



PowerHash: a hybrid grouping scheme by leveraging power-law properties of data

Xun Wei¹ · Xiaowang Kong¹ · Yanfeng Zhang¹ · Ge Yu¹

Received: 20 January 2019 / Accepted: 12 June 2019 / Published online: 25 June 2019
© Springer Nature Switzerland AG 2019

Abstract

We study GroupBy implementation scheme which is widely used in distributed systems and databases. The GroupBy operation partitions a set of out-of-order records into groups. Due to the massive data size, many I/O-efficient grouping schemes that exploit external memory have been proposed. In this paper, we observe that the group sizes of many real data exhibit power-law property and the grouping schemes' performance varies a lot for data with different group sizes. The *indexing-filling* approach prefers data with big group size, while the *partitioned hash* approach prefers data with small group size. Based on this observation, we propose a hybrid approach, *PowerHash*, which invokes different grouping schemes for different data. The group size information is approximately estimated by the count-min sketch so that the big groups and small groups can be distinguished from each other. With a given memory budget, our results show that PowerHash can improve performance by up to six times over the existing GroupBy implementations.

Keywords Key-value grouping · Power-law distribution · GroupBy · Hash table

1 Introduction

Key-value grouping operation, known as GroupBy clause, is a key operation in database systems [11, 15, 16, 20]. It groups a set of out-of-order records/rows into multiple groups according to a certain key/column. GroupBy operation is often used with aggregate functions which produces a single row of summary information for each group, e.g., GroupBy-Aggregate in SQL. GroupBy is also widely used in distributed computing frameworks for distributing and locating the data. MapReduce [8] expresses the computational process as three phases: map, shuffle and reduce. During the shuffle phase, the key-value pairs that have the same key are aggregated together to form a map output file, which are sent to the same reduce processor where the final cluster-wide aggregation (reduction) is applied on. The efficiency of the key-value grouping step is crucial to both relational databases and distributed computing systems.

There are two categories of GroupBy implementations in general, *sort-based grouping* and *hash-based grouping*. Basically, the sort-based grouping makes the data records

sorted in the order of the group key, so that the records with the same group key are located together, while the hash-based grouping typically hashes key-value pairs to an in-memory hash table structure where multiple key-value pairs in the same group (sharing the same group key) are stored in the same bucket.

Recall that key-value pairs (kv-pairs) grouping operation is the key operation in Hadoop MapReduce [8]. The map output kv-pairs are locally grouped by keys before they are shuffled to reduce workers. These map output kv-pairs (i.e., reduce input kv-pairs) from various map workers are further merged to obtain a global grouped kv-pairs. Each group of kv-pairs sharing the same key is the input of a reduce function. In Hadoop MapReduce implementation, a kind of sort-based grouping, *merge-sort grouping*, is used, which can perform quite general grouping tasks at scale even in the absence of available memory. However, merge-sort grouping involves large amount of redundant computation and I/Os. The previous works [13, 19, 22] have shown that the merge-sort grouping adopted by Hadoop was found to be among the worst-performing choices.

For many applications, hash-based grouping is adopted because these applications require only unsorted grouping [14, 22]. However, hash-based grouping consumes more memory than sort-based grouping because it requires to load

✉ Yanfeng Zhang
zhangyf@mail.neu.edu.cn

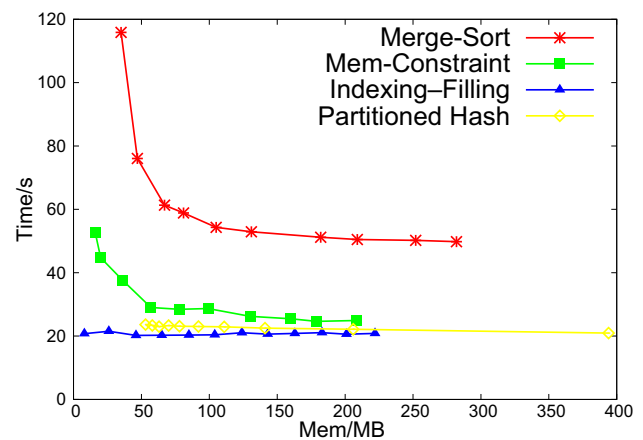
¹ Northeastern University, Shenyang, China

all records into memory. The performance heavily depends on the amount of available memory. A variant of hash grouping has been used in MariaDB [3], Oracle [20], PostgreSQL [15] and SQL Server [2]. It creates an in-memory hash table for grouping rows. If the hash table becomes too large to be fit in memory, the input records are partitioned into smaller work tables which are recursively partitioned until they fit into memory. Once all input groups have been processed, the completed in-memory groups are output and repeat the algorithm by reading back and aggregating one spilled partition at a time until all partitions have been processed. We refer to this approach as *memory-constraint hash grouping*. It excels at efficiently aggregating large datasets and performs better than merge-sort grouping in some situations.

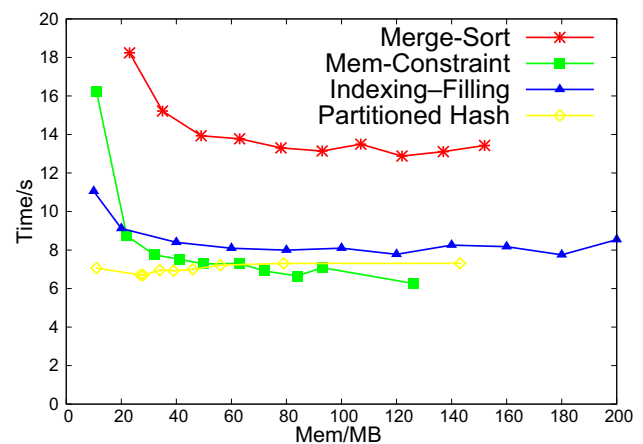
An alternate of hash grouping that can avoid memory overflow is using rehashing. The first hash grouping is applied to obtain coarse-grain kv-pair groups, where each group of kv-pairs covering multiple unique keys is written to a disk file. The second phase loads each kv-pairs file into memory and performs hash grouping on keys, so that the kv-pairs sharing the same key are grouped together in a bucket. These grouped kv-pairs are then written out for recycling memory to group next coarse-grain grouped kv-pairs (in the next file). The number of files (coarse-grain groups) can be tuned according to available memory, and further optimizations can be applied to avoid memory overflow when processing each coarse-grain group. We refer to this approach as *partitioned hash grouping*.

In addition, we can use an *indexing-filling* approach that takes advantages from both sort grouping and hash grouping. A nice property of sort-based grouping is that the start position of each group in the output file is determined before the group operation starts (given the fact that the group size is fixed). However, in hash-based grouping, the query of a specific group can be very fast. It first reads all data in a streamlined fashion and evaluates the group size information associated for each key, which is maintained in an in-memory table (offset table). The output file offsets of all groups can be then calculated. In other words, the output kv-pairs are “indexed” in the first indexing phase. Then, the data are parsed for the second time, and the kv-pairs are written out to specific file positions according to the offset table, which is the filling phase. Note that the write operations can be cached to be sequential for I/O efficiency.

After reviewing these grouping algorithms, one question is raised. Which one is the best? Our observation is that no one wins all the time. As known, the group sizes in real datasets follow power-law distributions, especially for the graph datasets where the group size depends on node degree. The big group sizes and small group sizes vary over an enormous range. We further observe that these implementations exhibit different performance and memory efficiency for big groups and small groups. We extract the kv-pairs



(a) Performance comparison on big groups



(b) Performance comparison on small groups

Fig. 1 Performance comparison of big groups and small groups

of big groups and small groups from a simulation dataset Pareto (see Sect. 4 for dataset description) and evaluate the above four algorithms on grouping time and memory usage. It is obvious that using more memory will obtain the grouping result faster, so we show the results with both memory cost and runtime. When spending the same memory cost, the method with shorter runtime shows better performance.

The result for grouping big groups is shown in Fig. 1a, where the indexing-filling method performs the best over others. For big groups, it is likely to result in extremely large partitions that will be spilled to disk due to limited memory. This will incur multiple times of data exchanges between memory and disk, no matter using merge-sort grouping, memory-constraint hash or partitioned hash. In this case, the indexing-filling approach is the most I/O-efficient one. On the other hand, for grouping small groups as shown in Fig. 1b, the hash-based grouping approaches (memory-constraint hash and partitioned hash) perform better due to its fast location ability. In the case of processing extremely

large input (i.e., with relatively small memory budget), the partitioned hash grouping runs faster with limited memory.

Based on this observation, we propose a hybrid grouping scheme *PowerHash* that leverages power-law property of the data group sizes. The big groups and small groups are processed separately. The kv-pairs of big groups are grouped by indexing–filling method, and the kv-pairs of small groups are grouped by the partitioned hash grouping approach. However, there is a key problem raised. How to estimate the group sizes efficiently and how to distinguish between the big groups and small groups? We use the count-min sketch [6] to approximately estimate the group sizes and identify the big groups and small groups. Given a memory budget, we comprehensively use indexing–filling grouping and partitioned hash grouping to group the data with the three-phase algorithm *PowerHash*. Our experimental results show that *PowerHash* always outperforms the other counterparts on real datasets. With the same memory cost, *PowerHash* is 2–6 times faster than the merge–sort grouping and 1.25–5 times faster than the memory-constraint hash grouping.

The rest of this paper is organized as follows. Section 2 describes the related work. In Sect. 3, we propose our grouping method *PowerHash* and present the key optimizations. The experimental results are presented in Sect. 4. Finally, Sect. 5 concludes the paper.

2 Related work

This section provides the backgrounds on existing grouping techniques and reviews the count-min sketch that will be used in our algorithm.

2.1 Commonly used grouping implementations

2.1.1 Merge–sort grouping

One of the most widely used grouping algorithm is merge–sort grouping, which is widely used in MapReduce [8] and SQL GroupBy operator [2,4,15–17]. The Hadoop MapReduce uses a merge–sort at both map and reduce phases to group kv-pairs. These kv-pairs to be grouped will be constantly written to the memory buffer. The buffer is used to collect kv-pairs in batches, so as to minimize the impact of disk I/O. The entire buffer size is limited. As the amount of data in the buffer reaches a threshold, a background spilled thread sorts these kv-pairs in memory buffer in accordance with the key and writes these kv-pairs that have been sorted in the buffer to disk as a partial sorted file. Each spilling operation generates a spilled file on disk. Finally, a merge phase is required to sort these partial sorted files to generate the final sorted file (i.e., the data in the file are grouped). Merge–sort can achieve the purpose of aggregating kv-pairs for any

amount of data, which is highly scalable. However, keeping different groups in order is unnecessary because our goal is only making the kv-pairs with the same key continuously stored on disk. Therefore, merge–sort grouping will result in unnecessary computational overhead.

2.1.2 Memory-constraint hash grouping

Many SQL databases (e.g., MariaDB) use a hash grouping aggregation strategy for GroupBy operations [3]. We refer to this approach as *memory-constraint hash grouping*. The memory-constraint hash is similar to hash join. It does not require sort but more memory. Memory-constraint hash partitions the large task into smaller subtasks where the subtasks can be executed in memory completely [5,9,21]. If the size of data to be grouped exceeds the memory limit, one or more buckets will be spilled to disk. Then, the records that are hashed to the spilled buckets will be divided up into the corresponding partitions on disk. Once all input groups have been processed, the completed in-memory groups will be output, and the algorithm is repeated by reading back and aggregating one spilled partition at a time. However, when processing datasets whose group sizes follow power-law distributions, the sub-tables sizes are not balanced, which leads to multiple recursive partitions and downgrades grouping performance.

2.2 Group size distribution

Power-law distributions [18] occur in an extraordinarily diverse range of phenomena, e.g., the frequency of word usages in any human language [10], the number of hits on Web pages [1]. As shown in Fig. 2, the group sizes in four datasets (see Sect. 4 for dataset description) also follow the power-law distributions and vary over an enormous dynamic range. Various techniques [12] can be used to detect this skewness. For some extremely large groups whose sizes are larger than the available memory, when the groups are processed by some traditional grouping approaches, the endless recursive reading/writing between memory and disk may occur, and these approaches do not work. The distributions are all right-skewed, which means that a bulk of groups occurs for fairly small sizes and only a small number of groups are with large sizes.

2.3 The count-min sketch

The count-min sketch (CM sketch) [6,7] is a probabilistic data structure that serves as a frequency table of events in a data stream and uses hash functions to map events to frequencies at the expense of overcounting some events due to collisions.

A CM sketch is represented by a two-dimensional array counting with width w and depth d . Additionally, d hash

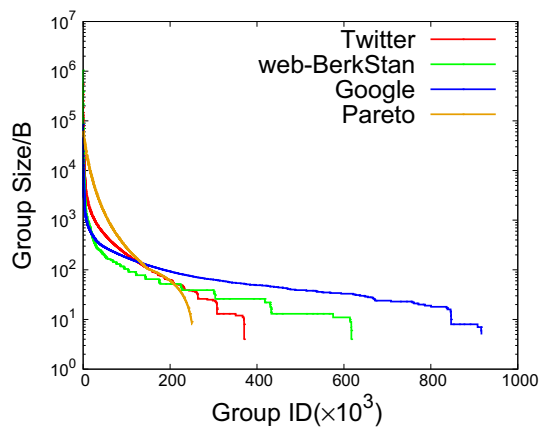


Fig. 2 Group size distribution for different datasets

functions $h_1 \dots h_d$ are chosen uniformly at random from a pairwise-independent family. The CM sketch is simple to construct and can be used to count the frequencies of the distinct items in data stream quickly. The count-min sketch is named after the two basic operations, counting first and computing the minimum next.

Frequency Counting When it gets an item a_i with a quantity of c_i , then c_i is added to one count in each row; the counter is determined by the d hash function. Formally, set $\forall 1 \leq j \leq d : \text{count}[j, h_j(a_i)] \leftarrow \text{count}[j, h_j(a_i)] + c_i$.

Frequency Computing The frequency of a unique item a_i is the minimum of statistical results in each row, and for $\forall 1 \leq j \leq d$, it is given by $f_i = \min_j \text{count}[j, h_j(a_i)]$.

Our strategy is to leverage the CM sketch to count the group sizes roughly, and the rough group sizes are used to distinguish between the big groups and small groups. Then, the big groups and small groups are processed separately. The big groups are grouped by the indexing–filling method to reduce the I/O cost, and the small groups are processed by partitioned hash grouping approach. PowerHash can finish the grouping operation in limited memory with high efficiency.

3 PowerHash design and implementation

When processing large dataset, the memory-constraint GroupBy implementations need to spill part of data to disk. These spilled partitions will be read back and will be merged as the final result. For hash-based grouping approaches, it is possible that the reads and writes of a specific data partition are executed recursively when this partition contains one or more big groups. This will degrade the performance seriously. As discussed in Sect. 1, indexing–filling could be a better implementation for big groups. We take the group size information into account and propose our PowerHash grouping scheme.

Our algorithm contains three phases: *groups distinction*, *big groups grouping* and *small groups grouping*. The whole process is shown in Fig. 3. The groups distinction phase is to distinguish between big groups and small groups. First, it estimates the approximate group sizes by the CM sketch. With a user-specified ratio, the groups indexed by keys are classified into big groups and small groups. The grouping phase for big groups leverages the indexing–filling method. It generates an offset index that records the output positions of big groups firstly and then fills the result file by file random access according to the offset index. Since the number of big groups is small, the index size is small. The grouping phase for small groups leverages the partitioned hash grouping. After grouping big groups, this division of small groups can avoid unbalance to a great extent because the small group sizes vary over a small range. If a certain partition exceeds the available memory size, it will be split again until any of its sub-partitions can be processed in memory. In the following, we will describe the three phases in detail.

3.1 Phase 1: groups distinction

The groups distinction phase is to distinguish between the big groups and small groups. The distinction relies on count-min (CM) sketch. As shown in Fig. 3, an in-memory two-dimensional array is maintained to count the approximate size of each group. Based on the approximate group sizes, we can distinguish between big groups and small groups depending on a user-specified ratio and then employ different methods to deal with the big groups and small groups.

First, we need to obtain the group size of each group. The group size is the number of bytes a group occupies. The CM sketch will count the frequency of each distinct item with small space at the expense of some minor errors. In our algorithm, the counting process is the accumulation of the size of each key–value pair (in bytes) as introduced in Sect. 2. When processing a $\langle \text{key}, \text{value} \rangle$ pair, the group size counted by the CM sketch will be updated according to the kv-pair size and then the size is accumulated in one bucket on each row. Though there might be inaccuracies for group sizes owing to the collisions of hashing, the CM sketch guarantees that the estimated group size is bigger than the real group size. In other words, it is likely to obtain a big estimated group size for real big group, and the real group size is definitely small if the estimated group size is small.

Then, we need to identify the big groups and small groups on the basis of the CM sketch where the rough group sizes are stored. Users can specify the top ratio r groups as big groups and the rest as small groups. If we sort the whole group sizes, the cost of sorting is great. Each row of the CM sketch is the count result after passing the whole dataset, and we found that it can reflect the distribution of group sizes roughly, so we sort one row of the CM sketch instead of

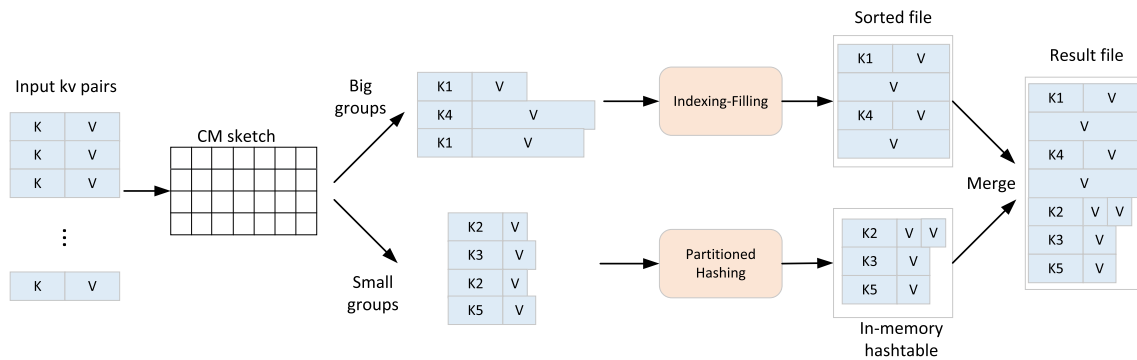


Fig. 3 Overall structure of PowerHash

sorting all group sizes. Suppose the width of CM sketch is w , if we sort a row of CM sketch in descending order, the threshold for separating big groups is determined as the value of the $(w * r)$ th position in the sorted list. The groups whose rough group sizes are larger than the threshold are considered as big groups.

The whole distinction process can be completed efficiently by using CM sketch, and it only takes up about 15% of the total time according to experimental results.

3.2 Phase 2: handling big groups

For the datasets that follow power-law distribution, big groups take up majority of space. If the big groups are processed by hash-based grouping methods, the in-memory hash table could be too large to degrade the performance. Once the hash table becomes too large to be fit in memory, the kv-pairs could be written out and read in many times. The repeat access to disk files can seriously degrade the performance. Grouping big groups by indexing-filling approach will avoid this problem. The property that big groups are only with a small number of distinct keys makes the offset index smaller. Filling output files based on the offset index will only lead to large amount of sequential writes but less number of random seeks.

Firstly, we need to construct an offset index that records the output positions for big groups. The offset is the calculated position of each group in the output file. With a specific output order, each big group's write-out position (i.e., offset) can be calculated given the group size of each big group. Thus, we need to count each big group's accurate size. When a group is judged as a big group, its kv-pairs' size (in bytes) will be accumulated.

Each group size implies the number of bytes the big group will take up in the final output file. If the big group sizes are stored in a group size table in $\langle \text{key}, \text{groupsize} \rangle$ format, we can obtain the group offsets by querying the group size table. The traversing order implicitly determines the order of these big groups in the result file. With this output order,

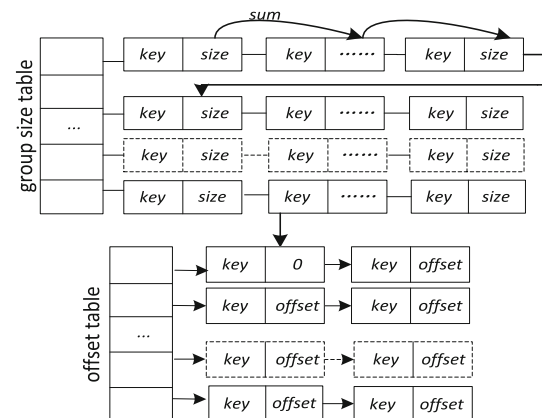


Fig. 4 Generation of offset index

each offset is the accumulated group sizes of all previous groups that have been accessed. Thus, the group size table with $\langle \text{key}, \text{groupsize} \rangle$ pairs can be transformed to the offset table with $\langle \text{key}, \text{offset} \rangle$ pairs. The whole process is shown in Fig. 4.

After constructing the offset index, the kv-pairs in big groups are written to certain positions in the result file. In the whole process, only the offset index is always kept in memory, so the memory usage for grouping big groups is very small. The indexing-filling process can be executed with limited memory. Compared to the hash-based grouping methods, it is not necessary to store each kv-pair in hash table. The kv-pairs are written to certain positions in the result file directly. There is no extra access to disk caused by memory limitation, so it runs faster in the case of limited available memory as shown in Fig. 1a. The profiling of this process shows that most of the time is spent on seek operations during the file filling. This is mainly due to the fact that the write of kv-pairs is not necessarily sequential. Filling the output files with big groups will lead to relatively large amount of sequential writes but less number of seeks.

3.3 Phase 3: handling small groups

For a dataset whose group sizes follow a power-law distribution, the majority of data have been written out to the result file after dealing with the big groups. The number of remainder small groups is greater than the number of big groups. If we still employ indexing–filling method to group the kv-pairs for small groups, the size of offset index can be huge and may cause out-of-memory problem in the case of limited memory. On the other hand, the small group sizes vary over an small range, which is likely to result in balanced partition if using in-memory hash partitioning. Next, we need to determine the partition number based on the available memory and the number of small groups.

Let T denote the total size of dataset, and let A denote the currently available memory. $sizeof(entry)$ is the entry size of hash table in hash grouping. In addition, the total size S of small groups can be obtained after the first phase and the number of big groups b can be obtained after the second phase. Combined with the ratio r of big groups, the small group number s is $b * (1 - r)/r$. The space occupied when small groups are being processed is $S + sizeof(entry) * s$. Because the actual size of hash table for small groups may be larger than the estimated size, we set an expansion factor α to expand the hash table appropriately. The partition number p of small groups can be obtained according to Eq. (1).

$$p = \frac{S + \alpha * sizeof(entry) * s}{A} \quad (1)$$

For each kv-pair in a small groups, we calculate the hash value H_{key} first and then hash the kv-pair to a partition depending on the partition id according to Eq. (2).

$$id = H_{key} \% p \quad (2)$$

Depending on the partition mode, the kv-pairs are hashed to different partitions. Then, the kv-pairs in small groups are grouped partition by partition by hash grouping. If the hash table is too large to be fit in memory when dealing with a large partition, the big partition will be divided into multiple parts and then be merged one by one. The repartition is unlikely to happen thanks to balanced partitioning, so the partitioned hash grouping is very fast.

3.4 Algorithm summary

We summarize the whole process including the three phases in Algorithm 1. The algorithm starts by creating a two-dimensional CM sketch array with width w and depth d . Then, each $\langle key, value \rangle$ read from the input file F is hashed to the CM sketch to calculate the rough group size. In order to avoid traversing the whole input file in the second phase,

Algorithm 1 PowerHash

Require: File F , ratio r , available memory A , width w and depth d of CM sketch

Ensure: result files R

```

1:  $C :=$  a two-dimensional array with width  $w$  and depth  $d$ 
2:  $H :=$  {for each  $i$  do generate a hash function  $h_i$ ,  $0 \leq i < d$ }
3:  $key\_file :=$  a file of recording the key and value size
4: initialize  $C = \{C[i][j] = 0, 0 \leq i < d, 0 \leq j < w\}$ 
5: for each input  $\langle key, value \rangle$  in  $F$  do
6:   write  $\langle key, valuesize \rangle$  to the  $key\_file$ 
7:   for each  $h_i \in H$ ,  $0 \leq i < d$  do
8:      $C[i][h_i(key)] += sizeof(key) + sizeof(value)$ 
9:   end for
10: end for
11:  $T :=$  a group size table for the big groups
12: calculate threshold  $t$  between big groups and small groups.
13: for each  $\langle key, valuesize \rangle$  in  $key\_file$  do
14:   group size  $f_{key} = \min(C[i][h_i(key)]), 0 \leq i < d$ 
15:   if  $f_{key} > t$  then
16:     insert the  $\langle key, valuesize \rangle$  into  $T$ 
17:   end if
18: end for
19: calculate partition number  $p$  depending on Formula 1
20: offset index  $O := Convert(T)$ 
21:  $P :=$  {for each  $i$  do generate a partition file  $s_i$ ,  $0 \leq i < p$ }
22: for each input  $\langle key, value \rangle$  in  $F$  do
23:   if there is a match in  $O$  then
24:     write the  $\langle key, value \rangle$  into  $R$ 
25:     update the corresponding offset in  $O$ 
26:   else
27:      $id = H(key) \% p$ 
28:     insert the  $\langle key, value \rangle$  into  $s_{id}$ 
29:   end if
30: end for
31: for each  $s_{id}$ ,  $0 \leq i < p$  do
32:   Grouping( $s_{id}$ ) by hashing and append to  $R$ 
33: end for
34: remove  $key\_file$  and partition set  $P$ 
35: output  $R$ 

```

we use an intermediate file key_file to record each key and its corresponding value size in $\langle key, valuesize \rangle$ format, i.e., we transform the original format $\langle key, value \rangle$ of the input into $\langle key, valuesize \rangle$ format, by which we reduce the I/O cost and the computation cost for accurate group sizes in the second phase.

The threshold t to separate big groups from small groups is calculated according to CM sketch and ratio r (Line 12). We first sort one row of the two-dimensional array C in descending order, and the threshold t is the $(w * r)$ th element of the sorted row. Then, the groups can be distinguished by their rough sizes. While traversing the key_file , the $\langle key, valuesize \rangle$ belonging to big groups are inserted to the group size table to calculate the accurate group size based on Eq. (3). The other $\langle key, valuesize \rangle$ are accumulated to obtain the total size of small groups. We can obtain the group size table after traversing the key_file . The partition number p can be also calculated according to Eq. (1):

$$groupsize+ = sizeof(key) + valuesize \quad (3)$$

The *Convert* function converts the group size table to offset index (Line 22). Precisely, it traverses the elements in a top-down manner and then accumulates the group sizes to obtain the output offset depending on the traversing order. The first entry of the table T (i.e., the first $\langle key, groupsize \rangle$) is the first group, so the initial offset of the entry equals 0. It means that the first kv-pair mapped to the group will be stored at the beginning of the result file. Let $group_{pre}$ be the group size of the previous entry and off_{pre} be the initial offset of the previous group. So the offset of current entry $off_{cur} = off_{pre} + group_{pre}$. After calculating the offset of each group, we will obtain the in-memory offset table which contains $\langle key, offset \rangle$ information.

Filling the result file needs another pass of the input file (Line 22–25). For each kv-pair, we retrieve the offset off_{key} by querying the offset index O . If there is a match, it belongs to a big group and will be written to position off_{key}^{th} of the result file R . At meantime, the offset for this key should be updated by increasing the value size, which is the update operation in Line 25. If there is no match, it will be written to the corresponding partition in P (Line 26–28).

After the big groups are processed successfully, the small groups of each partition stored on disk are still unordered. These small group partitions are then read into memory partition by partition and processed with hash grouping. Finally, these small group partitions with ordered kv-pairs are appended to the result file R (Line 31–33).

3.5 Analysis

We analyze the time complexity and I/O cost of our approach as follows.

Time Complexity Let n be the number of records in the dataset, w and d be the width and depth of CM sketch respectively, and r be the ratio of big groups. In the group distinction phase, it takes $O(dn)$ time to compute d hash values for building CM sketch and takes $O(\log w)$ time to sort one row of CM sketch to obtain the threshold for separating big groups. It also needs $O(n)$ time to separate groups. In the second phase for grouping big groups, PowerHash takes $O(n \cdot r)$ time to convert group size table to offset index table and takes $O(n \cdot r)$ time again to output these big groups by querying offset table. In the third phase for grouping small groups, PowerHash takes $O(2n \cdot (1 - r))$ time to realize rehashing by hashing the small groups twice. Therefore, PowerHash takes $O((d + 3) \cdot n)$ time in total for obtaining the grouped result.

I/O Cost The I/O cost is critical to big data systems, especially in a limited memory environment. Our approach needs to first read the whole file to construct CM sketch in order to obtain a big-group–small-group separation threshold. Then,

Table 1 Datasets

Dataset	Size (MB)	Illustration
Twitter	180.4	Social network data
Web-BerkStan	102.5	Web graph data
Google	97.2	Web graph data
Pareto	575.4	Simulation data

it requires another pass of the file to separate big groups from small groups. For big groups, we need to write them out after obtaining the group size statistics, and another read is required to build offset index. For small groups, two reads and two writes of all small groups are required for rehashing. So it totally requires three reads and two writes of the whole data. Comparing with recursively read/write by memory-constraint hash method, our approach is more I/O efficient.

4 Experimental evaluation

This section presents the performance evaluation for PowerHash. We compare our work against existing typical grouping approaches, merge-sort [8] and memory-constraint hash [3]. For merge-sort, we use the implementation from the official sites. There is no source code for memory-constraint hash available, so we implemented our own hash aggregation version following the pseudocode described in [9]. We used typical real datasets: (a) the Higgs¹ Twitter dataset, providing the information about activity on Twitter during the discovery of Higgs boson; (b) web-BerkStan² dataset, a web graph containing 685,230 nodes and 7,600,595 edges; (c) Google³ data set, a web graph dataset containing 875,713 nodes and 5,105,039 edges; and a simulation dataset that obeys a power-law distribution Pareto. Table 1 summarizes the datasets used.

All serial methods were implemented in C++ and compiled with g++ version 4.8.4 in Linux. The experiments were executed on a machine with two quad-core Intel CPUs at 2.67 GHz and 32 GB RAM. In order to have a memory-constraint environment, we limit the memory usage in our experiments.

4.1 Overall performance evaluation

The following experiments exhibit the overall performance of PowerHash and its performance comparison against merge-sort and memory-constraint hash. For PowerHash, the ratio

¹ <http://snap.stanford.edu/data/higgs-twitter.html>.

² <http://snap.stanford.edu/data/web-BerkStan.html>.

³ <http://snap.stanford.edu/data/web-Google.html>.

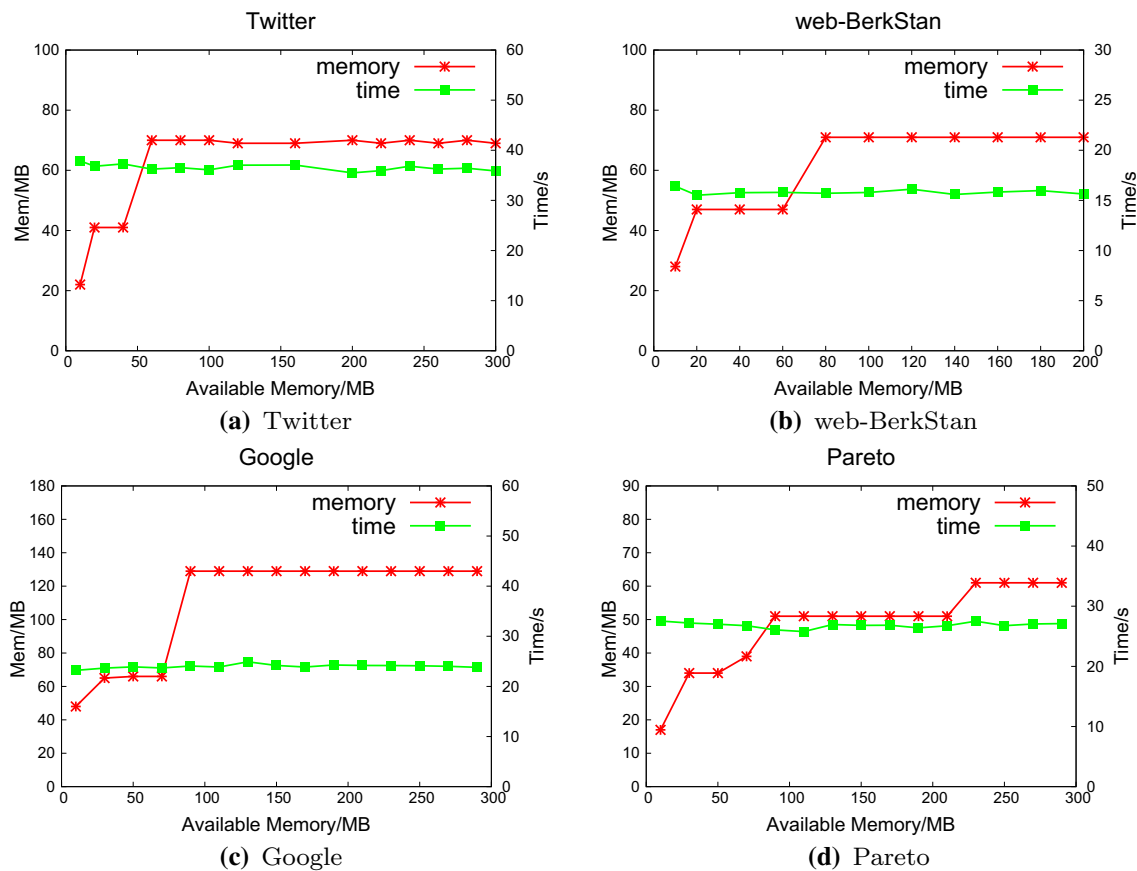


Fig. 5 Overall performance on various datasets

of big groups is set as 0.2 and the expansion factor discussed in Sect. 3 is set as 2.

Figure 5 shows the overall performance of PowerHash on different datasets, and it shows the real memory consumption and grouping time by varying the available memory. As the available memory varies, the grouping time is almost stable; even if the available memory is very small, the time cost in which case is nearly equal to the time cost with sufficient memory in Fig. 5. The reason is that the available memory only influences the partition of small groups. The grouping time of big groups keeps steady no matter how much memory is available. In addition, the grouping time for small groups is nearly the same when group sizes are evenly distributed, so the time cost of PowerHash keeps stable.

From Fig. 5, we can also see that the memory consumption grows steadily and then stays stable. The memory consumption can be considered as the peak of memory usage in each experiment. According to the experimental results, the memory usages are always the space occupied by the hash table in small groups hash grouping process. So each jump of memory consumption implies a decrease in small groups partition number. The memory usage stays stable finally because there is enough memory to load the kv-pairs in small groups and

the partition number decreases to 1. The minimal memory consumptions shown in Fig. 5 reveal the offset index size in the indexing–filling phase, e.g., it is only 22 MB as shown in Fig. 5a. In other words, PowerHash can achieve its best performance with just about 22 MB on Twitter. We can see that PowerHash can complete the key grouping operation with a little memory.

The next experiment investigates the scalability of our method. Figure 6 shows the comparison of grouping time against merge–sort and memory–constraint hash with different memory budgets. We show the experimental results on different datasets. The x-axis depicts the memory usage, and the y-axis depicts the spent time. With increased memory, the time used should be reduced accordingly. So the curves shown in the figure exhibit better performance when they are closer to the left-bottom corner.

For merge–sort and memory–constraint hash, the execution time continues to decline with the available memory increasing constantly. When the memory is large enough so that we can process kv-pairs in memory completely, the performance is no longer increasing and the grouping time stays stable, at which point the memory–constraint hash has reduced to the pure hash grouping method. The merge–sort

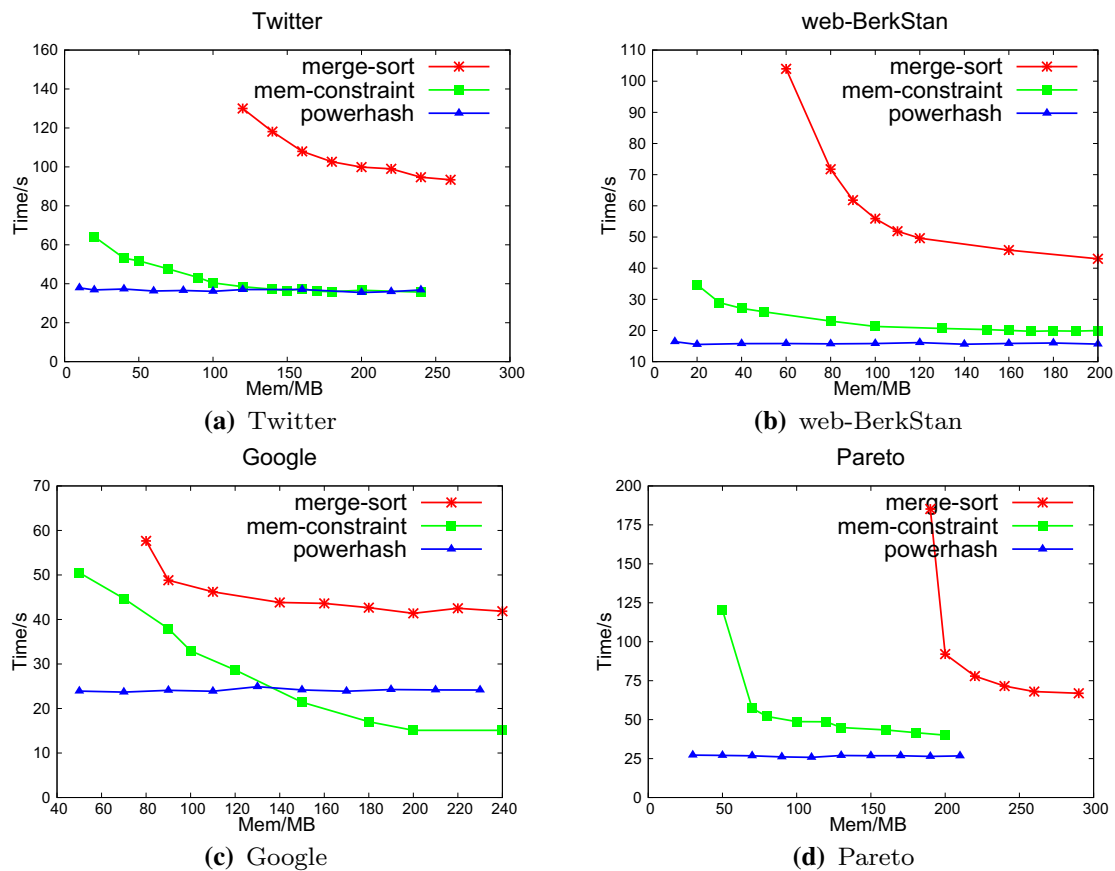


Fig. 6 Performance comparison on various datasets

needs more than ten times memory and memory-constraint hash needs five times (i.e., 120MB) memory for Twitter to achieve the best performance compared to our algorithm as shown in Fig. 6a. The experimental results on other datasets are similar.

Compared to merge-sort and memory-constraint hash, the difference is greater when memory is limited. The runtime of memory-constraint hash is at least two times slower than PowerHash with the same memory budget on the four datasets. The runtime difference between merge-sort and PowerHash is even greater. The difference continues to decrease with more available memory. Even if the memory-constraint hash reaches the stable state, our algorithm runs at least 25% faster on web-BerkStan and at least 60% faster on Pareto as shown in Fig. 6b, d. Memory-constraint hash can achieve the same performance on Twitter and Google, while its time consumption is still greater than PowerHash with the same memory. It also outperforms PowerHash on Google with enough memory.

We make the following explanation for this phenomenon. Recall that in Sect. 2, for merge-sort algorithm, there is unnecessary computational overhead for aggregating kv-pairs. The sort process reduces its performance. The time

cost is always significant even if the memory is enough. For memory-constraint hash, one or more buckets will be selected to be spilled to disk when the memory usage has reached a threshold. These spilled partitions will be read back for remerging subsequently. The power-law distribution makes it more difficult to divide the data into small uniform partitions. For a larger bucket spilled to disk, subsequent reading back and remerging may lead to recursive execution of the algorithm. The I/O overhead increases sharply. However, PowerHash deals with the big groups and small groups separately, where the big groups are processed by indexing-filling, and the small groups are processed by partitioned hash grouping approach. The two grouping methods can be carried out with limited memory to avoid repeat accesses to disk. Therefore, the performance of memory-constraint hash is slower than PowerHash with the same memory budget.

If there is enough memory, all kv-pairs can be loaded in memory, memory-constraint hash is downgrade to into the pure hashing grouping method. Its performance can be faster than PowerHash as shown in Fig. 6c. Therefore, our hybrid approach is more efficient with limited memory compared with other algorithms.

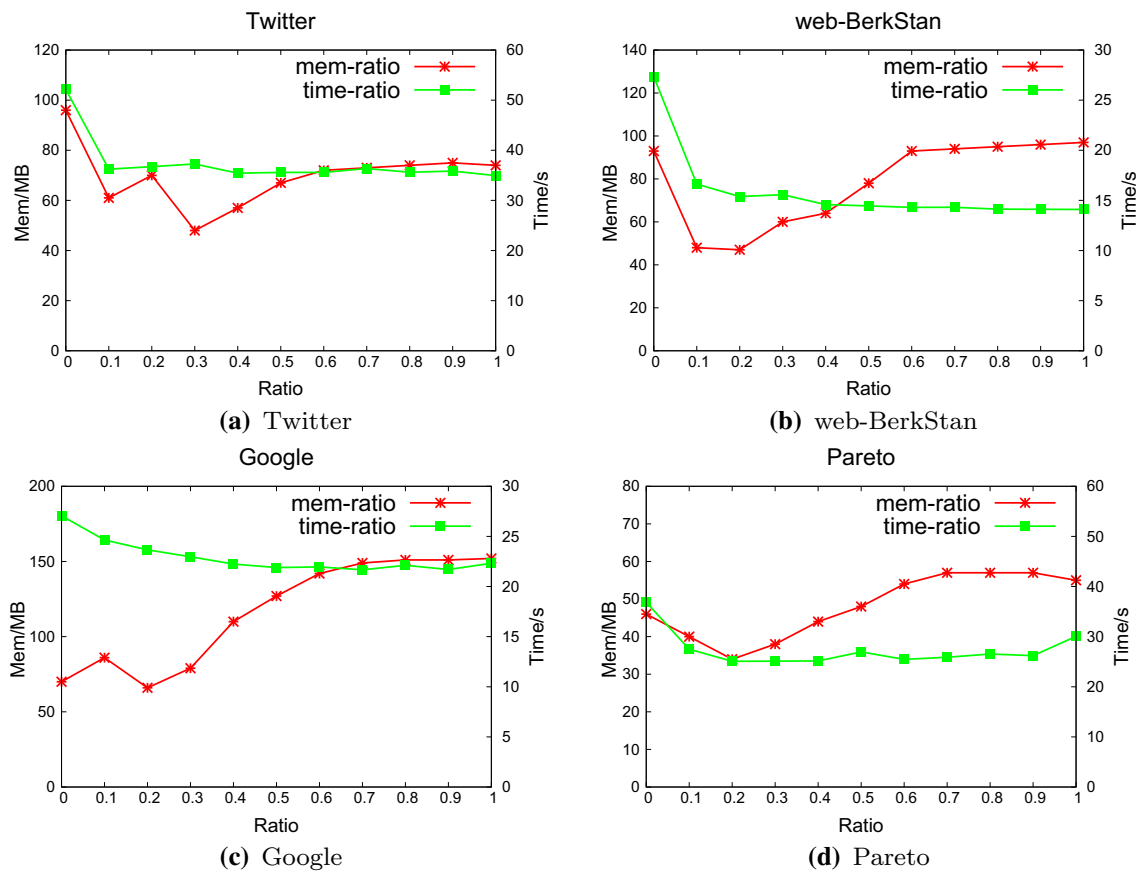


Fig. 7 Performance when varying big group's ratio

4.2 Parameters study

We also evaluate the effect of parameter settings.

Ratio of big groups The ratio r of big groups is a key parameter in PowerHash. A smaller r may lead to skewed partitions of small groups, while a bigger r may result in too big offset index size; as a result, memory is wasted. As shown in Fig. 7, we show the performance when r is varying from 0 to 1. The available memory is fixed and set as 50 MB. PowerHash is exactly the partitioned hash method when the ratio is 0, and turns to the indexing-filling method when the ratio is 1. The memory usages in both cases are much larger than 50 MB, so that out-of-memory errors might occur. With the ratio increasing, the execution time decreases and then stays stable. The memory usage continues to decline and then continues to increase. When least memory usage is required, it indicates that the offset index size and small group partition sizes are the most suitable. In Fig. 7b–d, the memory usage reaches the lowest point when the ratio is 0.2. In Fig. 7a, our algorithm performs the best when the ratio is 0.3.

The parameters of CM sketch The next experiment shows the effect of CM sketch parameters. Recall that in Sect. 3 we have mentioned that the CM sketch is employed to estimate

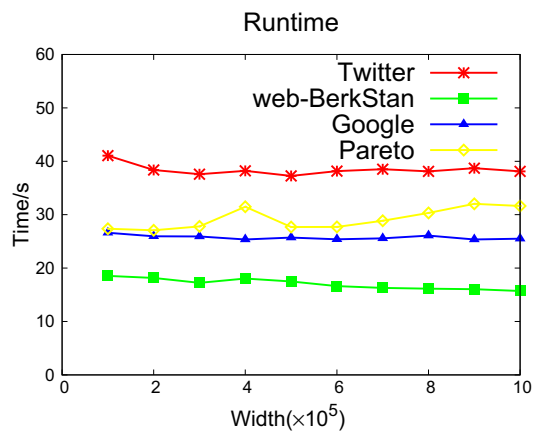
the distribution of group sizes. We sort a row of the CM sketch and then obtain the threshold between big group sizes and small group sizes according to the big groups ratio. We then divide the big groups and small groups based on their approximate sizes. If the error of group sizes is within a factor of ε with probability δ , the depth is $\lceil \ln(1/\delta) \rceil$ and the width is $\lceil e/\varepsilon \rceil$. The depth can be set easily. The width of CM sketch indicates the accuracy of threshold. Figure 8 shows the memory usage and time cost on different data sets when varying the width of CM sketch.

The runtime on each dataset is almost steady as shown in Fig. 8a, and the memory usage is influenced by the width of CM sketch as shown in Fig. 8b. It is appropriate to set the width from 200,000 to 800,000 which takes up little memory usage.

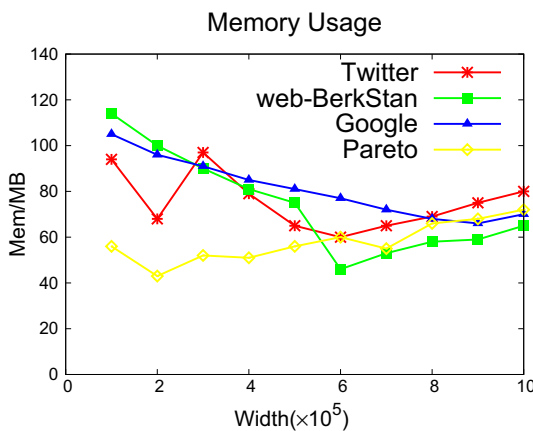
4.3 Phase evaluation

In this subsection, we evaluate the runtime of each phase. Figure 9 shows the time percentage and memory consumption on the three phases.

Figure 9a shows the time percentage of the three phases on different datasets. It is noticeable that the time percentage



(a) Runtime

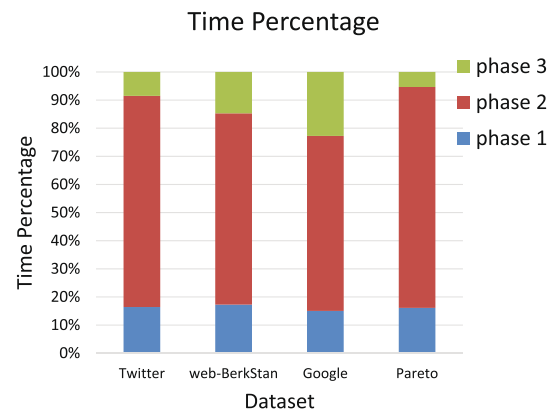


(b) Memory usage

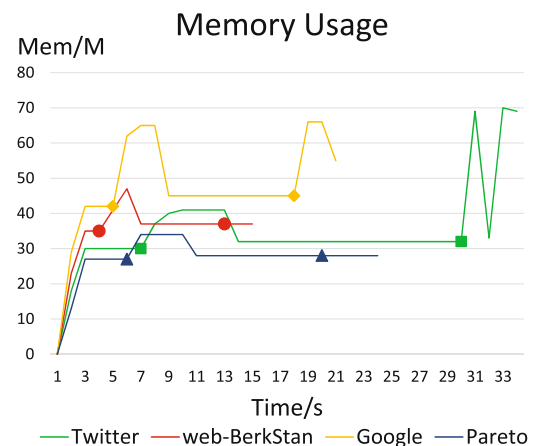
Fig. 8 Runtime and memory usage by varying the width of CM sketch

of group distinction phase is close to 15%; i.e., the preparation time of key grouping operation occupies 15% of the total time. The grouping phase for big groups takes up most of the runtime, and the grouping phase for small groups takes up much less time cost, which is under expectation. The grouping phase for small groups processes data in memory so it is very fast. The majority of dataset is processed in the second phase. Large portion of time is spent on seek operations.

We also compare the memory usage for the three phases on various datasets as depicted in Fig. 9b. The point on each line separates three phases. The memory usage continues to increase and keeps stable in the first phase. Then, the memory usage increases over time gradually in the second phase. Its growth resulted from the generation of offset index and decreased after the memory occupied by the CM sketch is released. In the third phase, the memory usage is also steady because the partitions of small groups are balanced and the size of each hash table generated is similar.



(a) Time percentage



(b) Memory usage

Fig. 9 Runtime and memory usage comparison for each phase

5 Conclusions

Grouping key–value pairs is essential to many practical systems, including Hadoop MapReduce, Databases, etc. In this paper, we propose PowerHash, a method that works efficiently even if memory is limited and is easily implemented. Extensive experimental evaluation with real datasets has shown that our method is much more efficient than existing ones in terms of speed and computational resources. When the available memory is limited, our algorithm runs faster than merge–sort grouping and memory–constraint hash grouping. The merge–sort hash and memory–constraint hash need several times memory space to achieve the best performance compared to our algorithm. This grouping scheme can be integrated into big data systems (e.g., Hadoop and Spark) for processing massive data.

Acknowledgements This work was partially supported by National Key R&D Program of China (2018YFB1003404), National Natural

Science Foundation of China (61672141) and Fundamental Research Funds for the Central Universities (N181605017).

References

- Adamic, L.: The nature of markets in the world wide web. *Quarterly J. Electron. Commer.* **1**(1) (2000)
- Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V., Syamala, M.: Database tuning advisor for microsoft SQL server 2005: demo. In: *ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pp. 930–932. ACM (2005)
- Bartholomew, D.: Mariadb vs. MYSQL. *Dostopano* **7**(10), 2014 (2012)
- Boicea, A., Radulescu, F., Agapin, L.I.: MongoDB vs oracle-database comparison. In: *EIDWT 2012*, pp. 330–335 (2012)
- Bratbergsengen, K.: Hashing methods and relational algebra operations. *VLDB* **1984**, 323–333 (1984)
- Cormode, G.: Count-min sketch. *Encycl. Algorithms* **29**(1), 64–69 (2009)
- Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. In: Farach-Colton, M. (ed.) *LATIN 2004: Theoretical Informatics*, pp. 29–38. Springer, Berlin (2004)
- Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113 (2008)
- Freedman, C.: Hash aggregate (2006). <https://blogs.msdn.microsoft.com/craigfr/2006/09/20/hash-aggregate/>. Accessed 2018
- George, K., George, K.: *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley Press, Boston (1949)
- Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.* **1**(1), 29–53 (1997)
- Khattree, R., Bahuguna, M.: An alternative data analytic approach to measure the univariate and multivariate skewness. *Int. J. Data Sci. Anal.* **1**, 1–16 (2018)
- Li, B., Mazur, E., Diao, Y., McGregor, A., Shenoy, P.: A platform for scalable one-pass analytics using mapreduce. In: *ACM SIGMOD International Conference on Management of Data (SIGMOD 2011)*, pp. 985–996 (2011)
- Lin, L., Lychagina, V., Liu, W., Kwon, Y., Mittal, S., Wong, M.: Tenzing a SQL implementation on the mapreduce framework. *PVLDB* **2011**, 1318–1327 (2011)
- Momjian, B.: *PostgreSQL: Introduction and Concepts*, vol. 192. Addison-Wesley, New York (2001)
- MySQL, A.: *Mysql 5.1 reference manual*, 2006 (2009). <http://dev.mysql.com/doc>. Accessed 2018
- Nasir, M.A.U., Morales, G.D.F., García-Soriano, D., Kourtellis, N., Serafini, M.: The power of both choices: Practical load balancing for distributed stream processing engines. In: *IEEE 31st International Conference on Data Engineering (ICDE 2015)*, pp. 137–148. IEEE (2015)
- Newman, M.: Power laws, pareto distributions and zipf's law. *Contemp. Phys.* **46**(5), 323–351 (2005)
- Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: *IEEE Symposium on Mass storage systems and technologies (MSST 2010)*, pp. 1–10. IEEE (2010)
- Stephens, S.M., Chen, J.Y., Davidson, M.G., Thomas, S., Trute, B.M.: Oracle database 10g: a platform for blast search and regular expression pattern matching in life sciences. *Nucleic Acids Res.* **33**(1), D675–D679 (2005)
- Teffer, D., Srinivasan, R., Ghosh, J.: Adahash: hashing-based scalable, adaptive hierarchical clustering of streaming data on mapreduce frameworks. *Int. J. Data Sci. Anal.* **1–11**, (2018). <https://doi.org/10.1007/s41060-018-0145-7>
- Yu, Y., Gunda, P.K., Isard, M.: Distributed aggregation for data-parallel computing: interfaces and implementations. In: *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009)*, pp. 247–260. ACM (2009)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.