have the following four sequences of the status condition.

$$T^{(1)} = \left(s_1^{(1)} = 0, s_2^{(1)} = 0, s_3^{(1)} = 0, \ldots, s_h^{(1)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(2)} = \left(s_1^{(2)} = 0, s_2^{(2)} = 0, s_3^{(2)} = 1, \ldots, s_h^{(2)} = 1\right) \text{ or } (0, 0, 1, \ldots, 1)$$
$$T^{(3)} = \left(s_1^{(3)} = 0, s_2^{(3)} = 0, s_3^{(3)} = 0, \ldots, s_h^{(3)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(4)} = \left(s_1^{(4)} = 0, s_2^{(4)} = 0, s_3^{(4)} = 0, \ldots, s_h^{(4)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$

Obviously, the value of $c_2 = \sum_{t=1}^{h} s_t^{(2)}$ is larger than that of c_1, c_3, and c_4.

Case 3 (type-3): Suppose the sorted sequence $W^{(t)} = (w_1, w_2, \ldots, w_h)$ is in type-3. Then, we have the following four sequences of the status condition.

$$T^{(1)} = \left(s_1^{(1)} = 0, s_2^{(1)} = 0, s_3^{(1)} = 0, \ldots, s_h^{(1)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(2)} = \left(s_1^{(2)} = 0, s_2^{(2)} = 0, s_3^{(2)} = 0, \ldots, s_h^{(2)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(3)} = \left(s_1^{(3)} = 0, s_2^{(3)} = 0, s_3^{(3)} = 1, \ldots, s_h^{(3)} = 1\right) \text{ or } (0, 0, 1, \ldots, 1)$$
$$T^{(4)} = \left(s_1^{(4)} = 0, s_2^{(4)} = 0, s_3^{(4)} = 0, \ldots, s_h^{(4)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$

Obviously, the value of $c_3 = \sum_{t=1}^{h} s_t^{(3)}$ is larger than that of c_1, c_2, and c_4.

Case 4 (type-4): Suppose the sorted sequence $W^{(t)} = (w_1, w_2, \ldots, w_h)$ is in type-4. Then, we have the following four sequences of the status condition.

$$T^{(1)} = \left(s_1^{(1)} = 0, s_2^{(1)} = 0, s_3^{(1)} = 0, \ldots, s_h^{(1)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(2)} = \left(s_1^{(2)} = 0, s_2^{(2)} = 0, s_3^{(2)} = 0, \ldots, s_h^{(2)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(3)} = \left(s_1^{(3)} = 0, s_2^{(3)} = 2, s_3^{(3)} = 0, \ldots, s_h^{(3)} = 0\right) \text{ or } (0, 0, 0, \ldots, 0)$$
$$T^{(4)} = \left(s_1^{(4)} = 0, s_2^{(4)} = 0, s_3^{(4)} = 1, \ldots, s_h^{(4)} = 1\right) \text{ or } (0, 0, 1, \ldots, 1)$$

Obviously, the value of $c_4 = \sum_{t=1}^{h} s_t^{(4)}$ is larger than that of c_1, c_2, and c_3.■

Inserting numbers into the combination set of compact groups

After creating the first combination set of compact group $Q^{(1)}$ and obtaining a new incoming sequence, the current compact groups must be updated according to the number in the incoming sequence. There are seven possible cases where a new incoming number can be inserted into any compact group or in between a compact group and a single number. Let the α new incoming number d_α is located according to each case as follows. $Q^{(t)}$ is the set of combinations of compact groups and a set of single numbers at time t.

Case 1: d_α is at the front of $Q^{(t)}$. *Case 2*: d_α is at the rear of $Q^{(t)}$. *Case 3*: d_α is in a compact group $(u_j, v_j)^{(p)}$. *Case 4*: d_α is in between two compact groups $(u_j, v_j)^{(p_j)}$ and $(u_k, v_k)^{(p_k)}$. *Case 5*: d_α is in between a single number w_j and a compact group $(u_k, v_k)^{(p_k)}$. *Case 6*: d_α is in between a compact group $(u_j, v_j)^{(p_j)}$ and a single number w_k. *Case 7*: d_α is in between two single numbers w_j and w_{j+1}.

The details of each case and the insertion steps are in given in Algorithm 2.

EXPERIMENTAL RESULTS AND DISCUSSION

Three issues are discussed in this section. The first issue illustrates the snapshot of sorting outcomes as the results of incoming data chunks, current compact groups of different types, and sets of single numbers. The second issue discusses the relation between the sorting time and the number of streaming numbers. The third issue shows how the size of working storage changes during the sorting process.

Sorting examples

The proposed algorithms are implemented in MATLAB R2016a. The computing results are run on 3.4 GHz Intel Core i7 6700 and 16 GB of 2400 MHz RAM with the Windows 10 platform. To illustrate how the proposed algorithm works, three experiments were conducted by using a set of 100 single integers ranging from 1 to 100. These 100 numbers were randomly permuted to produce three different experimental data sets. The total size of storage is assumed to have only 60 working addresses. Forty of them are used for storing temporary data generated during the sorting process, which includes $\mathcal{W}^{(t)}$, $W^{(t)}$, and $Q^{(t)}$ at different times. The rest of storage is for storing some variables in the sorting program.

To illustrate the continuous results during the sorting process, three data sets in the experiment were generated from three permutations of integer numbers from 1 to 100 to avoid any duplication. These permuted numbers are sliced into a set of input chunks of at most 40 numbers in each chunk. Let $|m_{tot}|$ be the total size of the working storage, which is equal to 60 in the experiment. The experimental results are shown in Fig. 3, where the x-axis represents each w_i in $W^{(t)}$ and $Q^{(t)}$ and the y-axis represents the time line of iterations. Each datum w_i is represented by $\times$. Each type of compact group in $Q^{(t)}$ is denoted by a solid line with a specific color as follows.

Type-1 is denoted by gray line.

Type-2 is denoted by blue line.

Type-3 is denoted by green line.

Type-4 is denoted by red line.

Algorithm 1 **Creating compact groups.**

```
Input: a sorted sequence $W^{(t)} = (w_k, w_{k+1}, \ldots, w_{k+h})$ of length $h$ at time $t$.
Output: a combination of a set of compact groups and a set of single numbers.
1. $j = k$.
2. $S = \emptyset$. /* set of single numbers */
3. $C = \emptyset$. /* set of compact groups */
4. $Q^{(t)} = \emptyset$.
5. For $l = 1$ to 2 do /* packing order types 1, 4 before 2, 3 */
6.     $c_1 = c_2 = c_3 = c_4 = 0$.
7.     If $|w_k - w_{k+1}| > 2$ then
8.         $S = S \cup \{w_k\}$.
9.         $j = k+1$.
10.    EndIf
11.    For $i = k+1$ to $k+h-1$ do
12.        If $|w_i - w_{i+1}| \leq 2$ then
13.            Set the values of $s^{(1)}, s^{(2)}, s^{(3)}, s^{(4)}$ by definitions 7–10.
14.            $c_l = c_l + s^{(l)}$.
15.            $c_{5-l} = c_{5-l} + s^{(5-l)}$.
16.        else
17.            If $j = i$ then
18.                $S = S \cup \{w_j\}$. /* single number */
19.                $j = i + 1$.
20.            else
21.                $p = arg\ \max_{i \in \{l, 5-l\}} (c_i)$. /* compact types */
22.                Create a compact group $(w_j, w_i)^{(p)}$.
23.                $C = C \cup \{(w_j, w_i)^{(p)}\}$.
24.                $c_1 = c_2 = c_3 = c_4 = 0$.
25.                $j = i + 1$.
26.            EndIf
27.        EndIf
28.        $Q^{(t)} = Q^{(t)} || S || C$.
29.    EndFor
30. EndFor
```

A single number in the compact window is represented by •.

In the first data set, there are 40 numbers entering the process at the starting time. After being grouped by Algorithm 1, the result appears in time step $t = 1$ (at the line above the bottom line in Fig. 3A. There are four compact groups of type-1, two compact groups of type-2, three compact groups of type-4, and nine single numbers. Also, at time $t = 1$, there are four new incoming numbers, each of which is represented by ×. Algorithm 1 sorts

Algorithm 2 Inserting d_α into $Q(t)$.

```
Input: (1) set $Q^{(t)}$. (2) a new number $d_\alpha$.
Output: a new $Q^{(t+1)}$.
1. Identify the case of insertion for $d_\alpha$.
2. Case:
3.      1: If the first element of $Q^{(t)}$ is $(u_1, v_1)^{(p_1)}$
           then
4.             Let $U$ be the retrieved $(u_1, v_1)^{(p_1)}$ by using Eqs. (2), (3) and (4).
5.         else
6.             Put $d_1$ in $U$.
7.         EndIf
8.         Use Algorithm 1 with $d_\alpha$ and $U$ to generate a new set of elements.
9.      EndCase
10.     2: If the last element of $Q^{(t)}$ is $(u_m, v_m)^{(p_m)}$
           then
11.            Let $U$ be the retrieved $(u_m, v_m)^{(p_m)}$ by using Eqs. (2), (3) and (4).
12.        else
13.            Put $d_m$ in $U$.
14.        EndIf
15.            Use Algorithm 1 with $d_\alpha$ and $U$ to generate a new set of elements.
16.     EndCase
17.            3: Let $U$ be the retrieved $(u_m, v_m)^{(p_m)}$ by using Eqs. (2), (3) and (4).
18.            Use Algorithm 1 with $d_\alpha$ and $U$ to generate a new set of elements.
19.     EndCase
20.            4: Let $U$ be the retrieved $(u_j, v_j)^{(p_j)}$ by using Eqs. (2), (3) and (4).
21.            Let $V$ be the retrieved $(u_k, v_k)^{(p_k)}$ by using Eqs. (2), (3) and (4).
22.            Use Algorithm 1 with $d_\alpha$, $U$ and $V$ to generate a new set of elements.
23.     EndCase
24.            5: Let $U$ be the retrieved $(u_k, v_k)^{(p_k)}$ by using Eqs. (2), (3) and (4).
25.            Use Algorithm 1 with $d_\alpha$, $w_j$ and $U$ to generate a new set of elements.
26.     EndCase
27.            6: Let $U$ be the retrieved $(u_j, v_j)^{(p_j)}$ by using Eqs. (2), (3) and (4).
28.            Use Algorithm 1 with $d_\alpha$, $U$ and $w_k$ to generate a new set of elements.
29.     EndCase
30.            7: Use Algorithm 1 with $d_\alpha$, $w_j$ and $w_k$ to generate a new set of elements.
31.     EndCase
32. Repeat
33.            Use Algorithm 1 with the new set of elements and the unpacked element next to the new set
               next to the new set of elements to generate the next new set of elements in $Q^{(t)}$.
34. Until no more new elements.
35. Rename $Q^{(t)}$ as $Q^{(t+1)}$.
36. EndCase
```

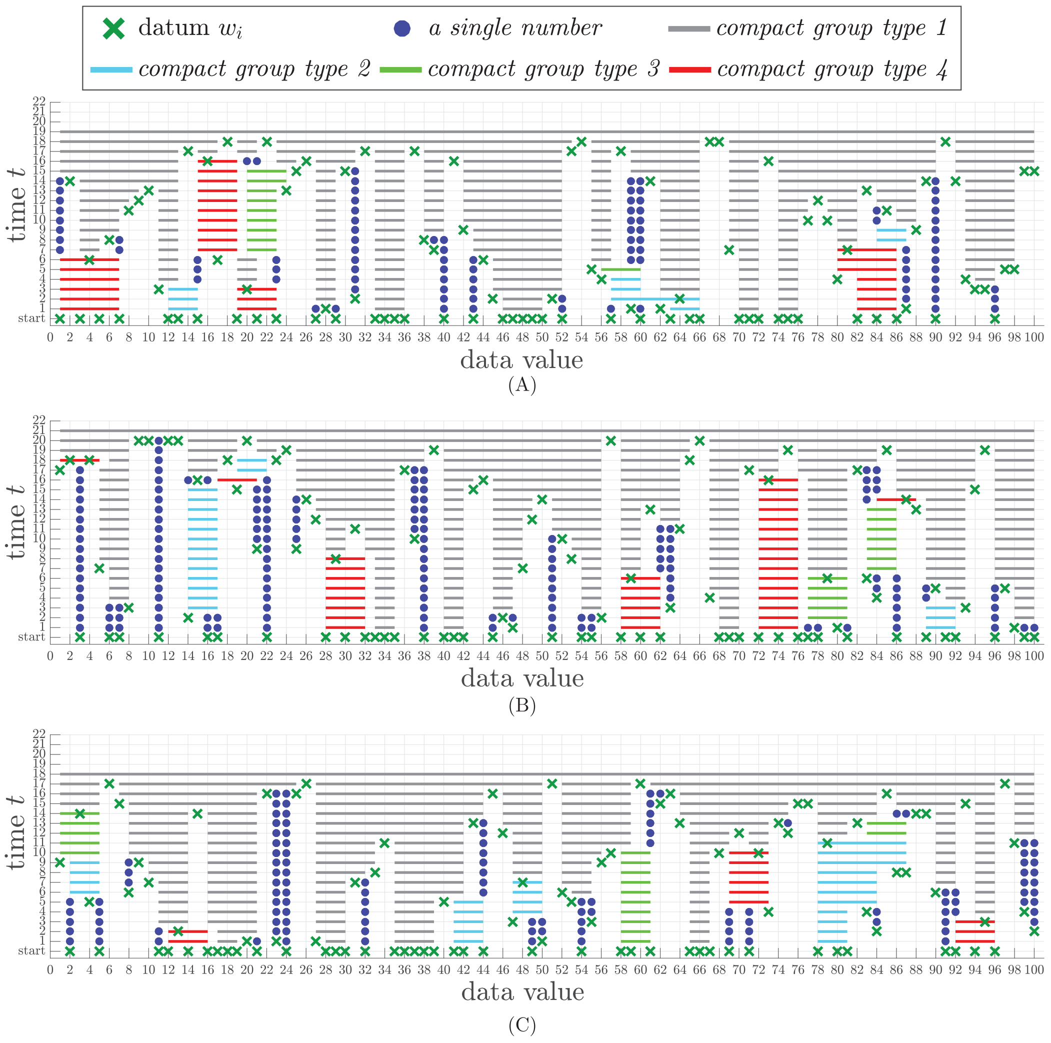

Figure 3 **Snapshots of sorting results from three different permuted data sets, each of which contains 100 numbers. (A) The time steps of the sorting result of data set 1. (B) The time steps of the sorting result of data set 2. (C) The time steps of the sorting result of data set 3.**
Full-size DOI: 10.7717/peerj-cs.355/fig-3

all incoming 100 numbers appearing in various chunks within only 19 time steps, whereas it takes 21 and 18 time steps for the second and the last data sets, respectively.

Execution time vs working storage size

In this experiment, the relation between the total numbers to be sorted and the sorting time was investigated. The total storage, m_{tot}, is partitioned into two portions. The first portion, m_{prog}, is for the sorting program. The size of m_{prog} is fixed throughout the sorting process. The second portion, m_{work}, is the working storage for storing all compact groups, sets of single numbers, and other relevant variables occurring during in the sorting algorithm. Since the sorting time directly depends upon the size of m_{work}, the size of m_{work} is thus set as a function of the total numbers to be sorted. Let $n \gg |m_{\text{work}}|$ be the total

Table 3 **Sorting execution time of the proposed algorithm with respect to size of working storage.**

n	Execution time (s)			
	γ = 50%	γ = 45%	γ = 40%	γ = 35%
10^3	0.87	0.98	1.16	1.30
10^4	7.15	8.08	9.33	11.02
10^5	71.06	79.17	92.80	109.88
10^6	709.49	791.17	933.80	1,065.71

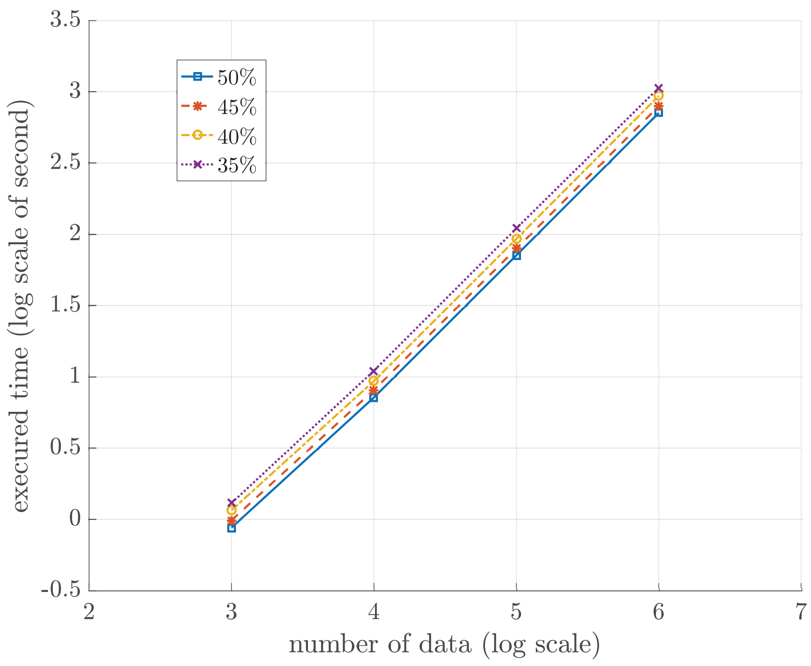

Figure 4 **Log-scaled sorting execution time in seconds for different sizes of working storage** $n = 10^3$**,** 10^4**,** 10^5 **and** 10^6**.** Full-size DOI: 10.7717/peerj-cs.355/fig-4

numbers to be sorted. All numbers to be sorted flow gradually and continuously into the working storage one chunk at an initial time. To investigate the execution time of the sorting process with respect to the quantity of numbers and $|m_{\text{work}}|$, the size of m_{work} is set in terms of n as follows.

$$|m_{\text{work}}| = \gamma \times n \tag{5}$$

where $\gamma \in \{0.50, 0.45, 0.40, 0.35\}$ and $n \in \{10^3, 10^4, 10^5, 10^6\}$.

Table 3 summarizes the proposed sorting algorithm time of different quantities of incoming numbers with respect to the different sizes of the working memory. The incoming numbers were randomly generated and permuted. No duplicated numbers appear in the data sets. To visualize the trend of the sorting time vs the size of data sets, Fig. 4 shows the log-scaled trend of each data set. There are four lines in blue, red, yellow, and purple representing different sizes of m_{work}. Note that the sorting time of each data set linearly

Table 4 Sorting execution time of external sorting with respect to buffer size.

n	Execution time (s)			
	Buffer = 50%	Buffer = 45%	Buffer = 40%	Buffer = 35%
10^3	2.30	2.69	2.64	2.66
10^4	26.95	21.04	20.19	20.18
10^5	209.41	207.27	206.87	205.15
10^6	6,187.14	5,312.09	4,754.30	4,539.04

increases. Then, the experiment has a linear polynomial time complexity of $O(n)$. Table 4 summarizes the time of *external sorting* of different quantities of incoming numbers with respect to the different sizes of buffer. The execution time of proposed sorting algorithm is approximately 8.72 times faster than the execution time of *external sorting* at a million data when storage size is limited to 50%. Furthermore, the proposed algorithm run on 4 GB of numeric data takes about 4.21 days.

Fluctuation of compact groups and single number sets

Since the proposed sorting algorithm is designed to cope with a streaming data environment where the set of numbers to be sorted can overflow the working storage and the chunks of numbers gradually flow into the working storage, there are three interesting behavioral periods concerning the number of compact groups and sets of single numbers created during the sorting process. It is remarkable that the number of compact groups and sets of single numbers increase during the beginning period due to random values of incoming numbers. The length of beginning period depends upon the random sequence of numbers, which is unpredictable. After the beginning period, some new incoming numbers may fall into the existing compact groups and some of them may form new compact groups with some sets of single numbers. Some existing compact groups can be merged with new compact groups created from some sets of single numbers into new compact groups. These conditions make the number of compact groups almost stable for some period of time. In the last period, those new incoming numbers obviously fall to combine with the existing compact groups. Some sequences of compact groups are possibly merged into new compact groups with more elements in the groups. Thus, the number of compact groups decreases until there is one compact group that contains all sorted numbers. Figure 5 illustrates the fluctuation of compact groups with sets of single numbers vs the time steps for different sizes of working storage. During the sorting process, the number of compact groups and sets of single numbers increases and decreases. The fluctuation of used and unused areas of working storage of the results in Fig. 5 is summarized in Fig. 6. Notice that the proposed algorithm can reduce the working space to 65% of the data size. In the other words, the working space of the proposed algorithm is 35% of the data size.

Comparison of storage size used and correctness of sorted order

Regardless of the sorting types, either exact sort or approximate sort, the order of each number in the sorted list must be correct according to the value of each number for

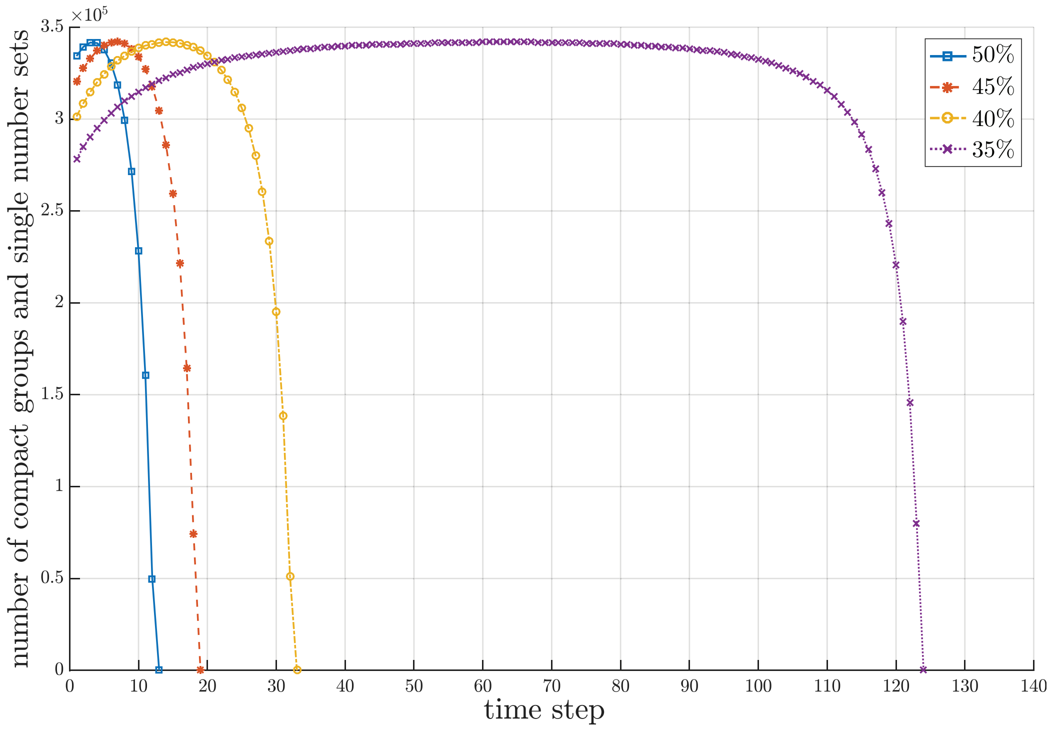

Figure 5 **Snapshot of fluctuation of compact groups and sets of single numbers at a million data during the sorting period for different sizes of working storage.**
Full-size DOI: 10.7717/peerj-cs.355/fig-5

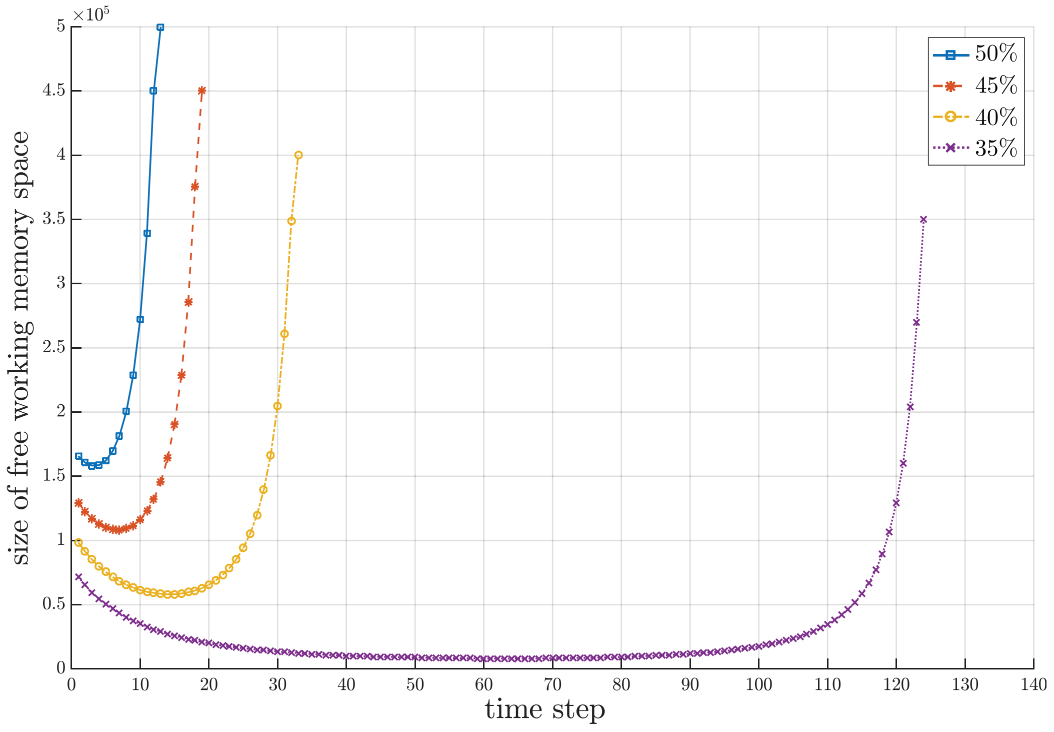

Figure 6 **Snapshot of fluctuation of unused area of working storage at a million data during the sorting period for different sizes of working storage.** Full-size DOI: 10.7717/peerj-cs.355/fig-6

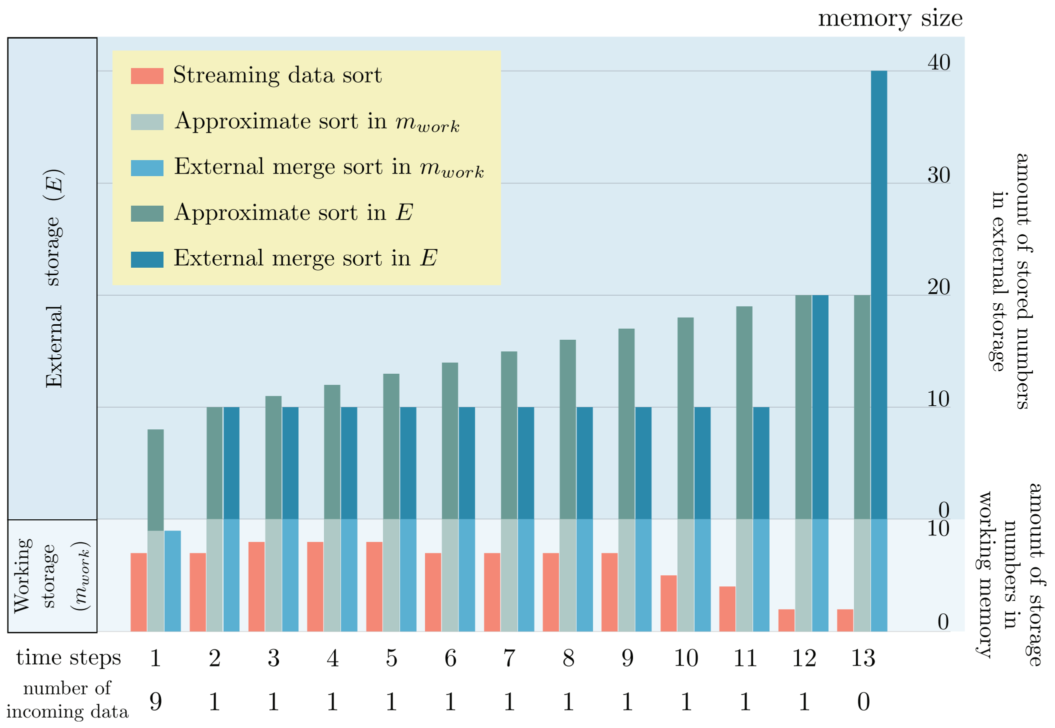

Figure 7 Comparison of storage size change as the results of sorting a set of streaming data by the proposed streaming data sort algorithm, approximate sorting (*Farnoud, Yaakobi & Bruck, 2016*) algorithm and external sorting. Full-size DOI: 10.7717/peerj-cs.355/fig-7

both ascending and descending sorts. If it is not so, the sorted list is useless in any applications. To verify the efficiency and the accurate order of sorted numbers as the result of proposed *streaming data sort* in a streaming data environment with limited storage size, the result was compared with the result of *approximate sorting* algorithm (Farnoud, Yaakobi & Bruck, 2016) capable of handling streaming data, and *external sorting*. The following set of numbers was experimented: {18, 1, 10, 6, 2, 12, 9, 3, 16, 19, 14, 11, 17, 13, 20, 8, 4, 15, 5, 7}. In order to simulate streaming data, the set of numbers was decomposed into several consecutive chunks. The first incoming chunk contains nine numbers. The other following chunks contain only one number. Two issues concerning the change of storage size during the sorting process and the wrong sorted order were recorded in the experiment. Since *streaming data sort* algorithm uses only one working storage of fixed size throughout the sorting process, there is no change of storage size for this algorithm. But in case of *approximate sorting* and *external sorting* algorithms, both of them require working storage of fixed size and also external storage of variable size. Hence, the change of storage size only occurs in the external storage. Figure 7 snapshots the storage size change at each time. m_{work} is a constant denoting the fixed size of working storage. The size of external storage is expandable according to the amount of temporary data generated during the sorting algorithms.

It is remarkable that the proposed *streaming data sort* does not require any space in the external storage. Only working storage space alone is enough to complete the sorting process. But both *approximate sorting* and *external sorting* need some additional spaces in the external storage. These spaces keep increasing when there are more incoming numbers to be sorted. The sorted order of all numbers obtained from *streaming data sort* is perfectly correct. But the sorted orders of numbers 7, 8, 13, 14 obtained from *approximate sorting* are not correct. Although the sorted order obtained from *external sorting* is perfectly correct, this algorithm requires a large size of external storage which is impractical for streaming data environment.

Time complexity analysis

There are two main phases in the sorting process. The first phase is to sort the first incoming chunk of numbers to obtain the first set of compact groups as well as sets of single numbers. The second phase is to sort the consequent chunks with the existing compact groups and single numbers. Let $h \leq |m_{\text{work}}|$ be the size of each input chunk. The time complexity of each phase is as follows.

Phase 1: The operation of this phase is in Algorithm 1. Obtaining h numbers takes $O(h)$. These h numbers must be sorted to create compact groups and sets of single numbers. The time to sort h numbers is $O(h \log(h))$. After sorting, the time to create compact groups and sets of single numbers takes $O(h)$. Thus, the time of this phase is $O(h) + O(h \log(h)) + O(h) = O(h \log(h))$.

Phase 2: From Algorithm 1, all compact groups at any time are in set C, and all single numbers are in set S. The time complexity of this phase can be analyzed from Algorithm 2. There are seven cases to be identified for inserting a new number d_α at step 1. The identifying time takes $O(|C|) + O(|S|) = \max(O(|C|), O(|S|))$. Then, applying Eqs. (2), (3) and (4) to retrieve the numbers from a compact group takes $O(1)$. After retrieval of the numbers, Algorithm 1 is applied to create a new compact group and a set of single numbers with the new incoming d_α. This step takes at most $O(h)$. At steps 32–34, Algorithm 1 is repeatedly applied to update sets C and S. This takes at most $O(h \times |C|) + O(|S|)$. Since $|C| \leq h$ and $|S| \leq h$, the time complexity of steps 32–34 is $O(h^2)$. Thus, phase 2 takes $\max(O(|C|), O(|S|)) + O(1) + O(h) + O(h^2) = O(h^2)$ for each d_α. If there are in total n numbers to be sorted, then the time complexity is $O(h \log(h)) + O((n - h) \times h^2) = O(nh^2)$. However, h is a constant. Hence, the time complexity of the sorting process is $O(n)$.

Storage usage analysis

The behavior of storage usage is in the form of a capsized bell shape, as shown in Fig. 5. The descriptive rationale behind this behavior was briefly provided in Fluctuation of Compact Groups and Single Number Sets Section. This section will theoretically analyze this behavior based on the probability of all seven cases for a compact group. Suppose there are n total streaming numbers to be sorted. All incoming n numbers are assumed to be randomly permuted and partitioned into $\frac{n}{h}$ input chunks of size h each. Let n_i be the numbers in the i^{th} input data chunk. After obtaining the first input data chunk, the probability of each case for the next new incoming number d_α for any compact group q_i is as follows.

Case 1: d_α is at the front of q_i. The probability of case 1 is calculated by the probability of picking d_α from $n - n_1$ and the probability of having d_α in the next input chunk. The probability of picking d_α from $n - n_1$ numbers is $\frac{1}{n-n_1}$. However, if d_α is the next new incoming number, then d_α must be in the next input data chunk. The probability that d_α is in the next input chunk is $\frac{1}{\frac{n}{h-1}}$. Thus, the probability of case 1 is as follows.

$$p_1 = \frac{1}{n-n_1} \cdot \frac{1}{\frac{n}{h}-1} \tag{6}$$

Case 2: d_α is at the rear of q_i. The probability of case 2 can be analyzed as that of case 1.

Case 3: d_α is in a compact group q_i, types 2, 3, and 4 are compact groups only.

If q_i is a type-2 compact group, then the probability that d_α is in q_i is $\left\lfloor \frac{|q_i|}{2} \right\rfloor \cdot \frac{1}{n-n_1}$, where $|q_i|$ represents the numbers compacted in q_i.

If q_i is a type-3 compact group, then the probability that d_α is in q_i is $\left\lfloor \frac{|q_i|}{2} \right\rfloor \cdot \frac{1}{n-n_1}$.

If q_i is a type-4 compact group, then the probability that d_α is in q_i is $\frac{(|q_i|-1)}{n-n_1}$.

$$p_3 = \begin{cases} \left\lfloor \frac{|q_i|}{2} \right\rfloor \cdot \frac{1}{n-n_1} \cdot \frac{1}{\frac{n}{h}-1} & \text{types 2 or 3} \\ \frac{|q_i|-1}{n-n_1} \cdot \frac{1}{\frac{n}{h}-1} & \text{type 4.} \end{cases} \tag{7}$$

Case 4: d_α is in between two compact groups q_i and q_{i+1}. The probability of case 4 can be analyzed as that of case 1.

Case 5: d_α is in between a single number w_j and a compact group q_i.

If q_i is a type-1 compact group, then the probability that d_α is in q_i is $\frac{1}{n-n_1}$.

If q_i is a type-2 compact group, then the probability that d_α is in q_i is $\frac{1}{n-n_1}$.

If q_i is a type-3 compact group, then the probability that d_α is in q_i is $\frac{1}{n-n_1}$.

If q_i is a type-4 compact group, then the probability that d_α is in q_i is $\frac{1}{n-n_1}$.

The probability of case 5 can be analyzed as that of case 1.

Case 6: d_α is in between a compact group q_i and a single number w_k. The probability of case 6 can be analyzed as that of case 1.

Case 7: d_α is in between two single numbers w_j and w_{j+1}. The probability of case 7 can be analyzed as that of case 1.

Note that the probability of all cases for the first input data chunk is written as follows.

$$p = \begin{cases} \left\lfloor \frac{|q_i|}{2} \right\rfloor \cdot \frac{1}{n-n_1} \cdot \frac{1}{\frac{n}{h}-1} & \text{case 3 (type 2 or type 3)} \\ \frac{|q_i|-1}{n-n_1} \cdot \frac{1}{\frac{n}{h}-1} & \text{case 3 (type 4)} \\ \frac{1}{n-n_1} \cdot \frac{1}{\frac{n}{h}-1} & \text{other cases} \end{cases} \tag{8}$$

After the first input data chunk, the probability of each case after m next input data chunks can be written in a generic form as follows.

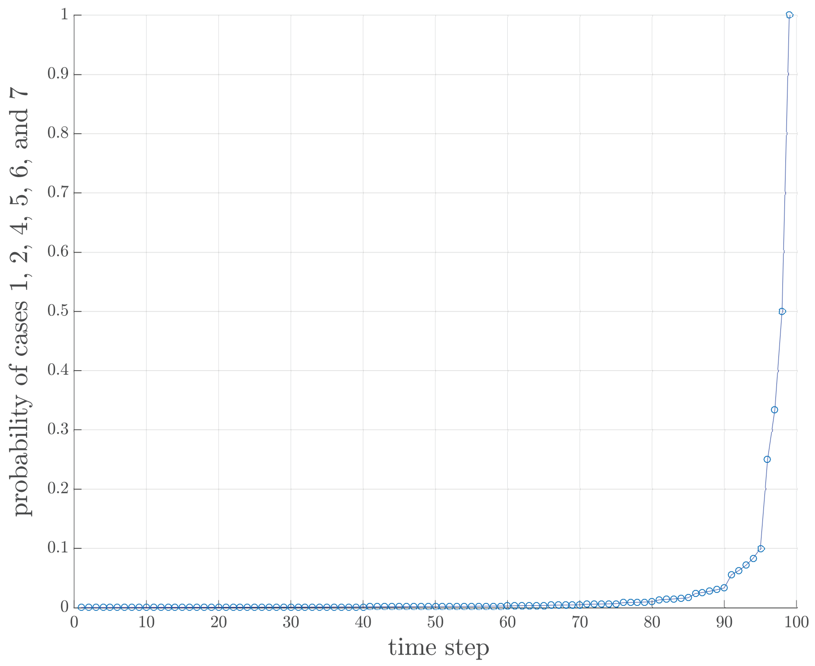

Figure 8 **Probability of cases 1, 2, 4, 5, 6 and 7 of 100 data where h = 4.**
Full-size DOI: 10.7717/peerj-cs.355/fig-8

$$p = \begin{cases} \left\lfloor \dfrac{|q_i|}{2} \right\rfloor \cdot \dfrac{1}{n - \sum_{i=1}^{m} n_i} \cdot \dfrac{1}{\frac{n}{h} - m} & \text{case 3 (type 2 or type 3)} \\ \dfrac{|q_i| - 1}{n - \sum_{i=1}^{m} n_i} \cdot \dfrac{1}{\frac{n}{h} - m} & \text{case 3 (type 4)} \\ \dfrac{1}{n - \sum_{i=1}^{m} n_i} \cdot \dfrac{1}{\frac{n}{h} - m} & \text{other cases} \end{cases} \tag{9}$$

Note that the value $n - \sum_{i=1}^{m} n_i$ and $\dfrac{n}{h} - m$ will finally approach 1. This implies that the number of compact groups decreases and eventually there should be only one compact group. However, the time during which the probability approaches 1 depends upon the value of h, as shown in Fig. 8. If h is large, then the chance that an input chunk contains a tentative sorted sequence is also high.

Theorem 2 *The only possible existing case to be tested for the last incoming data chunk is case 1, case 2, case 4, case 5, case 6, or case 7.*

Proof: The only probability approaching 1 is the probability of case 1, case 2, case 4, case 5, case 6, and case 7 as defined in Eqs. (9).■

CONCLUSION

This study proposed a concrete concept and practical algorithm to sort streaming numbers in the case where the total numbers overflow the actual storage. No secondary storage is involved in this constraint. The size of the working storage, h, for sorting is fixed throughout the sorting event. The incoming numbers are captured by new proposed data architectures in the forms of sets of single numbers and compact groups of sorted numbers. The actual order of each number with respect to the total numbers, n, in the streaming sequence can be correctly retrieved within $O(h)$. The time complexity of the proposed algorithm is $O(n)$, and the space complexity is $O(M)$. From the experiments, it was found that the proposed algorithm can correctly and stably handle the streaming data size of at least 2.857 times larger than the size of the working storage. Furthermore, the sorted order obtained from the proposed algorithm is absolutely correct, no approximate order. In addition, each number can be directly retrieved from any *compact group* by its type. The analysis of dynamic change of used and unused working storage areas during the sorting process was also provided.

Although the proposed algorithm is primarily designed for a single processor, the proposed algorithm can be practically extended to be implemented on a multiprocessor architecture with a slight modification. In the case of a multiprocessor architecture, more than one chunk of data can simultaneously flow into the machine by one chunk per processor. The proposed algorithm can be deployed by each processor to sort each incoming chunk and to merge the final sorted results from all processors later. In fact, there are several real applications requiring this kind of sorting process where the data always overflow the working memory. Some applications are the followings:

1. Managing tremendous information inside large organizations by sorting transactions according to account numbers, locations of customers, date stamp, price or popularity of stock, ZIP code or address of mail, and so on (*Sedgewick & Wayne, 2011*). The proposed algorithm can reduce memory storage for keeping those data.
2. Reducing the search time of huge streaming data by sorting the data first and representing them in compact groups as implemented in *streaming data sort* algorithm.
3. Computing order statistics, quartile, decile, and percentile of big streaming data continuously flowing into an internet-scale network monitoring system and database query optimization (*Buragohain & Suri, 2009*).
4. Checking duplicated data for fraud detection or fake social engagement activities such as bidding on an item, filling out a form, clicking an advertisement, or making a purchase (*Metwally, Agrawal & El Abbadi, 2005*; *Li et al., 2016*).

Even though the proposed *streaming data sort* successfully sorts the streaming data under the defined constraints but some of the following further studies of streaming data sorting based on other constraints can be pursued.

1. Developing a new structure of compact group whose type can be adapted to any arbitrary different value of two temporal consecutive numbers.
2. Extending the sorting concept to cope with various data types such as a character string or a floating point number which exist in other engineering, scientific, and business problems.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding
This work was supported by Development and Promotion of Science and Technology Talents Project (DPST) and Thailand Research Fund under grant number RTA6080013. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Grant Disclosures
The following grant information was disclosed by the authors:
Development and Promotion of Science and Technology Talents Project (DPST).
Thailand Research Fund: RTA6080013.

Competing Interests
The authors declare that they have no competing interests.

Author Contributions
- Suluk Chaikhan conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Suphakant Phimoltares conceived and designed the experiments, analyzed the data, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Chidchanok Lursinsap conceived and designed the experiments, analyzed the data, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.

Data Availability
The following information was supplied regarding data availability:

Data, including permutations of random integers, are available at the UCI repository (https://archive.ics.uci.edu/ml/datasets.php). Specifically:

- Beijing PM2.5 Data Data Set: https://archive.ics.uci.edu/ml/datasets/Beijing+PM2.5+Data
- Census Income Data Set: http://archive.ics.uci.edu/ml/datasets/Census+Income
- Covertype Data Set: https://archive.ics.uci.edu/ml/datasets/covertype
- Diabetes Data Set: https://archive.ics.uci.edu/ml//datasets/Diabetes
- PM2.5 Data of Five Chinese Cities Data Set: https://archive.ics.uci.edu/ml/datasets/PM2.5+Data+of+Five+Chinese+Cities
- Incident management process enriched event log Data Set: https://archive.ics.uci.edu/ml/datasets/Incident+management+process+enriched+event+log
- KEGG Metabolic Relation Network (Directed) Data Set: https://archive.ics.uci.edu/ml/datasets/KEGG+Metabolic+Relation+Network+(Directed)

- KEGG Metabolic Reaction Network (Undirected) Data Set: https://archive.ics.uci.edu/ml/datasets/KEGG+Metabolic+Reaction+Network+(Undirected)

- Buzz in social media Data Set: https://archive.ics.uci.edu/ml/datasets/Buzz+in+social+media+.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.355#supplemental-information.

REFERENCES

Agrawal A, Sriram B. 2015. Concom sorting algorithm. In: *015 4th International Conference on Computer Science and Network Technology*. Vol. 1. 229–233.

Al-Fuqaha A, Guizani M, Mohammadi M, Aledhari M, Ayyash M. 2015. Internet of things: a survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials* **17(4)**:2347–2376 DOI 10.1109/COMST.2015.2444095.

Babcock B, Babu S, Datar M, Motwani R, Widom J. 2002. Models and issues in data stream systems. In: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02*. New York, 1–16.

Bey Ahmed Khernache M, Laga A, Boukhobza J. 2018. Montres-nvm: an external sorting algorithm for hybrid memory. In: *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. Piscataway: IEEE, 49–54.

Buragohain C, Suri S. 2009. Quantiles on streams. In: *Encyclopedia of Database Systems*. New York: Springer-Verlag US, 2235–2240.

Cardenas A, Manadhata P, Rajan S. 2013. Big data analytics for security. *IEEE Security & Privacy* **11(6)**:74–76 DOI 10.1109/MSP.2013.138.

Cormen TH, Leiserson CE, Rivest RL, Stein C. 2009. *Introduction to algorithms*. Third Edition. Cambridge: The MIT Press.

Dave M, Gianey H. 2016. Different clustering algorithms for big data analytics: a review. In: *2016 International Conference System Modeling & Advancement in Research Trends*. 328–333.

Dua D, Graff C. 2019. *UCI machine learning repository: school of information and computer science*. Irvine: University of California.

Elder M, Goh YK. 2018. Permutations sorted by a finite and an infinite stack in series. In: *International Conference on Language and Automata Theory and Applications*. 220–231.

Farnoud F, Yaakobi E, Bruck J. 2016. Approximate sorting of data streams with limited storage. *Journal of Combination Optimization* **32**:1133–1164.

Faro S, Marino FP, Scafiti S. 2020. Fast-insertion-sort: a new family of efficient variants of the insertion-sort algorithm. In: *SOFSEM (Doctoral Student Research Forum)*. 37–48.

Franceschini G. 2004. Proximity mergesort: optimal in-place sorting in the cache-oblivious model. In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*. Vol. 15. New York: ACM, 291–299.

Gantz J, Reinsel D. 2012. The digital universe in 2020: big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future* **2007(2012)**:1–16.

Goel S, Kumar R. 2018. Brownian motus and clustered binary insertion sort methods: an efficient progress over traditional methods. *Future Generation Computer Systems* **86(4)**:266–280 DOI 10.1016/j.future.2018.04.038.

Huang X, Liu Z, Li J. 2019. Array sort: an adaptive sorting algorithm on multi-thread. *Journal of Engineering* **2019(5)**:3455–3459 DOI 10.1049/joe.2018.5154.

Idrizi F, Rustemi A, Dalipi F. 2017. A new modified sorting algorithm: a comparison with state of the art. In: *2017 6th Mediterranean Conference on Embedded Computing.* 1–6.

Kanza Y, Yaari H. 2016. External sorting on flash storage: reducing cell wearing and increasing efficiency by avoiding intermediate writes. *VLDB Journal* **25(4)**:495–518 DOI 10.1007/s00778-016-0426-5.

Katal A, Wazid M, Goudar R. 2013. Big data: Issues, challenges, tools and good practices. In: *2013 Sixth International Conference on Contemporary Computing.* 404–409.

Kehoe B, Patil S, Abbeel P, Goldberg K. 2015. A survey of research on cloud robotics and automation. *IEEE Transactions on Automation Science and Engineering* **12(2)**:398–409.

Keim D, Qu H, Ma K-L. 2013. Big-data visualization. *IEEE Computer Graphics and Applications* **33(4)**:20–21.

Laga A, Boukhobza J, Singhoff F, Koskas M. 2017. Montres: merge on-the-run external sorting algorithm for large data volumes on ssd based storage systems. *IEEE Transactions on Computers* **66(10)**:1689–1702.

Lee Y-S, Quero LC, Kim S-H, Kim J-S, Maeng S. 2016. Activesort: efficient external sorting using active ssds in the mapreduce framework. *Future Generation Computer Systems* **65**:76–89.

Li Y, Martinez O, Chen X, Li Y, Hopcroft JE. 2016. *In a world that counts: clustering and detecting fake social engagement at scale*. Republic and Canton of Geneva: International World Wide Web Conferences Steering Committee.

Liang Y, Chen T, Chang Y, Chen S, Wei H, Shih W. 2020. B*-sort: enabling write-once sorting for non-volatile memory. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* Piscataway: IEEE.

Mehmood A, Natgunanathan I, Xiang Y, Hua G, Guo S. 2016. Protection of big data privacy. *IEEE Access* **4**:1821–1834.

Metwally A, Agrawal D, El Abbadi A. 2005. Duplicate detection in click streams. In: *Proceedings of the 14th International Conference on World Wide Web.* 12–21.

Mohammed AS, Amrahov a E, Çelebi FV. 2017. Bidirectional conditional insertion sort algorithm; an efficient progress on the classical insertion sort. *Future Generation Computer Systems* **71**:102–112.

Osama H, Omar Y, Badr A. 2016. Mapping sorting algorithm. In: *2016 SAI Computing Conference (SAI).* 488–491.

O'Malley O. 2008. Terabyte sort on apache hadoop. *Available at http://sortbenchmark. org/Yahoo-Hadoop*.

Sedgewick R, Wayne K. 2011. *Algorithms*. Boston: Addison-Wesley Professional.

Singh H, Sarmah M. 2015. Comparing rapid sort with some existing sorting algorithms. In: Das K, Deep K, Pant M, Bansal J, Nagar A, eds. *Proceedings of Fourth International Conference on Soft Computing for Problem Solving: Advances in Intelligent Systems and Computing.* Vol. 335. New Delhi: Springer.

Tambouratzis T. 1999. A novel artificial neural network for sorting. *IEEE Transactions on Systems, Man, and Cybernetics: Part B* **29(2)**:271–275.

Thusoo A, Shao Z, Anthony S, Borthakur D, Jain N, Sarma J, Murthy R, Liu H. 2010. Data warehousing and analytics infrastructure at facebook. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010.* 1013–1020.

Vatrapu R, Mukkamala RR, Hussain A, Flesch B. 2016. Social set analysis: a set theoretical approach to big data analytics. *IEEE Access* **4**:2542–2571.

Vignesh R, Pradhan T. 2016. Merge sort enhanced in place sorting algorithm. In: *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT).* 698–704.

Witayangkurn A, Horanont T, Shibasaki R. 2012. Performance comparisons of spatial data processing techniques for a large scale mobile phone dataset.

YiLiang S, Zhenghong Y. 2016. Communication rules learning strategy in big data network based on SVN neural network. In: *2016 International Conference on Intelligent Transportation, Big Data & Smart City.* 309–313.

Zhai C, Zhang Y, Hu P. 2018. Modeling and analysis of material supply network based on big data packed with traffic. In: *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis.* Piscataway: IEEE, 490–494.

Zhao C, Chang H, Liu Q. 2017. Bayesian algorithm based traffic prediction of big data services in openflow controlled optical networks. In: *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA).* Piscataway: IEEE, 823–826.