PowerHash: A Hash Grouping Scheme by Leveraging Power-Law Properties of Data

Xun Wei
Northeastern University
Liaoning, 110819, China
weixun@stumail.neu.edu.com

Xiaowang Kong
Beijing Baidu Netcom Science and Technology Co.,Ltd.
kongxiaowang@baidu.com

Yanfeng Zhang
Northeastern University
Liaoning, 110819, China
zhangyf@mail.neu.edu.cn

We study the GroupBy implementation scheme 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 input 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 the performance by 1.25-2 times over the existing GroupBy implementations.

***Index Terms*—key value grouping, power-law distribution, GroupBy, hashtable.**

I. INTRODUCTION

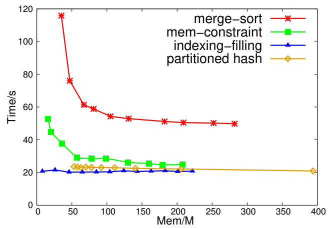
Key value grouping operation, known as GROUP BY clause, is a key operation in database systems [1], [2], [3], [4]. It groups a set of out-of-order records into multiple groups according to a certain key. GroupBy operation is often used with aggregate functions which produces a single row of summary information 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 [5] 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, and they are sent to the same Reduce processor where the final cluster-wide aggregation (reduction) is applied on them. 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 a hash table structure where multiple key-value pairs in the same group (sharing the same group key) are stored in the same bucket.

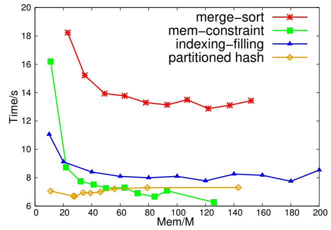
Recall that key-value pairs (kv-pairs) grouping operation is the key operation in Hadoop MapReduce [5]. The map output kv-pairs are locally grouped by keys before they are shuffled to reduce workers. These map output kv-pairs (i.e., reduce input kv-pairs) from various map workers are further merged to 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 work [6], [7], [8] shows that merge-sort grouping adopted by Hadoop is found to be among the worst-performing choices.

For many applications, hash-based grouping is adopted because these applications require only unsorted grouping [9], [7]. However, hash-based grouping consumes more memory than sort-based grouping because it requires to load all records into memory. The performance heavily depends on the amount of available memory. A variant of hash grouping has been used in MariaDB [10], Oracle [2], Postgresql [4], and SQL Server [11]. 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 data sets 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 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



(a) Performance comparison on big groups



(b) Performance comparison on small groups

Fig. 1: Performance comparison of big groups and small groups.

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 output file position of each group is determined before the grouping starts. While in hash-based grouping, the query of a specific group can be very fast. It first reads all data in one pass and evaluates the group size information associated for each key, which is maintained in an in-memory 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 data sets follow power-law distributions, 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 of big groups and small groups from a simulation data set Pareto (see Sec. IV for dataset description) and evaluate the above four algorithms on grouping time and memory usage. It is obvious that using more memory will group data faster, so we show the results with both memory cost and runtime. With the same memory budget, the shorter runtime the better.

The result for grouping big groups is shown in Figure 1a, where the indexing-filling method performs the best over the 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 for merge-sort grouping, memory-constraint hash, or partitioned hash. In this case, the indexing-filling approach is the most I/O effective one. On the other hand, for grouping small groups as shown in Figure 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 an I/O efficient hash 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 [12] 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 1.25-2 times faster than the merge-sort grouping and 2-3 times faster than the memory-constraint hash grouping.

The rest of this paper is organized as follows. Section II describes the related work. In Section III, we propose our grouping method power-law hash and present the key optimizations. Section IV discusses the parameters setting. The experimental results are presented in Section V. Finally, Section VI concludes the paper.

II. Related Work

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

A. Commonly Used Grouping Implementations

Merge-Sort Grouping. One of the most widely used grouping algorithm is merge-sort grouping, which is widely used in MapReduce [5] and SQL Group By operator [14],[3],[15],[4],[11]. The Hadoop MapReduce uses a merge-sort at both the 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 the threshold, a background spilled thread sorts these kv-pairs in the 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 the aim is making the kv-pairs that are in the same key continuously stored on disk. Therefore, merge-sort grouping will result in unnecessary computational overhead.

Memory-Constraint Hash Grouping. Some SQL databases (e.g., MariaDB) that use a hash grouping aggregation strategy for GroupBy operations [10]. We refer to this approach as *memory-constraint hash grouping*. The memory-constraint

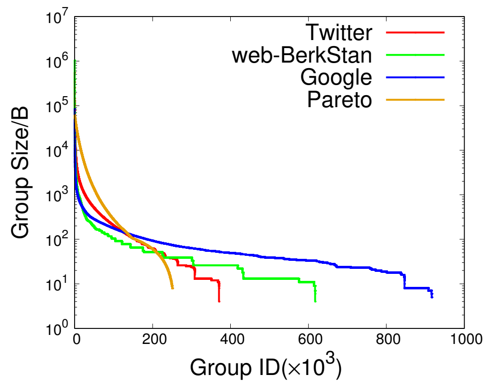


Fig. 2: The group size distributions for different data sets.

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 [16],[17]. 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 in disk. Once all input groups have been processed, the completed in-memory groups will be output and repeat the algorithm by reading back and aggregating one spilled partition at a time. However, when processing data sets whose group sizes follow power-law distributions, the sub-tables sizes are not balanced, which leads to multiple recursive partitions and downgrade grouping performance.

B. *Group Size Distribution*

Power-law distributions [18] occur in an extraordinarily diverse range of phenomena, e.g., the frequency of use of words in any human language [20], the number of hits on web pages [21]. As shown in Figure 2, the group sizes in four data sets(see Sec. IV for dataset description) are also follow the power-law distributions and vary over an enormous dynamic range. 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 from and writing to disk may occur, and then these approaches can not work.

The distributions are all right-skewed, it means that a bulk of groups occurs for fairly small sizes and only a small number of groups are much higher, in more detail, about 80% data are from top 20% big groups in Twitter, web-BerkStan and Google, the top 20% big groups occupy about 90% size in Pareto. This phenomena roughly conforms to the 80/20 rule(also known as Pareto principle)[26] which states that roughly 80% of the effects come from 20% of the causes for many events. Mathematically, the 80/20 rule is roughly followed by a power-law distribution for a particular set of parameters.

C. *The Count-Min Sketch*

The count-min sketch (CM sketch)[12],[28] is a probabilistic data structure that serves as a frequency table of events in a data stream, it 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 functions $h_1 ... h_d$ are chosen uniformly at random from a pairwise-independent family. The space used by Count-Min sketches is the two-dimensional array and d hash functions. The CM sketch is simple to construct and counting the frequencies of the unique 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 $\forall 1 \leq j \leq d : count[j, h_j(a_i)] \leftarrow count[j, h_j(a_i)] + c_i$.

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

Our strategy is to leverage the CM sketch to counting the group sizes in the data set roughly, the rough group sizes are used to distinguish between the big groups and small groups depending on Pareto principle. 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, the small groups are processed by partitioned hash grouping approach. PowerHash can finish the key grouping operation in limited memory with high efficiency.

III. PowerHash Design and Implementation

When processing large dataset, the memory constraint group-by 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 read and write for a data partition can be executed recursively when the memory is limited. This will degrade the performance seriously. As discussed in Sec. I, 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 Figure 3. The groups distinction phase is to distinguish between the 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 divided into big groups and small groups. The big groups grouping phase aims to group the kv-pairs from big groups using the indexing-filling method. It generates an offset index that records the output positions of big groups firstly, then fills the result file by file random access according to the offset index. Since the big groups are in a minority, the index size is small. The small groups grouping phase is to group the kv-pairs from small groups using 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 partition is still not fit in memory, it will be split again until its sub-partition can be processed in memory. In the following, we will describe the three phases in detail.

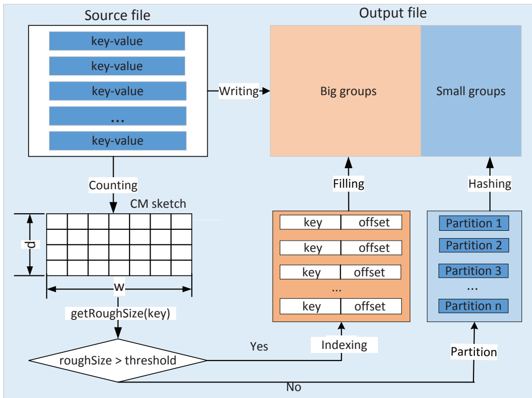


Fig. 3: Overview Intuition of PowerHash.

A. Phase 1: Groups Distinction

The groups distinction phase is to distinguish between the big groups and small groups, the whole process is based on the CM sketch. As shown in Figure 3, an in-memory two-dimensional array is maintained to count the rough size for each group, it is the working process of CM sketch. After counting the group sizes, we can distinguish between the big groups and small groups depending on a user-specified ratio, and then employ targeted methods to deal with the big groups and small groups, so the first phase is the basis of the whole algorithm.

First, we need to obtain the group size of each group, the group size is how many bytes a group occupies. The CM sketch can count the frequency of each distinct item in a data set quickly with small space if minor errors can be allowed. In our algorithm, the counting process is the accumulation of the kv-pairs sizes(in bytes) as introduced in section II. When getting a $\langle key, value \rangle$ pair, the group size counted by the CM sketch will be updated by the kv-pair size, then the size is added to one count in each row. Though the group sizes counted by CM sketch is inaccurate owing to the collisions of hashing, the CM sketch can ensure that the groups which are real big groups must have great statistical results and the groups with counting results must be real small groups.

Then we need to judge which groups are big groups on the basis of the CM sketch where rough group sizes are stored. Because the data set's group sizes follow a power-law distribution, if we can know the ratio r of big groups in the data set, the big groups and small groups can be divided easily. Referring to the Pareto principle (80/20 rule), the number of big groups only occupies around 20% of the total groups but their total size takes up about 80% of the total size, the ratio can be set around 0.2, the groups whose group sizes are the top 20% in the data set are big groups, the rest are small groups. If we sort the whole group sizes, the big groups can be obtained, but the cost of sorting the whole group sizes is great, we need to traverse the input data set again and then sort the rough group sizes. Each row of the CM sketch is the counting result after passing the whole data set, we found that it can reflect the distribution of group sizes roughly, so we determine to sort one row of the CM sketch instead of sort all of group sizes. The width of CM sketch is w, if we sort a row of CM sketch in descending order, the threshold between big group sizes and small group sizes is the $(w * r)^{th}$ value. The groups whose rough group sizes are larger than the threshold are big groups.

The whole distinguishing process can be completed efficiently by the employment of CM sketch, its time cost occupies about 15% of the total time according to experimental results.

B. Phase 2: Big groups grouping

In the data sets whose group sizes follow the power-law distributions, the total size of big groups takes up majority of the data sets size according to the Pareto principle. If the big groups are processed by the hash-based grouping methods in limited memory, the hash table will be great. Once the hash table becomes too large to be fit in memory, the kv-pairs would be written to and then read from disk frequently, the repeat access to disk reduces the performance of key grouping. Grouping big groups by indexing-filling can reduce the I/O cost above. The big groups are grouped separately in our algorithm, the property that the big groups are only a small minority in the data sets makes the offset index smaller, and filling the output files with big groups will lead to relatively large amount of sequential writes but less number of seeks.

Firstly, we need to structure an offset index that records the output positions of big groups. The offset here means the number of bytes from the written position of the group to the beginning of the file. With a specific output order, each big group's write-out position (i.e., offset) can be calculated, it is the accumulation of group sizes of all previous groups that has been accessed, so the first thing we need to do is to to count each big group's accurate size. When a group is judged as a big group, the kv-pairs in the big group will be accumulated according to the unique key of the group like Formula 1, the accurate group size is the accumulation of each kv-pair size(in byte).

$$groupsize += sizeof(key) + sizeof(value) \tag{1}$$

Each group size represents how many bytes the big group will take up in the final output file. With a specific output order, each offset is the the accumulation of group sizes of all previous groups that has been accessed. If the big group sizes are stored in a group size table in $\langle key, groupsize \rangle$ format, we can get the group offsets while traversing the group size table, in which case the traversing order is the output order, i.e., the order of these big groups in the result file is the same with the traversing order. Based on the method, the $\langle key, groupsize \rangle$s are transformed to $\langle key, offset \rangle$s saved in the offset index as shown in Figure 4.

After constructing the offset index, the kv-pairs in big groups are written to the certain position in the result file depending on the corresponding offset. In the whole process, only the offset index is always kept in memory, so the memory

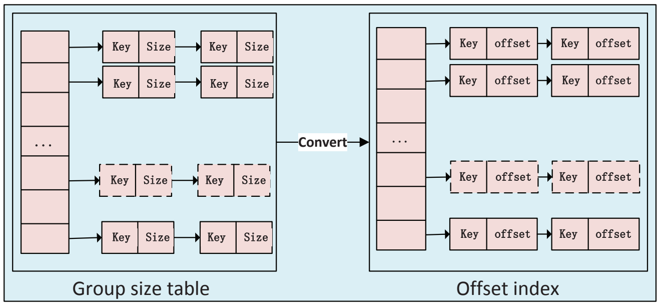


Fig. 4: The offset index generation.

usage of big groups grouping is very small, so it can be executed with limited memory. Compared to the hash-based grouping methods, each kv-pair does not need to store in hash table but is written to the certain position in the result file directly. Besides the necessary reading from and writing to the disk, there is no extra access to disk caused by the out-of-memory, so it can perform faster in the case of limited available memory as shown in Figure 1a. The profiling of this process shows that most of time is spent on seek operations during the file filling, this is mainly due to the fact that the writing of kv-pairs is not necessarily sequential, but filling the output files with big groups will lead to relatively large amount of sequential writes but less number of seeks.

C. Phase 3: Small groups grouping

For a data set whose group sizes follow a power-law distribution, the majority of data set has been written to the result file after dealing with the big groups, the number of the remainder small groups are greater than big groups, if we still employ indexing-filling method to group the kv-pairs in small groups, the size of offset index will be huge, it 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 can lead to balanced division to a great extent if we partition the small groups based on the available memory. In order to ensure the correctness of key grouping, the partition method we adopt is hashing. So we need to get the partition number that is determined by the available memory and the small groups number.

For better analysis, we first define the following identifiers: the total size T of data set, the current available memory A, each entry size $sizeof(entry)$ of the hash table in hash grouping. In addition, the total size S of small groups can be got after the first phase , the big group number 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 hash table in small groups key grouping may be larger than the calculation, so we set an expansion factor α to expand the hash table size appropriately. So the partition number P of small groups can be got like Formula 2.

$$p = \frac{S + \alpha * sizeof(entry) * s}{A} \tag{2}$$

For a kv-pair in a small groups, calculate the hash value H_{key} first because the key may has various forms, and then hash the kv-pair to a partition depending on the partition id calculated by Formula 3.

$$id = H_{key} \% p \tag{3}$$

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 partition, the big partition will be divided into two parts and then be re-aggregated one-by-one. The repartition can be reduced a lot due to the balanced division, so the partitioned hash grouping maintains the high performance of hashing grouping.

Algorithm 1 PowerHash

Input: File F, ratio r, available memory A, width w, depth d
Output: result files R

C:= a two-dimensional array with width w and depth d
H:={**for** each i **do** generate a hash function h_i, $0 \le i < d$}
key_file := a file of recording the key and value size
initialize C = $\{C[i][j] = 0, 0 \le i < d, 0 \le j < w\}$
for each input $\langle key, value \rangle$ in F **do**
  write $\langle key, valuesize \rangle$ to the *key_file*
  for each $h_i \in H$, $0 \le i < d$ **do**
    $C[i][h_i(key)]$ += $sizeof(key) + sizeof(value)$
  end for
end for
T:= a group size table for the big groups
calculate threshold t between big groups and small groups.
for each $\langle key, valuesize \rangle$ in *key_file* **do**
  group size f_{key} = $min(C[i][h_i(key)])$, $0 \le i < d$
  if $f_{key} > t$ **then**
    insert the $\langle key, valuesize \rangle$ into T
  end if
end for
calculte partition number p depending on Fumula 2
offset index $O := Convert(T)$
P:= {**for** each i **do** generate a partition file s_i, $0 \le i < p$}
for each input $\langle key, value \rangle$ in F **do**
  if there is a match in O **then**
    write the $\langle key, value \rangle$ into R
    update the corresponding offset in O
  else
    $id = H(key)\%p$
    insert the $\langle key, value \rangle$ into s_{id}
  end if
end for
for each s_{id}, $0 \le i < p$ **do**
  Grouping(s_{id}) by hashing and append to R
end for
remove *key_file* and partition set P
output R

D. Algorithm Analysis

We summarize the whole process including the three phases in Algorithm 1. The algorithm starts by creating a two-dimensional array with width w and depth d, the array is the CM sketch. Then each $\langle key, value\rangle$ pairs read from the input file F is hashed to the CM sketch to calculate the rough size of each group. In order to avoid traversing the whole input file unnecessarily in the second phase, we use an intermediate file *key_file* to record the 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 data set into $\langle key, valuesize\rangle$ format, by which we can reduce the I/O cost and the computation of accurate group sizes in the second phase.

The threshold t between the big groups and small groups is calculated depending on the CM sketch and the ratio r of big groups(Line 12), we first sort one row of the two-dimensional array C in descending order, 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 insert to the group size table to calculate the accurate groug size like Formula 1. The other $\langle key, valuesize\rangle$ pairs are accumulated to get the total size of small groups. We can get the group size table after traversing the *key_file*, the partition number p can be also calculated like Formula 2.

The $Convert$ function converts the group size table to offset index (Line 22), in more details, traverses the elements in table T from top to down and then accumulate the group sizes as the 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 to 0, it means that the first kv-pair mapped to the group will be stored in the beginning of the result file. Define the group size of the previous entry as $group_{pre}$, which means that the size of the previous group is $group_{pre}$. Define the initial offset of the previous group as off_{pre}. So the offset of current entry off_{cur} equals to the sum of off_{pre} and $group_{pre}$. After calculate the offset of each group, we will obtain an in-memory offset index which contains $\langle key, offset\rangle$ information.

Filling the result file phase needs another pass of the input file(line 22-25). For a kv-pair, get the offset off_{key} by searching the offset index O, if there is a match in the index, it belongs to a big group and will be written to the off_{key}^{th} bytes of the result file *R* depending on the offset, if there is no match, it will be written to the corresponding partition in *P*. With this kv-pair from input file having been output, the current offset of the corresponding group will increase the size of the kv-pair in bytes, which indicates the next kv-pair mapped to the same group will be stored in the $(off_{cur} + sizeof(\langle key, value\rangle))^{th}$ bytes of the file *R*, it is the update operation in line 25.

After grouping the big groups successfully, the small groups in each partitions stored in the disk are still unordered, these small group partitions are read into memory partition-by-partition and processed by the hash grouping, then these kv-pairs in small group partitions are appended to the result file *R* after grouping(line 31-33).

IV. Experimental Evaluation

This section presents the performance evaluation for PowerHash. We compare our work against existing typical grouping approaches, merge-sort [5] and memory-constraint hash [10]. For merge-sort, we downloaded 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 pseudo code in SQL database [17]. We used typical real data sets: (a) the Higgs[1] Twitter data set, providing the information about activity on Twitter during the discovery of Higgs-boson; (b) web-BerkStan [2] data set, a web graph containing 685,230 nodes and 7,600,595 edges; (c) Google[3] data set, a web graph data set containing 875,713 nodes and 5,105,039 edges; and a simulation data set that obey a power-law distribution Pareto. Table 1 summarizes the data sets used.

TABLE I: Data sets

dataset	size	illustration
Twitter	180.4MB	Twitter social network statistics data
web-BerkStan	102.5MB	Berkely and Stanford web graph data
Google	97.2MB	Google web graph data
Pareto	575.4MB	Pareto distributed simulation data

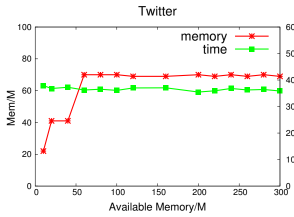
All serial methods were implemented in C++, and compiled with g++ version 4.8.4 in Linux. The experiments were executed on a machine with two quad-core Intel CPUs at 2.67GHz and 32GB RAM.

A. Overall Performance Evaluation

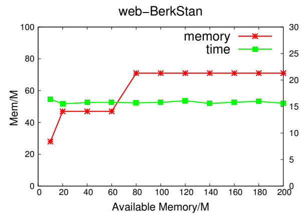
The following experiments exhibits the overall performance of PowerHash and its performance comparison against merge-sort and memory-constraint hash. When executing PowerHash algorithm on various data sets, the most important parameter — the ratio of big groups in these data sets is set as 0.2 depending the Pareto principle, the expansion factor proposed in section III to ensure each small group partition can be processed in memory safely is set as 2.

Figure 5 exhibits the overall performance of PowerHash for different data sets as showed in Table I, it shows the real memory consumption and grouping time with the available memory varying. As the available memory varies, the grouping time is almost unchanged, 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 Figure 5a 5b 5c 5d. The reason for this phenomenon is that the available memory only influences the partition of small groups if the ratio of big groups is fixed, the grouping time of big groups keeps steady in theory no matter how much memory is available, and the grouping time of small groups is nearly the same when the partition numbers are not much different, so the whole time cost of PowerHash always keep stable.

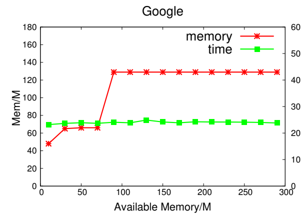
[1]http://snap.standford.edu/data/higgs-twitter.html
[2]http://snap.stanford.edu/data/web-BerkStan.html
[3]http://snap.standford.edu/data/web-Google.txt.gz



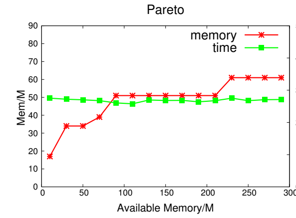
(a) Overall performance on Twitter



(b) Overall performance on web-BerkStan

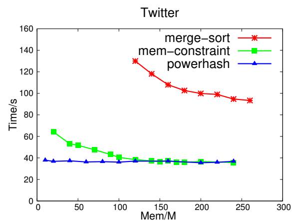


(c) Overall performance on Google

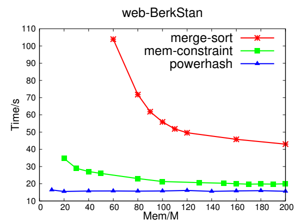


(d) Overall performance on Pareto

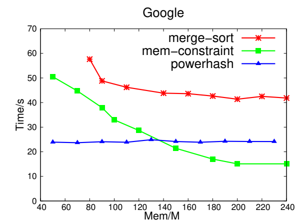
Fig. 5: Overall performance on various data sets.



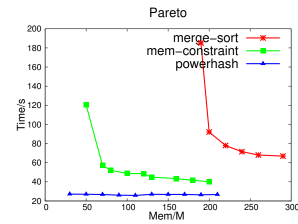
(a) Performance comparison on Twitter



(b) Performance comparison on web-BerkStan



(c) Performance comparison on Google



(d) Performance comparison on Pareto

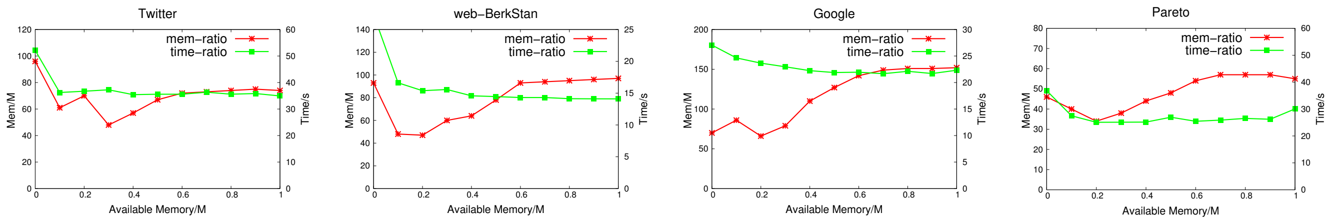
Fig. 6: Performance comparison on various data sets.

From Figure 5, we can also see that the memory consumption grows stage by stage and then stays stable. The statistical memory consumption in the line chart is the peak of memory usage in each experiment, according to the experimental results, these statistical peaks are always the space occupied by the hash table in small groups hash grouping process. So each jump of memory consumption represents a decrease of 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 decrease to 1. The minimal memory consumptions shown in Figure 5 reveals the offset index size in indexing-filling, e.g., it is only 22MB as shown in Figure 5a, in other words, PowerHash can achieve its best performance with just about 22MB 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 the memory available increasing, we list experimental results on different data sets. On the whole, our algorithm is almost distributed lower than other algorithms in Figure 6. Considering the extremely large input size, we would like to compare them with limited memory. When the memory is limited, our algorithm performs faster in the case of the same memory consumption, and it takes up less memory under the same time cost, the available memory smaller the advantage more obvious. For merge-sort and memory-constraint hash, the execution time continues to decline with the memory available increases constantly, when the memory is large enough to process kv-pairs in memory completely, the performance is no longer enhanced and grouping time stays stable, at that time memory-constraint hash degenerates into the pure hashing grouping method. The merge-sort needs more than 10 times memory and memory-constraint hash needs 5 times(i.e., 120MB) memory for Twitter to achieve the best state compared to our algorithm as shown on Figure 6a, the experimental results on other data sets are similar with a little different multiples.

Compared to merge-sort and memory-constraint hash, the runtime gap is greatest at the beginning of the line charts, in which case the runtime of memory-constraint hash is at least 2 times than PowerHash with the same memory on the four data set, the multiples of runtime between merge-sort and PowerHash is greater. Then their gap continues to decline with the available memory increasing. Even if 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 Figure 6b 6d. Memory-constraint hash can achieve the same performance on Twitter and Google, its time consumption is also longer than PowerHash with the same memory before this point where they have the same runtime in Figure 6a 6c, it can perform better than PowerHash on Google when the memory is sufficient.

We make the following analysis to this phenomenon: recall to section II, for merge-sort algorithm, there is unnecessary computational overhead in aggregating kv-pairs, the sort process reduces its performance, its time cost is always highest even if the memory is enough. For memory-constraint hash, one or more buckets will be selected to spill to disk when the memory used has reached the threshold, these spilled partitions will be read back to re-aggregate subsequently. The power-law distributions makes it more difficult to divide the work data into small uniform portions. For a larger bucket spilled to disk, subsequent reading back and re-aggregating may lead to recursively execute the algorithm many times when the memory is limited, the I/O overhead increases sharply. However, PowerHash deals with the big groups and small groups separately, the big groups processed by indexing-



(a) Ratio test on Twitter (b) Ratio test on web-BerkStan (c) Ratio test on Google (d) Ratio test on Pareto

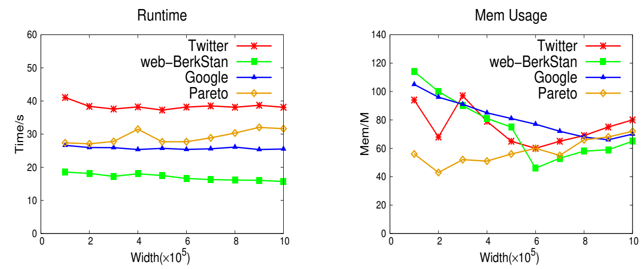
Fig. 7: Ratio test on various data sets.

filling, and the small groups are processed via partitioned hash grouping approach, the two grouping method used to process the big groups and small groups can be carried out with limited memory and avoid repeat access to disk as introduced in section III, so the performance of memory-constraint hash is slower than PowerHash with the same limited memory. If there is enough memory, all kv-pairs can be loaded in memory, memory-constraint hash degenerate into the pure hashing grouping method, its performance may be faster than PowerHash as shown in Figure 6c. Therefore, our algorithm is able to more efficient in limited memory compared to other algorithms.

B. Parameters Evaluation

In the work, we also evaluate the impact on different parameters of PowerHash. We have experimentally evaluated on various data sets in Table I and theoretically analyzed the effects of these parameters on the algorithm. Figure 7 shows the strong scalability results for the algorithm.

The ratio of big groups. The ratio of big groups is the most important parameter in PowerHash, it determines the division between big groups and small groups. Compared with the real ratio of big groups in the data sets, a smaller ratio may lead to the unbalance of small group partitions because some big groups are judged as small groups, and then cause the redivision of small groups; a bigger ratio may result in a great offset index of big groups and then causes the waste of memory, so the selection of ratio is important. As shown in Figure 7, the ratio is set from 0 to 1, the available memory is fixed and set as 50MB, PowerHash becomes the partitioned hash method when the ratio is 0, it turns to the indexing-filling method when the ratio is 1, their memory usage is much larger than 50MB in both cases and the out-of-memory problem will occur. With the ratio varying, the execution time decreases and then stays stable, the memory usage continues to decline and then continues to increase, the point with least memory usage represents the offset index size and small group partition sizes are the most suitable, and the memory usage is under 50MB in this case, i.e., the algorithm can be executed in memory smoothly without out-of-memory problem. In Figure 7b 7c 7d, the memory usage is lowest when the ratio is 0.2, this phenomenon satisfies the Pareto principle. In Figure 7a, our algorithm performs best when the ratio is 0.3. The experiment on these data sets can reflect the most appropriate ratio of big groups is the value around 0.2.



(a) Runtime with different width (b) Memory usage with different width

Fig. 8: Runtime and memory usage with different width of the CM sketch.

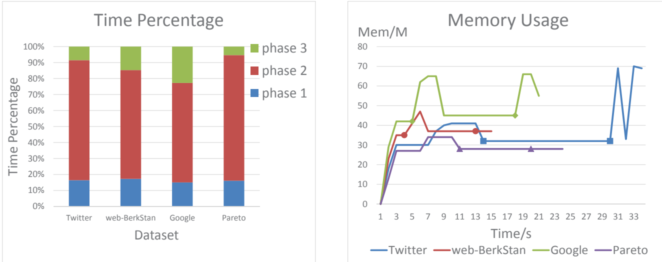
The parameters of CM sketch. The next experiment shows the effect of the CM sketch parameters. Recall from Distinction phase in section III that the CM sketch are employed to reflect the distributions of group sizes, we sort a row of the CM sketch and then obtain the threshold between big group sizes and small group sizes according to the big groups ratio, then divide the big groups and small groups based on their rough sizes, so the width and depth of CM sketch can influence the distinction of big groups and small groups. According to the CM sketch introduction in section II: if the error of group sizes is within a factor of ε with probability δ, the depth is $\lceil ln(1/\delta) \rceil$, the width is $\lceil e/\varepsilon \rceil$, the depth can be set easily. The width of CM sketch determines the accuracy of threshold, it directly affects the division between big groups and small groups in Distinction phase. Figure 8 shows the memory usage and time cost on different data sets with the width varying.

The runtime on each data set is almost steady as shown in Figure 8a, the memory usage is influenced by the width of CM sketch as shown in Figure 8b, it is appropriate to set the width from 200000 to 800000 with little memory waste.

C. Phase Evaluation

In this subsection, we discuss the impact and interrelated factor on *groups distinction*, *big groups grouping*, and *small groups grouping*. 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 data sets. We can notice that the time percentage of groups distinction phase is close to 15%, i.e., the preparation time of key grouping operation occupies 15% of the total time. The big groups grouping phase takes up most of the total time, the small groups grouping phase groups the small groups with



(a) Time percentage of each phase (b) Memory usage of each phase

Fig. 9: Runtime and memory usage comparison between each phase.

little time cost. This phenomenon satisfies expected result, the third phase is finished in memory so its time cost is small, the majority of the data set is processed in the second phase, and lots of time is spent on seeking operations, so its time percentage is highest.

We also compare the memory usage for the three phases on various data sets as depicted in Figure 9b. The mark point on each line separates the three phases. The memory usage continues to increase and keeps stable in the first phase, the memory usage in stable state is the size of CM sketch. Then the memory usage increases over time gradually in the second phase, its growth represents the generation of offset index, it decreases after the memory occupied by the CM sketch being released. In the third phase, the memory usage is also steady because the partition of small groups is balanced and each hash table in grouping process is almost the same.

V. Conclusions

Grouping key value pairs are essential for many practical applications that include MapReduce, SQL Hash Group By, etc. The volume of such data increases rapidly, therefore, it is essential to improve fast key value grouping methods. In this paper we proposed PowerHash, a method that supports very long records, large data sets and works efficiently even if the memory is very limited and is easily implemented. Extensive experimental evaluation with real data sets whose group sizes follow the power-law distributions revealed that our method is much more efficient than existing ones in terms of speed and computational resources, it can reduce the repeat access to disk caused by the distribution unbalance effectively when grouping these data sets. When the available is limited, our algorithm runs faster than merge-sort and memory-constraint hash with the same memory. The merge-sort and memory-constraint hash need several times memory to achieve the best state compared to our algorithm, it can help us save a large amount of memory when executing key grouping operation. This work is part of a large project that aims to develop an engine for grouping and processing of massive data. We are currently working on scaling our method to the distributed computing framework. As we know, stand-alone aggregating key value pairs of different groups is applied to both map phase and reduce phase in MapReduce independently. So PowerHash is easily parallelizable. We are also focusing on the parallel processing various types of grouping using the PowerHash.

References

[1] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[2] S. M. Stephens, J. Y. Chen, M. G. Davidson, S. Thomas, and B. M. Trute, "Oracle database 10g: a platform for blast search and regular expression pattern matching in life sciences," *Nucleic Acids Research*, vol. 33, no. suppl 1, pp. D675–D679, 2005.

[3] A. MySQL, "Mysql 5.1 reference manual, 2006," *Accessible in URL: http://dev. mysql. com/doc*, 2009.

[4] B. Momjian, *PostgreSQL: introduction and concepts*. Addison-Wesley New York, 2001, vol. 192.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[7] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 247–260.

[8] B. Li, E. Mazur, Y. Diao, A. Mcgregor, and P. Shenoy, "A platform for scalable one-pass analytics using mapreduce," in *ACM SIGMOD International Conference on Management of Data*, 2011, pp. 985–996.

[9] L. Lin, V. Lychagina, W. Liu, Y. Kwon, S. Mittal, and M. Wong, "Tenzing a sql implementation on the mapreduce framework," 2011.

[10] D. Bartholomew, "Mariadb vs. mysql," *Dostopano*, vol. 7, no. 10, p. 2014, 2012.

[11] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database tuning advisor for microsoft sql server 2005: demo," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005, pp. 930–932.

[12] G. Cormode, "Count-min sketch," *Encyclopedia of Algorithms*, vol. 29, no. 1, pp. 64–69, 2009.

[13] J. M. Hellerstein and J. F. Naughton, "Query execution techniques for caching expensive methods," *Acm Sigmod Record*, vol. 25, no. 2, pp. 423–434, 1996.

[14] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 137–148.

[15] A. Boicea, F. Radulescu, and L. I. Agapin, "Mongodb vs oracle-database comparison." in *EIDWT*, 2012, pp. 330–335.

[16] K. Bratbergsengen, "Hashing methods and relational algebra operations." in *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, 1984, pp. 323–333.

[17] C. Freedman, "Hash aggregate," https://blogs.msdn.microsoft.com/craigfr/2006/09/20/hash-aggregate/, 2006.

[18] M. Newman, "Power laws, pareto distributions and zipf's law - contemporary physics - volume 46, issue 5," *Contemporary Physics*.

[19] F. Auerbach, "Das gesetz der bevlkerungskonzentration," *Petermanns Geographische Mitteilungen*, vol. 49, no. 1, pp. 73–76, 1913.

[20] K. George and K. George, *Human behavior and the principle of least effort: An introduction to human ecology*. Addison-Wesley Press,, 1949.

[21] L. Adamic, "The nature of markets in the world wide web," *Quarterly Journal of Electronic Commerce*, vol. 1, no. 1, 2000.

[22] R. A. K. Cox, J. M. Felton, and K. H. Chung, "The concentration of commercial success in popular music: An analysis of the distribution of gold records," *Journal of Cultural Economics*, vol. 19, no. 4, pp. 333–340, 1995.

[23] R. Kohli and R. K. Sah, "Market shares: Some power law results and observations," *Working Papers*, 2004.

[24] J. C. Willis, G. U. Yule, J. C. Willis, and G. U. Yule, "Some statistics of evolution and geographical distribution in plants and animals, and their significance." *Nature*, vol. 109, no. 2728, pp. 177–179, 1922.

[25] V. Pareto, "Cours d'economie politique," *Reprinted As A*, no. 4, 1965.

[26] G. P. Box and R. Danielmeyer, "An analysis for unreplicated fractional factorials," *Technometrics*, vol. 28, no. 1, pp. 11–18, 1986.

[27] M. Gen, R. Cheng, and S. S. Oren, *Network Design Techniques Using Adapted Genetic Algorithms*. Springer London, 2000.

[28] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *LATIN 2004: Theoretical Informatics*, M. Farach-Colton, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 29–38.