1. **背景**
   1. **Count-Min Sketches**

**Algorithm 1** Count-Min sketch

initialize

initialize ,

{there are two actions of this algorithm}

**action 1：Counting**

{while an update arrives}

for j := 1 to d do

**action 2：GetFrequency()**

{while an item arrives}

f() min(|)

output f()

* 1. **bHash**

首先，我们了解了当前key grouping(Group By)的研究现状。第二，我们提出了一种高效且节约内存的key grouping算法Count-Min bHash，算法利用Count-Min Sketch实现big group和small group 的区分，然后采用bHash算法中的offset hash table实现对big group的跟踪读写而不需要将其全部存入内存，大大节省了内存；与此同时，由于hash table中保存big group的key-offset对，这也可以大大减少内存消耗，这一点对于skewed data更是明显。big group处理完成之后，利用Memory-Constraint hash grouping 算法实现small group 的纯基于内存的key grouping，由于这时只处理small group，进行纯基于内存的hash group并不需要在内存中临时存储big group，所以内存消耗比较小，如果想内存消耗不超过预估水平，可以利用hash函数对small group进行分块，然后分别处理these partitions。我们也分析整个算法的效率与内存消耗情况。第三，我们将Count-Min bHash算法与现有的key grouping算法如merge-sort、memory-constraint hash进行比较，我们的算法内存消耗比上面三种算法都少，这一点在处理大数据集时尤为明显；时间消耗仅次于bHash。

**Our contribution.**算法利用了数据集中group size的skewed分布的特点，也就是数据集中大部分group是small group，而big group只占少部分，于是提出了将big group和small group进行分开处理的思想，同时为了避免将group装入内存而造成的内存不足的问题，提出使用hash table存储位序信息的方法。其中将big group和small group进行区分是利用了Count-Min Sketch算法，同时利用hash table存储group的位序信息，在进行key grouping时，可以将kv-pairs直接写入输出文件的正确位置，这样就可以避免将group存入内存而造成内存不足的问题，这一方法称之为bHash，所以本篇论文提出的算法称之为Count-Min bHash算法。传统的key grouping算法是将整个数据集中的big group和small group进行统一处理，如果采用Memory-Constraint hash grouping方式，那么内存中就要存放groups, 一旦内存不足，就需要将input dataset通过hash的方式进行分块，以便内存能够存下当前分块的的所有group,一旦数据集中某一group特别大，大到内存都无法存储，那么Memory-Constraint hash grouping方式就无法正常运作，如果采用merge-sort key grouping方式，这种算法虽然能够克服内存不足的短板，但是由于算法自身效率不高，所以有时难以满足实际需求。Count-Min bHash算法利用hash table存储每个group的位序信息，所谓位序信息就是每个group在最终输出文件中的写入位置。如果对数据集中的big group和small group进行统一处理，将它们的位序信息全部存入hash table中，那么这个hash table的size也是十分巨大，有时甚至无法装入内存，这样就需要采用Memory-Constraint hash grouping方式中对数据集进行分块的方法，增大了整个算法的IO，降低了算法的效率。Count-Min bHash算法就是根据数据集中group size的skewed分布的特点，将big group和small group的位序信息分开存储，big group的·位序信息存储在offset hash table中, small group的信息存储在offset accumulate table中，由于数据集中big group只占小部分，相比于将所有group的位序信息都存储在一份hash table中，big group所对应的offset hash table的size大大减小，同时提高了检索效率，也就是提高了算法的效率。另外，算法利用small group可以放入内存中快速处理的特点，将small group进行分组处理，每组small group大小适当，每组small group的位序信息保存在offset accumulate table中，这样offset accumulate table的size也会大大减小，同时也保证每组small group可以完全放入内存。在对small group进行key grouping时，每次根据offset accumulate table中的位序信息读取一组small group放入内存，在内存中最大效率的完成key grouping，由于big group和small group分开处理，在处理small group时不用担心big group在内存中存储不下的问题。Count-Min bHash算法主要针对的是进行key grouping时内存不足的问题，整个算法需要对input dataset进行两次遍历，对中间结果进行一次遍历，算法在减小内存消耗的同时，算法的整体效率相对Memory-Constraint hash grouping几乎持平。

1. **算法**

In this section we present out main algorithm for Count-Min bHash in detail.整个算法需要将原始数据遍历2遍，对生成的中间结果遍历一遍，可以将整个算法分为3个阶段。第一阶段是rough counting，第二阶段是offset generation，第三阶段是key grouping。Rough counting阶段是利用Count-Min Sketch算法粗略统计每个group的size，这是为了了解整个数据集的整体分布，与此同时，这一阶段还会生成中间文件key\_file，中间文件中包含每一个kv-pair的key以及对应的value size信息。这一阶段获取的每个group的rough size信息可以帮助我们区分big group和small group，以便后续阶段处理这两类group。（这一步分为三小步，第一，遍历整个数据集，通过Count-Min sketch算法对每个group size进行粗略的统计；第二，将原始数据集通过hash函数分块；第三，将原始数据集中的<key，value>对转化为<key，sizeof(value)>对并通过中间文件key\_file保存下来。）Offset generation阶段遍历key\_file，根据上一阶段获取的rough group size判断当前group是big group还是small group，然后生成存储big group位序信息的offset hash table和存储small group位序信息的offset accumulate table，其中offset hash table只保存了big group在最终输出文件中的位序信息，offset accumulate table中只保存了small group在最终输出文件中的位序信息。Key grouping阶段，根据上一步中获取的offset hash table 和offset accumulate table，将<key，value>对直接写入结果文件的正确位置，这样就完成了key grouping。整个算法中，前两个阶段是第三阶段key grouping的基础，第三阶段中kv-pairs能够准确的写入结果文件的正确位置就是因为第二阶段生成的offset hash table和offset accumulate table。将整个数据集中的group分为big group和small group，并且将它们分开处理的根本原因就是为了减小算法的内存消耗，这是利用了数据集中group size呈skewed分布的特点，最直接的表现就是相对于传统的key grouping算法中巨大的hash table，本论文中offset hash table和offset accumulate table的size要小的多。为了减小整个算法IO的次数，算法在第三阶段利用了output buffer，output buffer将大量小范围的随机读写转化为少量的大范围顺序读写，大大提高了IO效率。此外，第一阶段中将input dataset进行分块然后便于在第三阶段中分块处理，将数据集进行分块处理以后，我们可以减小每个分块中group的数目，换言之也就是每个分块中拥有相同key的kv-pairs排列更集中，有利于第三阶段中kv-pairs向结果文件的写入操作。下面，我们会对算法的三个阶段进行详细介绍。

In this section we present out main algorithm for Count-Min bHash in detail. Count-Min bHash needs two passes of the whole set of kv-pairs and one pass of the intermediate result, it can be divided into three phrases. The first phrase is rough counting, the second phrase is offset generation and the third phrase is key grouping. Rough counting phase studies the global knowledge of group size information through the Count-Min sketch algorithm and generates the intermediate result key\_file that contains every key and value size information. In this phrase, we can get every group’s rough size, these group size can help us distinct the big groups and small groups which will be handles separately. Offset generation traverses the intermediate result and combines the rough group size, then generates a offset hash table and a offset accumulate table that contain ⟨group-key; file-position⟩s, and the offset hash table only holds the big groups’ position information in the final output files, the offset accumulate table only holds the small groups’ position information in the final output files. Key grouping phrase writes out kv-pairs to certain file positions according to the offset hash table and offset accumulate table. The key grouping phase is dependent on the first two phrases, i.e., in the third phase, the kv-pairs are stored directly in the grouping file according to the offset hash table and offset accumulate table in the second stage. To reduce the memory consumption, we take advantage of the skewed distribution of group size in the dataset and handle the big groups and small groups separately, e.g., compared with the huge hash table in the traditional key grouping algorithm, the size of offset hash table and offset accumulate table in this paper are much smaller. To reduce the number of I/Os, we exploit an output buffer design in the third phase which convert a large number of small random writes to less number of big sequential writes. Furthermore, by partitioning the kv-pairs groups in the first phase and processing kv-pairs partition-by-partition in the third phase, we can constrain the key range in each partition and make the kv-pairs more concentrate on a few keys in each partition of groups. In the following, we describe the three phases in detail and provide a complexity analysis.

* 1. **Count-Min bHash**

1. Phrase 1: Rough Counting

算法的第一阶段主要是为了了解整个数据集的分布情况，同时为后续阶段的工作做准备。这一阶段需要对input dataset进行一次遍历，然后完成三个任务，三个任务的实现都是建立在对input dataset的遍历的基础之上。第一，Rough counting of group size，在遍历input dataset数据集的同时，通过Count-Min Sketch算法粗略统计每个group的size, 获取rough group size是为了区分数据集中的big group和small group, 第一个任务就相当于对整个数据集的分布进行探测；第二，KV-Pairs Reorganization，这一任务是为了将数据集进行分块，每一块子数据集中的group数量会变少，每一个group中的kv-pairs会排列的更加紧凑，可以加快第三阶段中将kv-pairs写入结果文件，这一任务是通过hash的方式完成的，这样属于同一group的kv-pairs肯定会被分入同一partition。第三，Record Value Size，通过Count-Min sketch算法获得的group size不是准确的size, 但是本论文提出的算法必须要知道每一个group的准确大小，所以需要计算每一对kv-pairs的size，为了减小IO数量与IO的代价，在第一阶段遍历input dataset时，生成一个相对input dataset小的多的中间文件key\_file，其中记录的每个key和key对应的value的size，即以<key; sizeof(value)>的形式保存每一对kv-pairs的size信息，这样既可以减少读取input dataset的次数，又可以使下一阶段更高效的统计每一个group的size。下面来详细介绍第一阶段。

**Rough counting of group size.** 这是算法第一阶段的第一个任务，任务的目的是为了study数据集的分布。根据调查显示，很多数据集group size的分布是skewed分布，其中big group只占少部分，大部分group是small group，如果对dataset中的group进行统一处理，那么在第二阶段中获取的offset hash table将会十分巨大，不仅消耗巨大内存，而且影响算法的执行效率。但是如果将big group和small group分开处理，分别用offset hash table和offset accumulate table来保存big group和small group的位序信息，那么用于保存big group位序信息的offset hash table只需要占用少许空间，因为只有小部分group是big group；此外，为了存储small group的位序信息，会将small group进行分组，每一分组是由多个small groups组成，以每组为单位将位序信息保存在offset accumulate table中，方便后续phrase同时处理多个small groups, 这样offset accumulate table的size也会很小。为了区分big group和small group，所以需要了解各个group的大小，但是区分时并不需要严格的了解各个group的大小，所以采用Count-Min sketch算法粗略的统计各个group的大小。我们通过设置一个合适的阈值，将group size大于阈值的称为big group，小于阈值的称为small group。虽然Count-Min sketch算法中hash 函数会产生碰撞，从而导致有些group size偏大，但是这并不影响结果，因为Count-Min sketch算法统计获得的group size能够保证，被判断为small group 的肯定是small group，big group肯定不会被判断为small group。最后虽然会有部分small group被误判为big group，但是并不影响整个key grouping的效率。

**Rough counting of group size.**

**KV-Pairs Reorganization.** Recall that there is an output buffer (in the file filling phase) which aims at merging a large number of small random writes to less number of big sequential writes. If the random output kv-pairs are more concentrate on a key range, they are more likely to be continuously written out. That is, they can be merged together as less number of big sequential write. To improve the output buffer efficiency in the file filling phase, we should reorganize the read-in kv-pairs in a smart way. Our idea is to partition the kv-pairs and assign the kv-pairs that are in the same partition with close write-out positions, such that the partition-by-partition processing of kv-pairs in file filling phase is likely to result in big sequential writes. Specifically, we partition the kv-pairs into multiple partitions. This can be achieved by hashing, i.e., multiple ⟨key，valuegroup⟩s sharing the same Hg(key) are grouped together, where Hg() is a hash function. The keys in the same partition are assigned with close write-out positions. These kv-pairs in the same partition are written out in a spill as shown in Figure 1. Note that, the recorded write-out position is not for the spill but for the final output file (i.e., grouping spill in Figure 1) with grouped kv-pairs.

**Record Value Size.** Count-Min bHash算法真正进行key grouping时在第三阶段，它进行key grouping的过程是将kv-pairs直接写入output files的正确位置，这些正确的写入位置就是第二阶段生成的offset hash table和offset accumulate table,但是offset hash table和offset accumulate table的是根据每个big group以及每组small group的准确size转化得到的，所以获取每个group的size是算法必须执行的一个过程。为了减少IO的次数，同时减小IO的代价，在第一阶段遍历input dataset的同时，将每一对kv-pairs，即<key，value>对转化为<key, sizeof(value)>保存在中间文件key\_file中，显而易见，key\_file的size肯定不比input dataset大，在很多种情况下，key\_file都比input dataset小的多，这样在后面阶段统计每一个group的准确size时，只需遍历key\_files根据key和其对应的sizeof(value)就可以获取每个group的准确size，大大减小了IO代价。

**Record Value Size.**

1. Phrase 2: Offset Generation

第二阶段主要是为了获取将group写入output files的位序信息，即生成offset hash table和offset accumulate table。这一阶段分为两步，第一步，counting Group Size，这一步会利用第一阶段的中间结果，分别统计big group和small group的group size，将其存储在group size table中，第二步，offset table Generation，这一步根据group被读取的先后顺序，将group size table转化为offset hash table和offset accumulate table。Count-Min bHash算法的核心就是第二阶段，它将big group和small group分开处理，然后通过hash table存储每个big group或每组small group的位序信息，这样在进行key grouping过程时，无需将big group整个存入内存，或者在内存允许的情况下，一次处理多个small groups, 这样做既避免了内存不足的风险，也提高了对small groups进行key grouping的效率，这也是整个算法的核心思想。正是由于将big group和small group分开处理，才可以最大程度的减小算法的内存消耗，同时提升算法效率。本算法中的位序信息就是每个group在最终输出文件中的写入位置，只要知道每个group的最终输出位置，那么就可以避免处理group本身，特别是big group。算法中利用hash table存储group的位序信息，如果将big group和small group统一处理，将它们的位序信息统一保存在hash table中，那么hash table的size将十分巨大，很可能造成内存不足。由于big group数量少size大，small group数量大但是size小，所以hash table中大部分内容是small group的位序信息，如果能够将big group和small group分开处理，big group的位序信息单独保存在一份hash table中，那么hash table的size会大大减小，另外，内存可以同时处理多组small groups，如果对small group进行分组，利用基于纯内存的key grouping方法每次处理一组合适大小的small group,那么对于small group的位序信息只需要保存每组的位序信息即可，这样不仅可以减小small group对应的hash table的size, 还可以提高small group的key grouping的效率。

**Counting Group Size.** Counting group size是为了分别统计big group和small group的准确size，并将这些group size保存在hash table中。第一阶段中获得了每个group的rough size，如果人为设置一个阈值，将rough group size高于阈值的称为big group，低于阈值的称为small group。big group和small group的判断精确率与Count Min Sketch算法的参数设定有关。group的精确size的结果根据kv-pairs在输出文件中的格式而异，其计算方法就是将group中kv-pairs和格式控制符的size累加。计算精确group size时，对于big group 和small group有不同的处理方式：每个big group都需要计算其精确group size，将每组<key, sizeof(groupkey)>置于group size hash table中；对于small group, 由于small group的数量特别多，需要对small group进行分组，分于同一组的small group，我们统计其group size之和，然后将每组group size之和置于对应的accumulate table中。由于数据大多呈现倾斜分布，其中big group数量不多却占用大量存储空间，而small group数量多占用存储空间少，这样单独处理big group就可以避免产生一个超大的group size hash table，此外，将small group分组置于accumulate table中，这也是利用了small group 量大而占用存储空间小的特点，这样方便后续对small group进行key grouping操作时，可以将每组small group中的多个small groups一次性读入内存进行hash grouping，不仅内存利用率高，而且效率大大提升。

具体做法如下，遍历key\_file，通过key获取Count-Min sketch统计的每个group的粗略大小。我们设置一个group size阈值，如果group size大于阈值，则被判断为big group，然后将<key, sizeof(value)>添加至group size hash table中，通过value size的累加可以得到每个big group的精确size（单位byte），最终group size hash table中的元素为<key，sizeof(groupkey)>；如果group size 小于阈值，则通过hash函数对small group进行分组，然后添加至accumulate table，仍然采用value size累加的方式获取accumulate table中同一组的small group 的group size之和。由于accumulate table中只需记录small group中各个分组的group size之和，所以所需内存很小。整个counting group size的过程只需遍历一次key\_file。

**Offset table Generation.** 这一步前面统计了每个big group的精确group size，并将其保存在group size hash table，将small group分组，将每一组small group的group size之和存放于accumulate table中。由于知道了每个big group的size，所以在最后的输出文件中，我们能计算出每个big group的输出位置（offset），只需将group size hash table转化为offset hash table。拥有offset hash table之后，就可以实现在key grouping过程中，对big group的跟踪读写，而不需要将整个big group都存放于内存中。由于small group没有在offset hash table中没有记录其offset，所以在将group size hash table转化为offset hash table之后，需要紧跟着最后的offset，以该offset为起点，将offset accumulate table也转化的offset accumulate table，这样small group就可以写在最后输出文件的尾端，只要offset accumulate table存在，我们就可以将每个small group组再次读出，利用基于内存的hash grouping对这些small group实现key grouping。

1. Phrase 3: Key grouping

经过前两个阶段的准备工作，这一阶段就是key grouping阶段，实际上就是文件的读写阶段。遍历中间文件spills，根据offset hash table跟踪读写big group，避免了将整个big group存储于内存中；同时根据offset accumulate table将small group统一写入output file尾端。待spills遍历完成之后，big group也已完成key grouping，但是写入output files中的small groups仍然是毫无规律，所以需要根据offset accumulate table从output files读出，利用纯基于内存的hash grouping完成small group的key grouping，然后再将结果根据offset accumulate table写回至output files。

**Big Group GroupBy.** 上一阶段得到的offset hash table和offset accumulate table中存放的是将big group和small group写入输出文件的起始位置，这时只需要遍历中间文件spills，然后检测offset hash table中是否有对应的offset信息，如果有，则将kv-pairs根据相应的offset写入输出文件；如果没有相应的offset，说明该group为small group，然后从offset accumulate table获取其对应的offset信息，然后将kv-pairs也写入相同的位置。每次向output files中写入kv-pairs，都需要更新相应的offset信息，这样就可以保证所有kv-pairs能够无错的写入。

**Small Group GroupBy.** 遍历完spills文件以后，big group完成了key grouping，以正确的形式写入output files中，但是small groups没有像big groups那样为每个group建立一个<key，offset>对，而是将small groups进行分组，多个small groups毫无规律的写入output files，所以接下来的任务就是对这些没有实现key grouping的small groups进行groupBy。只需要根据最初的offset accumulate table将small groups从output files读入内存，然后利用hash grouping算法完成small groups的key grouping，完成以后再根据offset accumulate table将small groups 的key grouping结果写回到output files。

经过上面三个阶段就实现了key grouping过程，三个阶段中，第一阶段利用Count-Min sketch算法粗略统计各个group的group size，同时生成中间文件key\_file和spills，这一阶段是为后面两个阶段做准备；第二阶段遍历key\_file，

根据上一阶段统计的group size来区分group是big group或small group，然后根据key\_file中value size的信息，统计每个big group的精确group size和每组small groups的精确size，再将这个group size转化为offset信息，其中big group的offset信息存放于offset hash table中，每组small groups集合的offset信息存放于offset accumulate table中。第二阶段获取的offset信息可直接用于第三阶段key grouping中最终结果的读写；第三阶段就是利用上一步中offset信息，遍历spills文件，根据其中kv-pairs的信息，查看offset hash table和offset accumulate table，将每一条kv-pair写入output files，然后在利用offset accumulate table中的offset信息将没有进行key grouping的small groups利用hash grouping算法完成key grouping操作，并且根据offset信息将key grouping完成后的small groups写回至output files。

1. Making Them Together

We summarize the whole process including the three phases in Algorithm 2. 算法第一阶段主要是study整个数据集得分布，获取所有group的rough size，在第二阶段结合第一阶段得得到的rough group size，将group分为big group和small group，然后计算出每个big group的group size和每组small group的sum of group size，然后将group size 转化为output offset。以上两个phrases都是为了第三阶段的key grouping做准备。算法首先执行第一阶段，开始创建a two-dimensional array C with width w and depth d，然后遍历整个input dataset，每读取一条kv-pairs——<key,value>，将<key, sizeof(value)>写入中间文件key\_files，然后根据Count-Min Sketch算法（Algorithm 2）中的counting操作，对key对应的group进行累加，最终就可以得到每个group 的rough size, 这里的rough group size只是group中kv-piars的数目，这些rough group size都会存放在二维数组C中。第一阶段的最后就是利用hash方法对input dataset进行分块，首先创建m个sub files，再通过hash函数计算hash值H(key)，然后根据公式（1）判定当前kv-pairs应该写入哪一个spill，再将kv-pairs写入该spill。

spill id = (H(key)%b)/(b/m) （1）

hash bucket id = H(key)%b （2）

公式（1）中b代表用于存储big group位序信息的offset hash table的hash buckets的数目，公式（2）中H(key)%b就表示当前kv-piar将会被存储的hash bucket的id。由于input dataset被分区，hash table也被分为m个部分，每一部分包含b/m个hash buckets，也就是每个spill文件中的kv-pairs只会存储再b/m个hash buckets中，例如spill1中所有的kv-pairs最后都会被映射到hash table的前b/m个hash buckets中，这样在后续阶段遍历hash table时，可以减小遍历的代价，所以会采用公式（1）确定了spill id。算法第一阶段结束以后，内存中会存放一个二维数组C，数组中存放的时各个group的rough size，同时生成的中间文件key\_file和input dataset的分区文件spills被保存在外存中，作为第二阶段和第三阶段的准备阶段，这些中间结果都会在后续被用到files(as shown 7-14 in algorithm 2).。

通过第一阶段的准备，第二阶段就可以生成直接将kv-piars写入输出文件的位序信息offset table。首先创建用于保存small group位序新的accumulate table，一个二维数组A with with width m and depth a，然后遍历上一阶段产生的中间结果key\_file，每遍历一条<key，sizeof(value) >，就根据Count-Min Sketch算法的getFrequency操作，从二维数组C中获取key对应的group的rough size，根据初始设定的group size阈值t，判断该group是big group还是small group，如果group rough size大于t则是big group，那么可以通过公式（2）获取hash bucket的id，然后将<key, sizeof(value) >插入hash table T中。hash table T的视图是传统的key-value pair，其中key就是input dataset中每个group所独有的key，value就是每个group的准确size。将< key, sizeof(value) >插入的过程就是查找hash table T中是否有匹配的entry, 也就是看hash table中是否已将有key与<key, sizeof(value)>相等的entry，如果没有，则新建在hash table中新建一个entry, 该entry的key就是当前key, value就是当前的value size in bytes，如果有，则通过公式（3）更新entry。

sizeof(groupkey)+ = sizeof(key) + sizeof(value) （3）

如果group rough size不超过t则是small group，那么会根据公式（1）和公式（4）确定small group在accumulate table A中的存储位置，将<key, sizeof(value)>插入accumulate table中，插入的过程依然遵循公式（3）更新每组small group 的size。由于input dataset已经进行分区，每一个分区文件spill中都有大量的small group，仍需要对small group进行分组，分组的方式就是通过hash方式，也就是公式（4）所写的hash函数，所以采用二维数组的方式存储small group的group size信息。(as shown 15-26 in algorithm 2)

accumulate bucket id = H(key)%a （4）

整个中间文件key\_file遍历完成以后，会得到存有big group size信息的hash table和存有small group分组size信息的accumulate table，这时需要将hash table和accumulate table转化为含有位序信息的offset table(as shown 27-28 in algorithm 2)。最终的结果文件groupSpill共m个，每个spill文件对应一个groupSpill结果文件，所以每个spill文件进行第三阶段的key grouping时，都需要参照其对应的offset table，也就是offset table信息也有m组。整个Convert过程就是将hash table和accumulate table中的group size根据遍历的顺利，从0开始逐个累加。为了将offset信息与spill分组相对应，进行累加时，hash table中每b/m个hash buckets一组，一组累加完成以后，最后的offset信息作为small group小组的起始位置，再将accumulate table中spill对应的small group小组的size 信息累加，这样一个spill文件中所有group的位序信息就通过hash table和accumulate table转化得到，这些位序信息依然保存在hash table和accumulate table中，不过这时我们将它们称为offset hash table和offset accumulate table。例如我们想得到spilli中所有group的位序信息，那么就需要hash table T第i部分的b/m个hash buckets和accumulate table A的第i行的所有small group分组大小，即从上至下遍历T[(i-1)\*(b/m)]到T[i\*(b/m)-1]的b/m个hash buckets，从左至右遍历每一个hash bucket，spilli文件中kv-pairs写入结果文件groupSpilli的起点是0，所以offset信息的起点也是0，也就是T[(i-1)\*(b/m)] 所对应的hash bucket中的第一个entry所表征的big group就是在groupSpilli中起始写入位置就是0，我们定义当前group起始写入位置是offcur，下一个group的起始写入位置是offnext，那么offnext和offcur之间的关系满足公式（5），换言之，每一个group最后写入位置所对应的offset就是下一个group的起始写入位置。

offnext =offcur + sizeof(groupkey) （5）

从上至下遍历完T[(i-1)\*(b/m)]到T[i\*(b/m)-1]的b/m个hash buckets之后，也就是spilli中的所有big group的offset信息转化完成之后，这时offcur就是hash table中T[i\*(b/m)-1]所对应的hash bucket中最后一个entry所表征的big group在groupSpilli的最后写入位置，也就是accumulate table A[i]中第一组group的起始写入位置，即A[i][0]中应该存放这个offset值。然后参照offset hash table的Convert过程，将accumulate table中A[i]的size信息转化为offset信息。

通过上面的Convert过程，hash table转化为offset hash table，accumulate table转化为offset accumulate table，

所有的group在进行第三阶段的key grouping操作时，都有了可以向结果文件groupSpill直接写入的位序信息，但是由于small group是以分组的形式进行写入，每一分组中的small group仍然没有完成key grouping操作，而且offset accumulate table在第三阶段中会有更新过程，所以有必要在此时保存offset accumulate table，以便将small group以组的形式读取然后在内存中进行key grouping。(as shown 29 in algorithm 2)

第三阶段key grouping操作就是spill文件读写的一个过程，读写完成之后，big groups就完成了key grouping,然后再对每组small group进行基于内存的groupBy操作, 这样整个key grouping任务就算实现了。第二阶段完成之后，就可以参照offset hash table和offset accumulate table，边遍历spill文件，将读得的kv-pairs写入对应得groupSpill文件得正确位置。在遍历spilli文件时，读取一条kv-pair，就会根据公式（2）来计算这一kv-pair所对应得group的位序信息所在的hash bucket id，如果在offset hash table中没有找到相匹配的key,那么说明这个group是small group，然后根据公式（4）计算其在offset accumulate table中对应的位序信息，如果此时获取的位序信息时offkey,在将kv-pair成功写入groupSpilli的正确位置之后，需要更新offkey，这样才能保证读得同一组中的small group的其它kv-pairs时，依然能够正确的写入。更新offkey参照公式（6）。如果在offset hash table中找到行匹配的key,那么就可以确定这时一个big group，直接根据相匹配的offkey信息将kv-pair写入groupSpilli，写入之后依然需要更新offkey。(as shown 30 to 47 in algorithm 2)

offkey += offkey + sizeof(key) + sizeof(value) （6）

从这里就可以看出，根据offset hash table和offset accumulate table对spill文件进行读写之后，big group的key grouping任务完成，small group全部被集中转移至各个groupSpill文件的尾部，各组small group内部仍然是无序状态，没有完成key grouping任务。这时就需要将已经在groupSpill上写好的各组small group读入内存，在内存中进行groupBy操作。之所以能够将各组small group准确的读入内存，就是因为在第二阶段将未更新的offset accumulate table保存下来。未更新前的offset accumulate table中保存的是每组small group在groupSpill中的写入位置，根据offset accumulate table将各组small group读入内存之后，可以在内存中快速的完成key grouping，只要accumulate table的参数depth a选择合适，就可以最大限度的利用内存进行key grouping，不用担心内存不足的问题。每组small group在内存中完成key grouping之后，将结果再根据offset accumulate table写入groupSpill文件，这样groupSpill中不论big group还是small group都完成的key grouping过程。 (as shown 48 to 52 in algorithm 2)

最后的groupSpill文件就是整个算法的最终结果，groupSpill文件中所有所有group都完成了key grouping任务。

* 1. **DISCUSSTION**
  2. **EXPERIMENTAL EVALUATION**

**Algorithm 2** Count-Min bHash

**Input**: File *F*, sub file number*m*, hash buckets number *b*, accumulate buckets number *a*, frequency threshold *t* ;

the parameters of Count-Min sketch: width ***w***, depth *d*

**output**: set of result files *R*

1: *C* a two-dimensional array with width *w* and depth *d*

2: *T* the hash table consists of a basic array of size *b*

3: *A* the accumulate table consists of a two-dimensional array with width m and depth *a*

4: key\_file := a file of recording the key and value’s size

5: S := {**for** each *i* **do** generate a file spilli，*0* <*i* <*m*}

6: R := {**for** each *i* **do** generate a file groupSpilli，*0* <*i* <*m*}

{phrase 1}

7: initialize

8: **for** each input <key, value>in F **do**

9: write <key, value.size()> to the key\_file

{Count-Min sketch action 1}

10: counting(key , 1)

11: calculate the hash value H(key) on key

12: spill id = (H(key)%b)/(b/m)

13: append the <key, value> to the spillid

14: **end for**

{phrase 2}

15: initialize

16: **for** each <key, value.size()> in key\_file **do**

{Count-Min sketch action 2}

17: frequency f = GetFrequency(key)

18: calculate the hash value H(key) on key

19: **if** f>t **then**

20: hash bucket idT = H(key)%b

21: insert a entry <key, value.size()> into T[ idT ]

22: **else**

23: accumulate spill id = (H(key)%b)/(b/m)

accumulate bucket idA = H(key)%a

24. inset a entry<key, value.size()> into A[spillid][ idA ]

25: **end if**

26: **end for**

27: offset hash table T’ := Convert(T) to (T’)

28: offset accumulate table A’ := Convert(A) to (A’)

29: save the copy of the offset accumulate table A’\_copy= the copy of A’

{phrase 3}

30: **for** each spilli ∈ S，0<i<m **do**

31: **for** each <key, value> in spilli **do**

32: calculate the hash value H(key) on key

33: hash bucket id = H(key)%b

34: check for a matching offset in hash table T

35: **if** there is no match **then**

36: accumulate bucket idA = H(key)%a

37: get the offset in A’[i][idA]

38: write the <key, value> to the groupSpilli

39: update the offset in A’

40: **else**

41: offset bucket idT = H(key)%b

42: get the offset in T’[idT]

43: write the <key, value> to the groupSpilli

44: update the offset in T’

45: **end if**

46: **end for**

47: **end for**

48: **for** each groupSpilli ∈ R, 0<i<m, **do**

49: **for** each accumulate bucket in accumulate table A’\_copy **do**

50: goupBy these <key, value> pairs in memory

51: **end for**

52: **end for**

53: **output** R

