Groupwise Analytics via Adaptive MapReduce

Liping Peng*
School of Computer Science
University of Massachusetts, Amherst
Email: lppeng@cs.umass.edu

Kai Zeng*
Computer Science Department
University of California, Berkeley
Email: kaizeng@cs.berkeley.edu

Andrey Balmin*
Platfora, Inc.
Email: andrey@platfora.com

Vuk Ercegovac*
Google, Inc.
Email: vuk.ercegovac@gmail.com

Peter J. Haas
IBM Research – Almaden
Email: phaas@us.ibm.com

Yannis Sismanis*
Google, Inc.
Email: yannis@google.com

***Abstract*—Shared-nothing systems such as Hadoop vastly simplify parallel programming when processing disk-resident data whose size exceeds aggregate cluster memory. Such systems incur a significant performance penalty, however, on the important class of "groupwise set-valued analytics" (GSVA) queries in which the data is dynamically partitioned into groups and then a set-valued synopsis is computed for some or all of the groups. Key examples of synopses include top-k sets, bottom-k sets, and uniform random samples. Applications of GSVA queries include micro-marketing, root-cause analysis for problem diagnosis, and fraud detection. A naive approach to executing GSVA queries first reshuffles all of the data so that all records in a group are at the same node and then computes the synopsis for the group. This approach can be extremely inefficient when, as is typical, only a very small fraction of the records in each group actually contribute to the final groupwise synopsis, so that most of the shuffling effort is wasted. We show how to significantly speed up GSVA queries by slightly modifying the shared-nothing environment to allow tasks to occasionally access a small, common data structure; we focus on the Hadoop setting and use the "Adaptive MapReduce" infrastructure of Vernica et al. to implement the data structure. Our approach retains most of the advantages of a system such as Hadoop while significantly improving GSVA query performance, and also allows for incremental updating of query results. Experiments show speedups of up to 5x. Importantly, our new technique can potentially be applied to other shared-nothing systems with disk-resident data.**

I. Introduction

Modern enterprises have two primary tools for extracting insights from their petabyte-scale data archives: parallel processing and data synopses. Parallel processing in shared-memory or shared-nothing environments exploits heavy-duty hardware to allow analysis of massive data sets, and is increasingly being provided on a pay-as-you-go basis in computing-as-a-service environments. Often, the goal of parallel processing, especially in the context of exploratory analysis, is to compute *data synopses*, that is, small, lossy summaries that capture important characteristics of the data [1]. One common type of synopsis is a *top-k* or *bottom-k* synopsis; such subsets of the data capture data elements that are "significant" with respect to some criterion (wealthy citizens, costly insurance claims, nearest neighbors). An equally important kind of synopsis is a *uniform random sample*, which allows a user to get a feel for the data by manually inspecting a small set of representative elements. Importantly, a wide variety of analyses can be executed on a sample to quickly approximate the result of the analysis when applied to the entire dataset; such estimates can often be accompanied by probabilistic error bounds.

Shared-nothing environments with MapReduce-based infrastructures such as Hadoop are increasingly being used to collect, store, and analyze massive data, e.g., to perform the type of synopsis computation described above. Hadoop, in particular, is attractive because it is open-source, free, and can scale to hundreds or even thousands of nodes and many petabytes of data. A perhaps even more important reason for the pervasiveness of Hadoop installations is the simplicity of its parallel programming model, especially relative to traditional models based on asynchronous communication between nodes (e.g., via message passing systems such as MPI). Unfortunately, the simplicity of specifying analytics tasks in Hadoop is often obtained at the cost of query performance.

Specifically, consider *groupwise set-valued analytics (GSVA)* queries. For a GSVA query, the dataset D is first dynamically partitioned into $g \geq 1$ *groups* $G_1, G_2, \ldots, G_g$ (also called *strata*) such that $D = \bigcup_{i=1}^{g} G_i$. The partitioning is based on attribute values of the data elements as in a classic SQL GROUP-BY operation. Then a synopsis is computed for each group G_i by applying a set-valued query. In the case where the synopsis is a random sample, the process is usually called *stratified sampling*, and one can analogously refer to *stratified top-k* and *stratified bottom-k* computations. For the latter two types of queries, each tuple is assigned a *weight*, and then, for each group, the set of records with the k highest weights (top-k) or k lowest weights (bottom-k) are returned. In the context of micromarketing, for example, we might want to stratify the population by zip code and age, compute a sample for each resulting group, and then use an analytical tool such as Weka, R, BUGS, SPSS, or SAS to build a sample-based classifier for each group to predict the effectiveness of customized advertisements. As another example, we might stratify transaction records by county and by whether or not the transactions are fraudulent; for each resulting group of fraudulent transaction records, we examine the top-k transactions with respect to the total transaction amount, to look for informative patterns. Similar queries arise in root-cause analysis for problem diagnosis.

The difficulty with GSVA queries is that, under an ad hoc

*Work done while authors were at IBM Almaden Research Center

group definition, the records for each group will in general be initially distributed across the different processing nodes. Thus, to process the query in Hadoop, the data is first shuffled across the network so that all records from a given group are at the same node, and then the set-valued query is executed on the group to compute the desired synopsis. This massive data-shuffling phase is extremely time consuming and typically quite wasteful, because most of the shuffled records will not contribute to a small sample or small top-k synopsis and hence be discarded. It may be possible for each node to roughly estimate the set of records that will need to be shuffled and then buffer these records locally as they are scanned from disk, but because the nodes do not communicate with each other and need to be conservative in their estimates to ensure that no result records are missed, the buffered record sets will inevitably be too large, thereby wasting local memory and still using up too much network bandwidth. Besides impeding other users' jobs that are trying to run on the cluster, this waste of resources will translate into a waste of money in a pay-as-you-go setting. The question thus arises as to whether there are more efficient ways to process GSVA queries. We restrict attention to algorithms that require only a single pass over the data; multi-pass algorithms are usually too expensive.

One possibility is to try and execute the GSVA query over a uniform random sample of the entire data set. The difficulty is that smaller groups may not contribute any records to the sample, and will therefore be erroneously omitted from the query result. Similarly, a group may be represented, but with so few records as to make statistical inferences about the group unreliable. That is, naive sampling is inappropriate for such "needle-in-a-haystack" situations. For instance, the set of fraudulent transactions is usually much smaller than the set of legitimate transactions and hence can easily be missed when sampling.

Another possibility is to try and use indexes to filter out irrelevant records prior to shuffling; see, e.g., [2] and references therein. Such indexes are not standard in Hadoop, and can require non-negligible storage and maintenance effort; moreover, appropriate indexes may not be available for a given ad hoc query.

A third approach has been proposed in the context of distributed random sampling by Tirthapura and Woodruff [3], building on initial results in [4]; we denote the resulting algorithm as TW in the rest of the paper. Given a set of k nodes corresponding to k streams, TW maintains at all times a bottom-k set of the union of the streams. A coordinator maintains this set at a special coordination site by exchanging synchronous messages with the nodes. The authors in [3] show that, with high probability, the number of messages exchanged using their protocol is within a small constant factor of the minimum number possible. Unfortunately, this technique is not practical in our setting. TW requires synchronous and ad hoc message exchanges, which are incompatible with MapReduce. Moreover, when there are many groups, the communication overhead can become prohibitive because a sequence of coordination messages must be exchanged for each group.

In this paper, we investigate novel approaches for handling some key GSVA queries that slightly relax the shared-nothing assumption and allow tasks to occasionally access a small, common data structure. We allow each mapper to maintain some state information and use the "Adaptive MapReduce" framework developed in [5] to coordinate the mappers; this framework includes a "distributed metadata store" (DMDS) that serves as the shared data structure. The resulting algorithm, called *GARAM (Groupwise Analytics Running on Adaptive MapReduce)*, combines the two communication patterns embodied in "Vanilla" (i.e., standard) MapReduce and TW; in the former, all of the data is shuffled at the end of the job to produce the stratified synopses, whereas in the latter ad hoc synchronous coordination messages are continually exchanged to maintain a synopsis at all times. GARAM sends a small set of ad hoc asynchronous coordinating messages to minimize the amount of data exchanged during a final shuffling phase. Figure I shows some typical experimental results (see Section VIII for details), which indicate that, with respect to network resource consumption, GARAM outperforms both TW and Vanilla MapReduce by orders of magnitude. The savings in end-to-end response time is also dramatic: GARAM is up to 5x faster than Vanilla MapReduce. We emphasize that TW relies on ad hoc synchronous communication, which is prohibitively expensive in practice, whereas GARAM uses asynchronous communication between mappers, which is vastly more efficient. Note that GARAM requires fewer messages than the near-optimal TW algorithm. This is because TW maintains a synopsis at all times, whereas GARAM produces the final result only when the query completes.

		GARAM	TW	Vanilla
Coord.	size (GB)	0.3	77	N/A
	#msgs (M)	10 (async)	600 (sync)	
Shuffle	size (GB)	36	N/A	630
	#msgs (M)	300		6000

Fig. 1. Typical network utilizations.

Roughly speaking, the idea is to maintain, at each node and for each group, a local version of the desired synopsis. Using the DMDS, histogram summaries of these local versions are periodically combined to estimate a sequence of global *threshold* values that can be used to prune records at each node, thereby avoiding the need to shuffle these records at the end of the job. An important advantage of our approach is that, by maintaining the thresholds, the stratified synopsis can be incrementally updated efficiently as new data arrive; other approaches are hard pressed to provide this functionality. Moreover, the use of Adaptive MapReduce allows for suspending and then resuming a GSVA query, which can be useful for managing response times when multiple users are sharing a MapReduce cluster; standard Hadoop does not support such early-out functionality.

GARAM is useful not only for exploratory analysis, where the groups are specified according to the needs of the analyst, but also for decision support and business intelligence queries [6], [7], [8]. In these latter scenarios, the groups and their corresponding synopsis sizes are usually chosen based on past query workloads. Our work can complement these applications whenever they require the use of MapReduce clusters and approximate query results with good approximation guarantees.

In addition to our results on GSVA queries, we show how GARAM can be combined with approximate thresholding to handle *top-r stratified sampling queries*. Given integers $k, r >$

0, such queries return a sample of size k not for each group, as with a standard stratified sampling query, but only for the r largest groups. In our prior micro-marketing scenario, for example, resource or time limitations might dictate that we focus on the r largest (age, zip code) customer groups.

Although we focus primarily on MapReduce environments because of their ubiquity and broad commercial adoption, our techniques can potentially improve performance in other shared-nothing environments with disk-resident datasets that significantly exceed aggregate cluster memory. These include systems such as Tez [9], Hyracks [10], and Stratosphere [11]. We note that GARAM can also be implemented in systems that are designed primarily for in-memory processing, such as Spark [12]. Our methods, however, are primarily useful when the data will not fit in memory, a situation that, by design, occurs less often in systems like Spark.

II. Adaptive MapReduce

As mentioned above, our discussion centers on groupwise analytics queries in a MapReduce shared-nothing cluster environments. MapReduce is a parallel computation framework that was originally developed at Google [13] and implemented later as part of the Apache Hadoop open-source project [14]. The framework was designed to scan and aggregate large data sets in a robust, flexible, and scalable manner. The framework processes *jobs*, which consist of *map* and *reduce* stages. In the map stage, a set of *mapper tasks* scans the input data set, transforms each input record using a user-defined map function, and extracts a grouping key. In the reduce stage, the map outputs are *shuffled* across a network and grouped according to the grouping key, such that all records in a given group are sent to the same node; at this node, a *reducer task* aggregates the group using a user-defined reduce function, and writes out the result. Mapper tasks run independently and in parallel, as do reducer tasks. In Hadoop terminology, each processing node is divided into a fixed number of *slots*, which corresponds to the maximum number of concurrently running tasks; at most one task can run in a slot at any time. By design, the MapReduce framework can make progress on a job as slots become available, can balance the workload across heterogeneous processing nodes, and can tolerate failures.

The MapReduce programming paradigm significantly simplifies the specification of massive-scale analytics but, as discussed previously, can result in suboptimal performance. The *Adaptive MapReduce* environment introduced by Vernica et al. [5] improves performance and simplifies job tuning over MapReduce by modifying standard MapReduce to allow some limited dependence between mapper tasks. The Adaptive MapReduce framework is carefully designed to preserve the fault-tolerance, scalability, and programming API of MapReduce. Mappers exchange information through an asynchronous communication channel implemented with a *distributed metadata store (DMDS)*, so they are aware of the global state of the job and can collaboratively make optimization decisions.

To speed up GSVA queries, GARAM exploits two techniques from Adaptive MapReduce. *Adaptive Mappers* dynamically take multiple data partitions ("splits" in Hadoop terminology) and make a decision after every split to either checkpoint or take another split and "stitch" it to the already processed one(s), thereby minimizing task-startup overhead, and improving both data locality and load balancing. *Adaptive Combiners* improve local aggregation by maintaining a cache of partial aggregates for the frequent keys. These techniques are injected into the map tasks and they all rely on DMDS for global communication. GARAM also makes use of DMDS directly, and follows design principles of Adaptive MapReduce techniques to ensure its scalability and fault tolerance. See [5] for further details on Adaptive MapReduce.

III. GSVA Algorithms: Overview

In the following sections we develop a variety of algorithms for GSVA queries in MapReduce and extended MapReduce environments and show how buffering and coordination ideas can lead to performance improvements. We focus on stratified bottom-k, top-k, and sampling queries as primary representatives of the class of challenging GSVA queries. An example of another type of GSVA query that can be handled by methods similar to the ones discussed here returns for each group all records within $p\%$ of the minimum weight in the group, for some $p > 0$.

To handle the three types of queries considered, it suffices to restrict attention to bottom-k queries alone. Bottom-k queries return the records with the k lowest weights, so that top-k queries can be handled by multiplying the weights by -1 and then applying a bottom-k algorithm. Moreover, if the weights are generated as random numbers uniformly distributed between 0 and 1, then the bottom-k set for a group corresponds to a size-k uniform random sample of the group. This method is sometimes called "bottom-k sampling" and is used in distributed stream processing [3], [4], as well as in many other settings; see for example [15], [16], [17]. For simplicity, we assume throughout that all of the weights in a dataset are distinct; this assertion holds for stratified bottom-k sampling, and the extension to the general case involves some tedious details, but is straightforward. Also, to avoid trivialities, we assume that every group has at least k records overall.

In practice, pseudorandom number generators (PRNGs) are used to produce the weights needed for stratified-sampling queries. PRNGs recursively and deterministically produce a stream of numbers between 0 and 1 that "appear" random for all practical purposes. High quality PRNGs—having good "structural" properties and passing a wide range of statistical tests for randomness—are readily available; one example is given by the WELL family of generators [18]. Extra care has to be taken in distributed environments, where multiple substreams of independent numbers are required. In our work, we use "jump ahead" capabilities of the WELL generators [19] to generate disjoint (and hence effectively independent) substreams from the WELL generator, one substream per data split. It is also possible to use hash functions to compute the sampling weights.

The algorithms discussed in this paper can easily be extended to handle the case where the synopsis size k varies from group to group. For example, the techniques we describe in this paper can be used in conjunction with STRAT [8] and BlinkDB [6]; these systems use stratified synopses for Business Intelligence workloads, where the strata and their sizes are chosen to optimize query accuracy. It is often

Algorithm 1 (Bottom-k Computation)

```
D: Input dataset of (record, weight) pairs
L: list of k pairs with smallest weights seen so far
L.maxWt(): returns the largest weight in L
L.popMax(): removes element of L with largest weight

for each pair (r, w) ∈ D do
    if |L| < k then
        L.insert(r, w)
    else if w < L.maxWt() then
        L.insert(r, w)
        L.popMax()
    end if
end for
Return L as the bottom-k set
```

desirable, however, to keep the synopsis sizes equal. For example, if the group sizes vary widely, stratified sampling with samples of equal size ensures that sample-based estimates of groupwise characteristics can be made with equal accuracy across groups, thereby allowing statistically meaningful inter-group comparisons.

IV. Vanilla MapReduce

This is the "Vanilla" algorithm mentioned previously, as implemented in MapReduce. Each mapper emits $\langle$*key*, *value*$\rangle$ pairs, where *key* is the grouping key and *value* is the input record. The pairs are shuffled over the network, with the reducers collecting all pairs with the same *key* to form a group. Then the bottom-k set is computed for each group using Algorithm 1 and output to disk. This algorithm is the building block for all of our methods, so we briefly discuss its computational complexity. Clearly, the bottom-k set can be computed via a single scan of the data using $O(k)$ memory; the sorted list L can be implemented using, e.g., a priority queue. If the pairs are scanned in random order, then CPU cost for the reducer (over and above any CPU cost incurred by scanning the records) is given by the following result, which generalizes Theorem 1 in [15]; see the Appendix for a proof.

Theorem 1: For a group $G = \{(r_1, w_1), \ldots, (r_n, w_n)\}$ *with* $w_1 < \cdots < w_n$*, suppose that Algorithm 1 processes the pairs in the order* $(r_{\Pi(1)}, w_{\Pi(1)}), \ldots, (r_{\Pi(n)}, w_{\Pi(n)})$*, where* Π *is a permutation of* $\{1, 2, \ldots, n\}$ *that is selected randomly and uniformly from the set of all such permutations. Then the expected CPU cost to construct a bottom-*k *set is* $O(n + k \log k \log n)$*. The minimum cost is* $O(n + k \log k)$ *and the maximum cost is* $O(n \log k)$*.*

In the case of stratified sampling, running the bottom-k algorithm on a group is essentially equivalent to running a "reservoir sampling" algorithm, and optimizations for reservoir sampling can potentially be applied; see [20], for example. The Vanilla algorithm is typically quite slow because it requires shuffling of the entire data set.

V. Other MapReduce Approaches

In this section we discuss methods that improve upon Vanilla while allowing implementation in standard MapReduce environments.

A. Buffered MapReduce

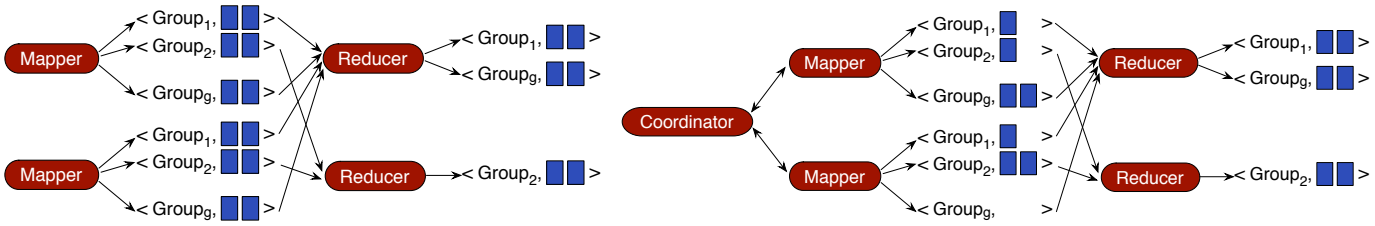
The *Buffered* MapReduce method for computing GSVA queries can reduce the shuffle volume dramatically by maintaining some state at every mapper, as shown in Figure 2(a). In the map phase, each mapper runs the bottom-k algorithm (Algorithm 1) and emits a local sample of k records per group (or fewer if a group contains less than k records locally). Then, for each group, the reducers collect all samples for the group and merge them into a global sample by further selecting the k records having minimum weights (again using Algorithm 1). Thus instead of shuffling all of the records, only $O(gkm)$ records are shuffled, where g is the number of groups and m is the number of mappers. Basically, the local nodes perform some initial filtering, removing records that cannot possibly belong to the final bottom-k synopsis. (Clearly, if a record does not belong to a local bottom-k synopsis, then it cannot belong to a global bottom-k synopsis.) The local synopses can be viewed as partial results of the kind usually associated with the Hadoop "combiner" operator. When the number of records that a mapper sees per group is larger than k, then Buffered can substantially reduce the volume of shuffled data. As the number of records per group approaches *k*, Buffered degrades to Vanilla.

As mentioned previously, Algorithm 1 essentially reduces to a reservoir sampling algorithm when the GSVA query is a stratified sampling query. Although optimizations as in [20] can be exploited by the reducers, they cannot be exploited by the mappers. Indeed, the optimizations allow skipping of records (and thus less I/O), but such skipping is impossible on the mapper side: because the strata are dynamically defined, each record has to be examined in order to be assigned to a stratum prior to any sampling.

Another important implementation issue is buffer management, which can be challenging when there is a "long tail" comprising a huge number of small groups. Indeed, the amount of memory consumed at each mapper is $O(gk)$, and the local buffer for a mapper can fill up. In our initial implementation of Buffered and related algorithms, the mapper is allocated a fixed chunk of memory. This memory is actually split between two buffers, one for data (full records) and another for "metadata" (the key and weight for each record). In experiments 1GB was used for each buffer. Whenever the buffer fills up, large groups of sample records are flushed to the output, to be sent to the reducers. (Flushing in large chunks maximizes efficiency.) The records with smallest weights are flushed first, because they have the best chance of making it into the final output. The metadata (i.e., the weights of flushed records) are retained, subject to memory constraints, so they can be used to filter future records as they are scanned. Whenever the metadata buffer fills up, a group is selected at random and its metadata is flushed. Finally, to maximize the benefit of buffering, our implementation uses the Adaptive Mappers mentioned in Section II, generating one map task per slot as opposed to the default of one task per data block.

B. Approximate Thresholding

Under some additional assumptions, the buffered approach can be improved further while still remaining within the standard MapReduce framework. In particular, suppose that (i)



(a) MapReduce

(b) MapReduce+Coordination

Fig. 2. GSVA query processing with MapReduce.

it is acceptable if, with some small probability, the synopsis size for a group is somewhat less than k, (ii) a good estimate n_i of the size of group G_i is available for each i, and (iii) for all i, the record weights for G_i can be accurately modeled as independent and identically distributed (i.i.d.) samples from some continuous cumulative distribution function F_i. Assumption (iii) holds in the case of stratified sampling, where F is the uniform distribution: $F(x) = x$ for $x \in [0, 1]$. The idea is to estimate for each G_i the *global threshold* $w_{i,(k)}$, which is defined as the kth smallest weight for all records in the group. Then Buffered is run as before, but now each mapper maintains the set of records whose weights do not exceed the threshold.

Observe that if $w_{i,(k)}$ is known exactly for G_i, then the total number of buffered G_i records over all of the mappers is k, and each mapper buffers only a small fraction of these k records. Thus the data shuffling effort can be significantly reduced from $O(gkm)$ to $O(gk)$. In practice, however, $w_{i,(k)}$ is usually unknown and must be estimated. If the estimate $\hat{w}_{i,(k)}$ is too large, then the total number of buffered records over all of the mappers will exceed k, but the excess "false positive" records will be filtered out by the reducers, and exactly k records will be produced. That is, performance will be suboptimal but exact results will be returned. If $\hat{w}_{i,(k)}$ is too small, then records will be erroneously filtered out by the mappers, so that the returned results will be an approximation of the exact solution, in that one or more of the stratified bottom-k synopses will contain fewer than k records. Thus our estimate should actually be an overestimate such that the probability of false negatives is small. Under Assumptions (ii) and (iii) above, we can proceed by observing that, for each group G_i, the threshold $w_{i,(k)}$ is the kth order statistic of a sample of size n_i from distribution F_i. By standard results for order statistics [21, p. 10], we have

$$P(w_{i,(k)} \le x) = \text{Beta}\big(F_i(x); k, n_i - k + 1\big) \stackrel{\text{def}}{=} H_i(x), \quad (1)$$

where $\text{Beta}(x; a, b) = \int_0^x t^{a-1}(1-t)^{b-1}\,dt \;/\; \int_0^1 t^{a-1}(1-t)^{b-1}\,dt$ denotes the standard beta distribution with parameters a and b. For some small value $\epsilon > 0$, let $q_{i,\epsilon}$ be the unique solution of $H_i(q) = 1 - \epsilon$, i.e., the $(1-\epsilon)$-quantile of H_i, and set $\hat{w}_{i,(k)} = q_{i,\epsilon}$. Then the probability is at most ϵ that $\hat{w}_{i,(k)}$ will underestimate the true threshold, so that one or more G_i records will be incorrectly filtered out by the mapper. With this choice of $\hat{w}_{i,(k)}$, it can be shown that the expected number of false positives is given by $\phi(q_{i,\epsilon})$, where

$$\phi(q) = \frac{n_i - k}{\text{IB}(1; k, n_i - k + 1)} \times \Big[F(q)\text{IB}\big(F(q); k, n_i - k\big) - \text{IB}\big(F(q); k+1, n_i - k\big)\Big]$$

and $\text{IB}(x; a, b) = \int_0^x t^{a-1}(1-t)^{b-1}\,dt$ is the incomplete beta function. Suppose, for example, that F is the uniform distribution, group G_i contains $n_i = 10,000$ elements, and we want a global sample of size $k = 100$. Taking $\epsilon = 0.01$, we find that $q_{i,\epsilon} \approx 0.012$, which yields about 25 false positives, i.e., there is roughly a 25% overhead of wasted shuffling. In the next section we show how, by allowing limited coordination between mappers, we can obtain general algorithms for GSVA queries that yield exact results (no records incorrectly filtered out) while effectively pruning records (i.e., false positives) in the map phase.

VI. Methods Based on Coordination

The methods in this section build on the Buffered approach. We exploit the fact that with lightweight coordination between mappers, the buffers can be significantly pruned while the mappers are running.

A. MHD Algorithm for Stratified Sampling

In the special setting of stratified sampling, a variant of the Buffered approach maintains local bottom-k samples and then executes a subsampling step before the shuffle phase in order to optimize the shuffle. Specifically, for group G_i, mapper M_j computes a local count $n_{i,j}$ of the number of G_i records. These counts are sent to a coordinator, who computes an aggregate count n_i for the group and then generates an exact sample size $K_{i,j}$ for each mapper using a multivariate hypergeometric distribution (MHD). That is, for nonnegative integers $k_{i,1}, k_{i,2}, \ldots, k_{i,m}$ such that $k_{i,1}+k_{i,2}+\cdots+k_{i,m} = k$, we have

$$P(K_{i,1} = k_{i,1}, \ldots, K_{i,m} = k_{i,m}) = \prod_{j=1}^{m} \binom{n_{i,j}}{k_{i,j}} \Big/ \binom{n_i}{k}.$$

Then each mapper M_j randomly selects $K_{i,j}$ records from its bottom-k sample of G_i records, and shuffles these samples to the reducers. This procedure transfers the final filtering step from the reducers to the mappers. Thus there are no false-positive records, and the reducers simply merge the partial G_i samples into a global sample from G_i.

Unfortunately, although this algorithm is optimal with respect to the shuffle, it is not well suited to the MapReduce architecture. The reason is that the optimization step is a synchronization barrier: the process needs to wait until all mappers have completed their data scans before shuffling can commence, and its overall performance is thus limited by the performance of "stragglers." This bottleneck can be severe; mappers can work at different speeds due to their heterogeneous performance characteristics, as well as to data skewness and layout (see Theorem 1), and some of them may even fail. In general, a MapReduce job cannot make any assumptions about its degree of parallelism, i.e., how many of its tasks will run concurrently. Thus effective algorithms must be asynchronous. Another drawback of the MHD method is that, like Buffered, the amount of memory consumed at each mapper is $O(gk)$, which can become large.

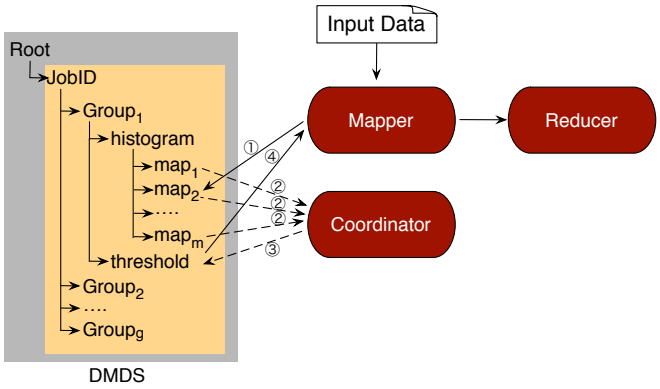


Fig. 3. Asynchronous coordination.

B. GARAM

We now describe the GARAM algorithm, which is applicable to general bottom-k queries, returns an exact result, approximately minimizes the number of shuffled records, and avoids synchronization barriers.

1) Coordinating Threshold Estimates: First recall that the (global) group threshold $w_{i,(k)}$ of group G_i is defined as the kth smallest weight for all records in the group. Mappers and reducers can have different views of the group threshold. Each mapper only sees a subset of records, so it computes a *local threshold*, i.e., the kth smallest weight of the G_i records that it sees. Reducers, on the other hand, see all of the weights, so they arrive at a global threshold.

Example 1: Suppose that the Buffered algorithm is applied with only a single group and with $k = 2$. There are six records, r_1 through r_6, and two mappers, M_1 and M_2, each of which processes three records. Suppose that mapper M_1 processes r_1 through r_3 having weights 0.5, 0.9, and 0.1, and mapper M_2 processes r_4 through r_6 having weights 0.7, 0.6, and 0.2. Each mapper maintains a local sample of size two. As a result, M_1 drops $(r_2, 0.9)$ and emits $\langle key, \{(r_3, 0.1), (r_1, 0.5)\}\rangle$; mapper M_2 drops $(r_4, 0.7)$ and emits $\langle key, \{(r_6, 0.2), (r_5, 0.6)\}\rangle$. The local thresholds for M_1 and M_2 are 0.5 and 0.6. In the reduce phase, the two samples are merged so that the two records with minimum random weights comprise the final output. So the reducer emits $\langle key, \{(r_3, 0.1), (r_6, 0.2)\}\rangle$ and the global threshold is 0.2.

In Example 1, four records (r_1, r_3, r_5 and r_6) are shuffled to the reducer and half of them (r_1 and r_5) are dropped in the reduce phase. That is, as discussed previously, the shuffle cost is $O(gkm)$ as opposed to the ideal cost of $O(gk)$ that corresponds to the case of no false-positive records. It is therefore desirable to remove the factor m, or at least reduce it to a constant close to 1, without affecting correctness.

The key idea in GARAM is to use coordination between mappers to better approximate the global threshold as shown in Figure 2(b). In the figure, the coordinator communicates with all mappers and thus its view of the global threshold is more accurate than those of the mappers. The coordinator periodically communicates to the mappers its view of the global threshold which mappers can use to pre-filter local samples. In this way, we can expect a reduced shuffle volume and consequent performance boost. The details are given below.

For each group, GARAM estimates the global threshold from a union of histograms collected by the mappers and having $b > 1$ bins per histogram. Each histogram is approximately an equi-depth histogram of the values in the buffer, subject to the restriction that each bin boundary corresponds to an actual weight that is present in the buffer. Specifically, a histogram is constructed by sorting the n values in the buffer (which are all distinct by assumption) in ascending order and then computing the first upper bin boundary as the $\lceil n/b \rceil$th smallest weight, the second upper bin boundary as the $2\lceil n/b \rceil$th smallest weight, and so on, until the maximum weight is encountered.[1] Thus the count for each bin but the last is $\lceil n/b \rceil$; the count for the last bin may be smaller.

Because the goal is to compute the bottom-k synopsis exactly, GARAM combines histograms in such a manner so as to guarantee that the threshold is not underestimated. (Recall that an overestimate is acceptable with respect to correctness since the false-positives will be filtered out by the reducers.) Each b-bin histogram provides an accurate picture of the distribution of weights in the buffer while ensuring that the bin counts are exact, as is needed when providing guarantees.

For ease of exposition, we first describe a synchronous version of the threshold-estimation protocol, and then describe GARAM's actual asynchronous implementation. Each mapper maintains a buffer of up to k records per group, similar to the Buffered MapReduce algorithm. Because GARAM uses thresholds, however, the average buffer size will be much smaller than k. For a given group G_i, the synchronous protocol runs the following three-step coordination rounds at fixed time intervals. First, each mapper computes an equi-depth histogram of b bins over the weights that it currently has in its G_i buffer. The mapper sends the histogram to the coordinator encoded as a list of b pairs of the form $\langle t_{i,l}, c_{i,l}\rangle$, meaning that bin l contains $c_{i,l}$ values between $t_{i,l-1}$ and $t_{i,l}$. Second, the coordinator sorts the list of all such pairs in order of ascending $t_{i,l}$ values. It then scans the list in this order and computes the running sum of $c_{i,l}$'s. Whenever the running sum becomes greater or equal to k, the current $t_{i,l}$ becomes the new global threshold estimate $\hat{w}_{i,(k)}$. If the sum of all $c_{i,l}$'s is less than k,

[1] Here $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.

Protocol 1 (GARAM Coordination Protocol for Group G_i)

1: Mapper M_j periodically computes and writes histogram pairs $\langle t_{i,l}, c_{i,l} \rangle$ to location $\texttt{Group}_i\texttt{/histogram/map}_j$ in the DMDS.
2: Periodically, the coordinator reads the most recent histograms for G_i and tries to compute a new global threshold using Algorithm 2.
3: If the coordinator computes a better global threshold for G_i, it writes it to $\texttt{Group}_i\texttt{/threshold}$ in the DMDS.
4: Mappers periodically update their local thresholds with the most recent threshold from DMDS and start using it to filter local samples.

the coordination round for G_i fails: not enough data has been scanned yet, or G_i contains less than k records. Finally, if the coordinator produces a global threshold that is better (i.e., smaller) than the current last one, it notifies all mappers to filter local samples with $\hat{w}_{i,(k)}$. This protocol guarantees that mappers will never receive an underestimated global threshold, so that the synopsis will be exact.

2) Asynchronous Coordination: Although the foregoing coordination protocol guarantees correctness, it relies heavily on the synchronization barriers; as discussed previously, such barriers usually do not work well in MapReduce. GARAM therefore uses a fully asynchronous coordination protocol (Protocol 1) that exploits the shared DMDS data structure from Adaptive MapReduce that was introduced in Section II. All of the mappers and the coordinator can read from and write to the DMDS asynchronously. The DMDS has a hierarchical data model, and stores a small amount of coordination state per group.

Figure 3 shows the groupwise state information maintained in the DMDS, as well as the data flow in the system. Each numbered arrow corresponds to a step in Protocol 1. The mapper maintains local records (with corresponding weights) for each group as it reads and partitions input data. When the mapper has seen a specified number of new records for a group, called an *update batch*, it writes a histogram for the group in the DMDS. Independently, the coordinator periodically reads updated histograms and computes a new global threshold estimate, which it also writes to the DMDS. In this way, each mapper can contribute its information at its own speed, and the coordinator produces a threshold based on the latest available information, which is never required to be completely up to date. Each mapper always has its own view of the global threshold, which is also allowed to be outdated, i.e., out of sync with the DMDS. If coordination and/or communication of thresholds falls behind, the mapper can still make progress filtering records with its most recent version of the threshold. Observe that thresholds decrease monotonically, so that if a mapper is lagging behind, its threshold is too large, which does not impede correctness. Whenever a the mapper receives a new, smaller threshold, it simply prunes its buffer as appropriate.

Note that, as the local threshold for a group decreases, the histogram bin widths also decrease, so that the histogram gives an increasingly accurate picture of the weight distribution below the threshold. Also note that the coordinator does not compute a global histogram of weights for a group: only the frequencies of weights that lie below the current global threshold are tracked, and the bin counts are based on observations taken at different time points. Finally, note that the mappers, which run Steps 1 and 4, and the coordinator, which runs Steps 2 and 3, are completely independent of each other. They communicate through the `histogram` and `threshold` locations in the DMDS; each of these locations has exactly one writer and one reader.

Algorithm 2 (Pseudocode of the Coordinator)

```
1: while !allDone do
2:     for each group G_i to be coordinated do
3:         for each M_j message for G_i in DMDS do
4:             if there is a previous proposal from M_j then
5:                 list[i].removeOld(j)
6:             end if
7:             list[i].addNew(j)
8:         end for
9:         thre ← list[i].findThreshold()
10:        if thre < lastThre[i] then
11:            updateThre(i, thre)
12:        end if
13:    end for
14: end while
```

The pseudocode for the coordinator algorithm is given in Algorithm 2, and its communication with the DMDS is shown by the dashed lines in Figure 3. Periodically, the coordinator receives histogram buckets for G_i from mapper M_j in the form of upper bin boundaries $t_{i,l}$ and corresponding bin counts $c_{i,l}$; the bin boundaries are candidates for the next threshold estimate. The coordinator updates a list of triples $\langle t_{i,l}, c_{i,l}, j \rangle$ by removing any previous buckets from M_j (lines 4-6), and adding the new buckets to the list (line 7). The list is kept sorted in increasing order of bin boundaries to facilitate the subsequent `findThreshold()` operation in line 9. This function scans the list in sorted order, starting from the head, and computes a running sum S_i of sample counts $c_{i,l}$. It stops scanning the list when $S_i \geq k$; all triplets after this point in the list are removed and the $t_{i,l}$ in the last retained triplet is returned as the putative updated threshold. If `findThreshold()` is successful and the resulting threshold is smaller than the last one, the coordinator updates the threshold in the DMDS. However, if S_i never reaches k, the threshold is not updated, and GARAM will try to coordinate G_i later.

It is important to note that the coordinator will post a threshold only if it has guarantees from the mappers that they already have at a total of least k records with weights less than or equal to this threshold. Until the coordinator can prove this, the mappers will use their local thresholds to filter samples. In the unlikely event that the coordinator never produces a threshold for a group, GARAM will degrade to running the Buffered algorithm for this group. In any case, no record that should appear in the final bottom-k synopsis will be filtered by a mapper.

Example 2: For a given group, suppose that $k = 5$ and $b = 2$, and that there are two mappers, M_1 and M_2 with update-batch size equal to 10; i.e., the mappers communicate after scanning every 10 records. Also suppose that the current threshold is equal to 1.0 at both the coordinator and the mappers, and that M_1 sends the message $\langle 0.4, 3 \rangle \langle 0.5, 2 \rangle$, which means it has three records with weights up to 0.4 and two more with weights in $(0.4, 0.5]$. Sometime later, after scanning its most recent batch of 10 records, M_2 sends the message $\langle 0.2, 3 \rangle \langle 0.45, 2 \rangle$. Upon collecting both messages, the coordinator has the list $\langle 0.2, 3, 2 \rangle \rightarrow \langle 0.4, 3, 1 \rangle \rightarrow \langle 0.45, 2, 2 \rangle$

$\rightarrow \langle 0.5, 2, 1\rangle$. *It scans the list from left to right until the cumulative sum S of the second field is at least k. The scanning stops at the second triplet because $S = 3 + 3 = 6 > k = 5$. Then 0.4, the first element in the second triplet, is the proposed new global threshold; since $0.4 < 1.0$, the proposal is accepted, and the coordinator declares 0.4 as the threshold. Once M_1 receives this threshold it can prune its two local records having weights greater than 0.4.*

3) Implementation Details: We implemented the DMDS on top of Apache ZooKeeper, which satisfied our requirements for performance, functionality, fault tolerance, and high availability. Many other tools could also be used to implement the shared data structure. Because the DMDS cannot execute user code, GARAM's coordinator component is run outside the DMDS. To prevent the coordinator from becoming a bottleneck, especially when the number of groups to be coordinated is large, GARAM distributes the coordination process by partitioning the groups and dynamically assigning them to different coordinators. Every map task of a GARAM Hadoop job runs a coordinator thread, as well as a ZooKeeper client thread that handles asynchronous communication with DMDS. In this way GARAM can piggyback on Hadoop and ZooKeeper to manage the coordinator processes and provide fault tolerance for coordination. Specifically, in case of a task failure, Hadoop restarts the task, including the coordinator thread. The new coordinator finds all the state of its predecessor in ZooKeeper and continues to execute the algorithm. This setup also allows threshold updates to be pushed from the DMDS to the mappers (Step 4 of Protocol 1), which improves performance.

As mentioned above, GARAM uses simple equi-depth histograms to estimate thresholds; the benefits of using more sophisticated histograms are far outweighed by the additional costs incurred. To determine the value b for the number of bins in the histograms, we ran some preliminary experiments using $b = 2$, 10, 50, and 250 buckets per mapper. On uniformly-sized groups the choice of b made no difference at all. With heterogeneous group sizes there was a slight impact: performance with $b = 10$ was about 5% faster than with $b = 2$, and performance with $b = 50$ and $b = 250$ was very slightly worse than with $b = 10$, though these latter differences were pretty much at the noise level. Based on these results, we used 10 buckets per mapper throughout.

GARAM uses the same basic buffer management strategy as for the Buffered algorithm (see Section V-A), spilling records to the reducers whenever the record buffer fills up but maintaining the metadata for the ejected groups, which can still be used to prune future records and compute better thresholds. As with Buffered, metadata is spilled, one group at a time, as the metadata buffer fills up. In GARAM, however, the most recent threshold for a group is kept when the group's metadata is spilled; in this case all weights are lost, however, so it is very hard to further improve the threshold for the group. Fortunately, our experiments indicated that spilling of metadata is a very rare event. During the experiments described in Section VIII, there was only one experiment (with a maximum value of k and a maximum number of groups) where Buffered spilled metadata. GARAM uses some additional methods to control the buffer size, and consequently never spilled metadata during the experiments. Specifically, if the buffers are in danger of overflowing, GARAM heuristically increases the communication rate in order to speed-up coordination and prune down the buffers before they overflow. GARAM also slows down the data scanning thread for a mapper to let the coordination and filtering processes catch up.

The update-batch size is determined by the throughput limit of the DMDS, which is the major source of coordination overhead. Suppose this limit is $Comm_{rate}$ messages per second. For a given job that scans records at an aggregate rate of $Scan_{rate}$ records per second, the minimum update-batch size is $U_{min} = \text{num}_{msg} \cdot Scan_{rate}/Comm_{rate}$. The constant num_{msg} is the average number of DMDS messages per update, which is roughly 5 in our setting. For example, we measured $Comm_{rate}$ to be around 90K messages per second. With our 20 machines the combined scan rate is about 2 GB/sec. On our dataset it translates to $Scan_{rate}$ of about 500K records per second, so U_{min}, in our experiments, is just under 30. Thus mappers should not update DMDS more often than once every 30 records. GARAM employs a heuristic that sets the first update-batch size for each group to U_{min} records, and all the subsequent ones to $5U_{min}$ unless the buffer size constraints force a lower batch size as described above.

Note that, depending on the update-batch size and other factors, it is possible for one or more groups to never communicate with the coordinator and hence never filter out any records. In this case the MHD algorithm of Section VI-A can be run on the set of unfiltered groups at the end of the map process; in this setting the number of unfiltered groups is usually very small so that the "straggler effect" is not too severe.

4) Other Features of GARAM: Besides superior performance (as indicated by our experiments), GARAM has a number of additional features that make it more attractive than other MapReduce sampling approaches.

Pause-resume GARAM can be easily controlled to suspend processing if, for example, a higher priority user needs to utilize the MapReduce cluster. Adaptive Mappers can be short circuited not to process any more splits, so Hadoop will believe that the job is complete and send all mapper output to reducers. When a job has been stopped in this manner, GARAM can still maintain its state in the DMDS. This includes a list of all splits that have been processed by the job. Given this state, the Adaptive Mappers can restart processing where they left off, using the most recent threshold per group. The code for a reducer needs only a trivial change to let the reducer read back its output file from the stopped job and append these records and their weights to the reduce function input. The reduce function itself remains the same.

Incremental Maintenance of Synopses The foregoing pause-resume mechanism can be used to incrementally maintain a stratified synopsis as new partitions are appended to the dataset. An *incremental* GSVA job J_i needs to know the ID of the previous GSVA job J_p that ran on the same dataset. Job J_i locates the data structures that J_p left in the DMDS, so that it can start with the latest available set of thresholds. The reducers of J_i also read the corresponding partition of the synopsis that J_p produced.

Stratified online aggregation The pause-resume capability can also be used to support a stratified form of online aggregation [17], [22]. Indeed, the Adaptive Mappers process blocks of records in random order. Thus, if GARAM is used

for stratified sampling and processing is stopped at some point, then there are no partially scanned blocks, and the GARAM sample is a true record-level stratified random sample of the set of blocks scanned so far; this set is representative of the entire data set because blocks are scanned in random order. It is therefore possible in principle to abort processing as soon as the samples are "good enough," either in themselves or with respect to the accuracy of an aggregation query or other analysis that is being performed over the samples. We intend to investigate this functionality in future work.

Overlapping Groups GARAM supports overlapping groups, i.e. extracting multiple keys from a record and sending the record to multiple buffers without duplicating it in memory. This feature is useful for running multiple GSVA queries concurrently while scanning the dataset only once. This feature also handles semi-clustering algorithms that potentially assign a record to more than one group.

Fault tolerance GARAM relies on MapReduce, a fault-tolerant DMDS (in our case, backed by ZooKeeper) and, in the case of stratified sampling, a PRNG to gracefully handle task and node failures. Recall that the PRNG generates disjoint substreams of independent pseudorandom numbers, where each sub-stream corresponds to a data split. For each split, the PRNG uses a seed that is tied to the split's id. Thus rerunning a task will always produce the same weight for the same record, which lets the Adaptive Mappers ensure that a task failure does not affect the final output.

VII. Top-r Stratified Sampling

In this section we discuss queries that return a uniform random sample of size k for each of the r largest groups in the set $\mathcal{G} = \{ G_1, \ldots, G_g \}$ of all groups. A straightforward implementation of top-k stratified sampling uses GARAM to compute a size-k sample for each of the g groups, computing the size n_i of each group G_i as a by-product. Then only the samples corresponding to the r largest groups are retained. This solution is wasteful, however, because every sample not belonging to the r largest groups is shuffled across the network. This wasted shuffling effort is especially severe when, as is often the case, the data has a "long tail" so that $r \ll g$ and each group has at least k elements.

Our top-r stratified sampling algorithm, called GARAM-TrSS, combines GARAM with the approximate thresholding technique in Section V-B. The key idea is to identify the top-r groups as early as possible and prune the remaining groups aggressively in the map phase. Specifically, GARAM-TrSS maintains a running estimate $\hat{\mathcal{T}}$ of the true set $\mathcal{T}$ of top-r groups; i.e. $\hat{\mathcal{T}}$ is based on the data seen so far. Our implementation uses the efficient distributed top-r monitoring algorithm proposed by Babcock and Olston [23]. The algorithm also maintains for each group G_i the number n_i^* of records scanned so far. Thus, if we order these counts as $n^*_{[1]} > n^*_{[2]} > \cdots > n^*_{[g]}$, then $\hat{\mathcal{T}} = \{ G_{[1]}, \ldots, G_{[r]} \}$. At a given time during the map phase, let q^*_ϵ be the unique solution of $\text{Beta}(q; k, n^*_{[r]} - k + 1) = 1 - \epsilon$, where ϵ is a small user-specified error tolerance. In GARAM-TrSS, each group initially uses standard GARAM to maintain a size-k sample, and then follows Protocol 2. In the protocol, the current threshold for a mapper is the smallest threshold seen so far (and thus only changes when a newly proposed threshold is smaller than the current threshold).

Protocol 2 (Top-r Stratified Sampling for Group G_i)

1: If $G_i \in \hat{\mathcal{T}}$, each mapper maintains a G_i sample using the threshold sequence generated by standard GARAM
2: If $G_i \in \mathcal{G} \setminus \hat{\mathcal{T}}$, each mapper maintains a G_i sample using a sequence of periodically updated q^*_ϵ values together with the standard GARAM threshold sequence

Observe that the threshold q^*_ϵ is based on the sizes of all the groups, and usually is relatively tight for any given non-top-r group. To see that this protocol yields approximately the desired sample, first note that, if a top-r group $G_i \in \mathcal{T}$ remains in $\hat{\mathcal{T}}$ throughout the map stage, then use of standard GARAM ensures a size-k sample. A non-top-r group $G_i \in \mathcal{G} \setminus \mathcal{T}$ will usually be effectively pruned by use of the q^*_ϵ threshold (and any surviving records will be eliminated at the reducers). The potentially problematic case concerns a top-r group $G_i \in \mathcal{T}$ that enters $\mathcal{G} \setminus \hat{\mathcal{T}}$ one or more times during processing and hence for certain time periods may be pruned using q^*_ϵ thresholds, and not just the standard GARAM thresholds (which will typically be larger). The bottom-k sampling mechanism will always produce a statistically correct sample, but there is a risk that the final sample size will be less than k. The following result shows, however, that with high probability q^*_ϵ will exceed the global threshold $w_{i,(k)}$ at all times, so that use of q^*_ϵ will not lead to over-pruning.

Consider a group $G_i \in \mathcal{T}$ and denote by A_i the event that $G_i \in \mathcal{G} \setminus \hat{\mathcal{T}}$ at least once during the map phase. When A_i occurs, denote by $\{ q^*_{\epsilon,l} \}_{l \geq 1}$ the sequence of q-thresholds used for G_i, with corresponding $n^*_{[r]}$ values $\{ n^*_{[r],l} \}_{l \geq 1}$.

Theorem 2: $P(w_{i,(k)} > \min_l q^*_{\epsilon,l} \mid A_i) < \epsilon$.

Proof: Denote by $n_{[r]}$ the true size of the smallest group in $\mathcal{T}$. Observe that $n_i \geq n_{[r]}$ since $G_i \in \mathcal{T}$ and that, for all sample paths in A_i, the sequence $\{ n^*_{[r],l} \}_{l \geq 1}$ is non-decreasing with $\max_l n^*_{[r],l} \leq n_{[r]}$. The function $\text{Beta}(x; k, n-k+1)$ is non-decreasing in n for any fixed $x \in (0,1)$ and $k \geq 1$, where $n \geq k$; see, e.g., [21, p. 90]. It follows that $\min_l q^*_{\epsilon,l} \geq q_\epsilon$ on A_i, where q_ϵ is the unique solution to $\text{Beta}(q; k, n_{[r]} - k + 1) = 1 - \epsilon$. Using (1), we have

$$\begin{aligned} P(w_{i,(k)} > \min_l q^*_{\epsilon,l} \mid A_i) &\leq P(w_{i,(k)} > q_\epsilon \mid A_i) \\ &= 1 - \text{Beta}(q_\epsilon; k, n_i - k + 1) \\ &\leq 1 - \text{Beta}(q_\epsilon; k, n_{[r]} - k + 1) = \epsilon, \end{aligned}$$

where the second inequality again follows from [21, p. 90]. ■

VIII. Experiments

For our experimental evaluation we focused on the application of GARAM to stratified sampling, that is, the weights are uniform random numbers in $[0, 1]$. We compared the various methods and studied the impact on relative performance of two key factors: the number of groups g and the sample size k per group.

Specifically, we considered stratified sampling of the largest table of the publicly available SkyServer dataset[2]. The

[2] http://cas.sdss.org

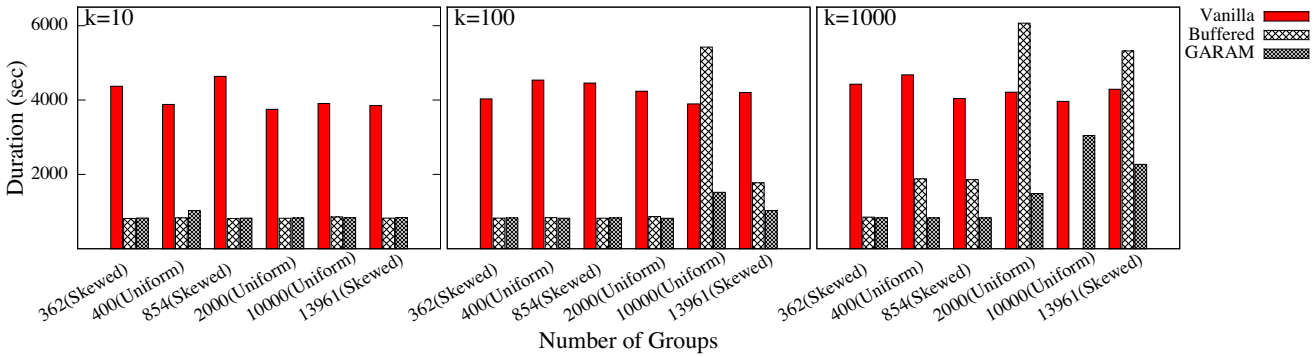


Fig. 4. Running time of GARAM, Vanilla, and Buffered algorithms.

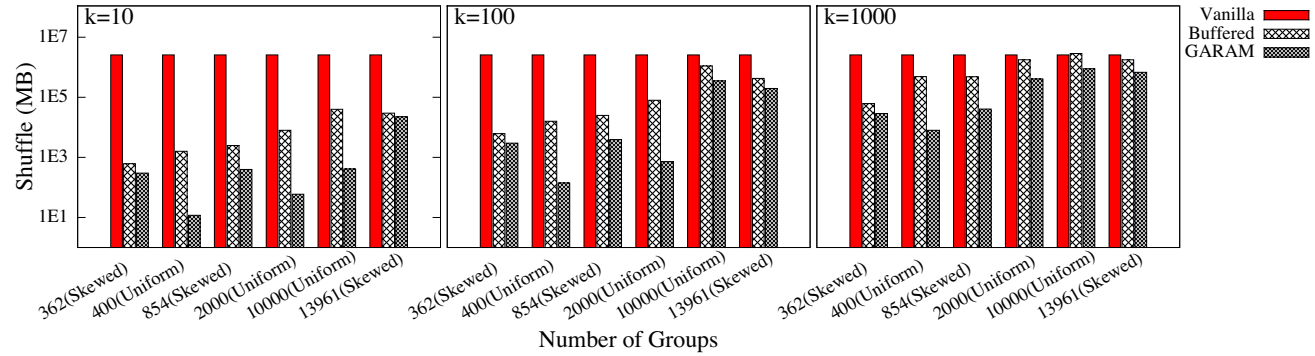


Fig. 5. Shuffle size (log scale) of GARAM, Vanilla, and Buffered algorithms.

table has 245 columns and 586 million records. We stored this data as a 2.45 TB text file in the Hadoop distributed file system (HDFS) running on our cluster.

Many columns of this table have non-uniform value distributions, which facilitated our evaluation. In particular, one set of experiments constructed a stratum for each distinct value of one column. We used three columns with 362, 854, and 13961 distinct values respectively. The distribution of stratum sizes was significantly skewed in these three cases. We also ran experiments where 400, 2000, or 10000 uniform strata were constructed by hash-partitioning the table.

The cluster has 12 nodes, each with one 12-core Intel Xeon CPU E5-2430 64-bit 2.20GHz processor, 96GB RAM, and 12 SATA disks. The nodes are connected by 10Gbs Ethernet. We used Hadoop version 1.1.2. Two nodes were reserved to run the Hadoop JobTracker/the NameNode for HDFS, and ZooKeeper server for DMDS respectively. The other 10 nodes ran the worker daemons. Each worker was configured with 8 map slots and 4 reduce slots. Thus, each worker node ran up to 12 concurrent tasks, and so we set each task's memory allocation to 7 GB.

We compared our GARAM implementation with the Vanilla and Buffered MapReduce approaches described in Secs. IV and V-A. We also implemented the TW algorithm [3] described in Sec. I, but it turned out not to be competitive. TW requires a relatively large number of very expensive synchronous communications. In our tests on a scaled-down dataset, this implementation was always significantly slower than the other approaches, and on the full dataset we had to cancel the job after it ran for an hour and processed only about 2% of the data.

A. *Performance of Vanilla and Buffered*

Recall that the Vanilla algorithm uses mappers to extract the grouping (i.e. strata) keys and shuffle all records of a given group to a single reducer. The reducer tasks then perform the sampling. This approach has very stable, albeit very poor performance, as it requires shuffling (i.e. repartitioning) of the entire data set to obtain the samples. Figure 4 shows that, on a 10-node cluster, Vanilla always completed in around 70 minutes, irrespective of the desired sample size k and number of groups g.

As discussed previously, the Buffered algorithm maintains a list of up to k samples per group locally in mapper memory. If the buffers are large enough to hold all of the groups, then every mapper outputs up to k records per group, along with their weights, and the reducers output the exact overall sample of k records per group. As long as every map task scans more than gk records, Buffered shuffles less data than Vanilla. Recall that Buffered uses Adaptive Mappers (with one task per slot versus one task per HDFS block); for example, in Figure 4, Buffered always ran 80 map tasks. We could have achieved the same number of tasks in standard Hadoop by setting the appropriate data partition (split) size, but the adaptive approach yields much better data locality and avoids load balancing problems associated with using very large splits.

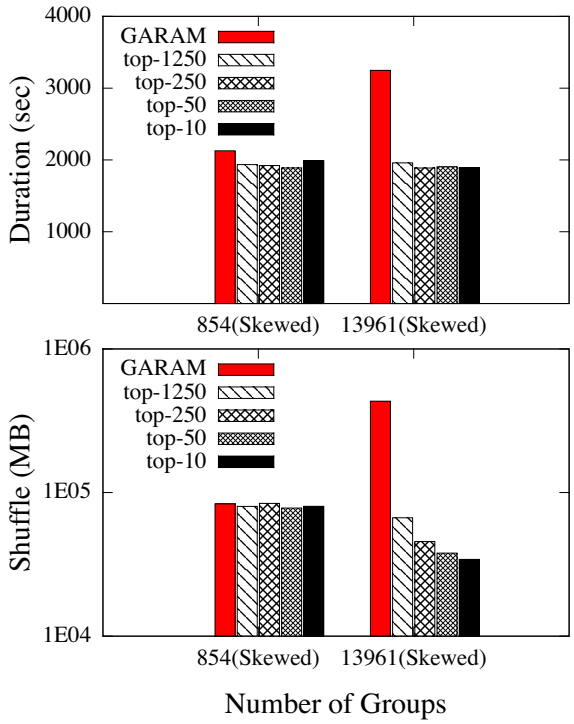


Fig. 6. Running time and Shuffle size (log scale) of GARAM and top-r stratified sampling

Buffered performs best when the entire local sample (of roughly gk records) fits in the buffer. In our experiments, the size of each buffer was 5GB. Buffering is done very carefully to minimize Java object overhead and avoid expensive creation and garbage collection of many small objects. Nevertheless, as the product gk scaled up past the size of the buffer—in our case the 5GB buffer can fit around 1 million records—the performance of Buffered degraded very rapidly. It actually performed worse than Vanilla because it used much more memory, so that less file-system cache was available for the external sorting used in MapReduce. Even when the samples fit into the buffers, data shuffling costs were often problematic. For example, as shown in Figure 5, when $k = 1000$ and $g = 400$, there is enough local memory to buffer the entire local sample of 400 K records but, once each of the 80 mappers produces such a sample, the reducers need to shuffle over 160 GB of map outputs, which is wasteful and degrades performance.

B. Performance of GARAM

GARAM uses the same buffering and local aggregation code as Buffered, but its performance degrades much less. By using the coordination protocol, GARAM can prune local samples much more aggressively and hence buffer many fewer records. Indeed, GARAM only needs to buffer approximately gk records across all workers, whereas Buffered needs to buffer approximately gk records per worker. This difference is apparent from Figure 5: GARAM shuffles about 100 times less data than Buffered for the uniform groups when $k = 10$ or 100 and $g = 400$ or 2000. This improvement factor corresponds to the 80 mappers used in the experiments.

GARAM is designed to scale well with the dataset and cluster size, however its current implementation is limited in the number of groups it can coordinate, due to limitations of the current DMDS, as discussed in Sec. VI-B3. This explains its performance degradation as the number of groups increases, especially for skewed data distributions, where buffers for some groups can fill up very quickly. However, analysts rarely look at a large number of strata. They are more likely to focus on a smaller set of the largest groups, which can be done by the top-r stratified sampling algorithm that we evaluate next.

C. Performance of Top-r Stratified Sampling

We focus on the cases where GARAM performance degraded when it produced large samples (k=1000) for many non-uniform groups. Figure 6 shows performance of top-10 stratified sampling for both uniform and skewed strata, compared against GARAM. Top-r stratified sampling significantly outperforms GARAM in the case of 13961 strata, i.e., top-r stratified sampling shuffles 20x less data, and runs 2x faster than GARAM, because it is able to focus on the largest groups and coordinate their thresholds quickly enough to keep up with the scan rate. In the less skewed case (e.g., 854 strata), top-r stratified sampling does not improve much over GARAM, because it only shuffles slightly less data than GARAM. It is worth noting, however, that the duration of top-r stratified sampling is still less than that of GARAM. Thus, although the top-r stratified sampling algorithm needs to maintain the running top strata and conduct extra thresholding, it introduces little overhead compared to GARAM.

IX. Conclusions

We have shown how extending a MapReduce shared-nothing framework to allow limited coordination between map tasks can dramatically improve performance for an important class of queries over massive disk-resident data sets—namely, groupwise set-valued analytics (GSVA) queries—while preserving the key advantages of the MapReduce framework. Specifically, we have introduced GARAM, a novel technique for processing GSVA queries in MapReduce environments. We also presented its specialization for stratified sampling of the largest groups. In both cases, the sparing use of a small shared data structure, implemented using the Adaptive MapReduce infrastructure, allows mappers to jointly estimate adaptive thresholds for more effective pruning of records, thereby reducing buffer space requirements and avoiding expensive and wasteful data shuffling. Extensive experiments using real datasets showed speedups of up to 5x, which indicates that our new techniques can facilitate exploratory analysis of massive data. Our techniques can potentially be applied in other shared-nothing systems for disk-resident data; such extensions are a topic for future research.

Acknowledgments

We would like to thank CWI Database group—especially Romulo Gonçalves, Martin Kersten, Niels Nes, and Arjen de Rijke—for providing us with the cleaned version of the SkyServer dataset that we used in our experiments. We would also like to thank David Woodruff for fruitful discussions on the GARAM and TW algorithms.

APPENDIX: PROOF OF THEOREM 1

Observe that the first k pairs are inserted into the priority queue (line 8) and each such insertion has a cost of $O(\log k)$, for an overall cost of $O(k \log k)$. Each subsequent pair incurs an $O(\log k)$ cost if it is inserted (line 10), or an $O(1)$ cost otherwise. The ith new pair ($i > k$) is inserted only if $\Pi(i) < M_i$, where $M_i = \max\{\, j : (r_j, w_j) \in L_i \,\}$ and L_i denotes the list L just before processing the ith pair. Straightforward combinatorial arguments show that

$$\begin{aligned} &P\big(\Pi(i) < M_i, M_i = m\big) \\ &= (m-1)\binom{m-2}{k-1}\binom{n-m}{i-k-1}\frac{(i-1)!(n-i)!}{n!} \\ &= \frac{k}{i}\binom{n}{i}^{-1}\binom{m-1}{k}\binom{n-m}{i-k-1}. \end{aligned}$$

for $m \geq k$ and $P\big(\Pi(i) < M_i, M_i = m\big) \leq P(M_i = m) = 0$ for $m < k$. Following the usual convention that $\binom{k}{l} = 0$ whenever $k < l$, we have

$$\begin{aligned} P\big(\Pi(i) < M_i\big) &= \sum_{m=k}^{n} P\big(\Pi(i) < M_i, M_i = m\big) \\ &= \frac{k}{i}\binom{n}{i}^{-1}\sum_{m=1}^{n}\binom{m-1}{k}\binom{n-m}{i-k-1} \\ &= \frac{k}{i}\binom{n}{i}^{-1}\sum_{m=0}^{n-1}\binom{m}{k}\binom{n-m-1}{i-k-1} = \frac{k}{i}, \end{aligned}$$

where we have used the identity [24, p. 169]

$$\sum_{m=0}^{n}\binom{m}{k}\binom{n-m}{i-k} = \binom{n+1}{i+1}.$$

We now proceed similarly to the proof of Theorem 1 in [15]: since $\sum_{i=1}^{n}(1/i) = O(\log n)$, the expected cost for handling the remaining $n-k$ pairs is

$$\begin{aligned} E[\text{Cost}] &= \sum_{i=k+1}^{n}\Big[(k/i)O(\log k) + \big(1-(k/i)\big)O(1)\Big] \\ &< \sum_{i=1}^{n}\big[(k/i)O(\log k)\big] + O(n) \\ &= O(k\log k)\sum_{i=1}^{n}(1/i) + O(n) \\ &= O(n + k\log k\log n). \end{aligned}$$

The overall expected cost is thus $O(n + k\log k + k\log k\log n) = O(n + k\log k\log n)$. The upper and lower bounds are obtained by considering the scenarios in which the pairs are processed in order of descending and ascending weights, respectively.

REFERENCES

[1] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2012.

[2] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrák, "Eagle-eyed elephant: split-oriented indexing in Hadoop," in *EDBT*, 2013, pp. 89–100.

[3] S. Tirthapura and D. P. Woodruff, "Optimal random sampling from distributed streams revisited," in *DISC*, 2011.

[4] G. Cormode *et al.*, "Optimal sampling from distributed streams," in *PODS*, 2010.

[5] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac, "Adaptive MapReduce using situation-aware mappers," in *EDBT*, 2012.

[6] S. Agarwal *et al.*, "Blink and it's done: interactive queries on very large data," *VLDB*, 2012.

[7] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic sample selection for approximate query processing," in *SIGMOD*, 2003.

[8] S. Chaudhuri, G. Das, and V. R. Narasayya, "Optimized stratified sampling for approximate query processing," *ACM Trans. Database Syst.*, 2007.

[9] "Tez," http://tez.apache.org.

[10] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *ICDE*, 2011, pp. 1151–1162.

[11] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *PVLDB*, vol. 5, no. 11, pp. 1268–1279, 2012.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI '12*, 2012.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.

[14] "Hadoop," http://hadoop.apache.org.

[15] K. S. Beyer, R. Gemulla, P. J. Haas, B. Reinwald, and Y. Sismanis, "Distinct-value synopses for multiset operations," *Commun. ACM*, vol. 52, no. 10, pp. 87–95, 2009.

[16] E. Cohen and H. Kaplan, "Summarizing data using bottom-k sketches," in *PODS*, 2007, pp. 225–234.

[17] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *SIGMOD*, 1997, pp. 171–182.

[18] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Trans. Math. Softw.*, vol. 32, no. 1, pp. 1–16, 2006.

[19] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer, "Efficient jump ahead for 2-linear random number generators," *INFORMS Journal on Computing*, vol. 20, no. 3, pp. 385–390, 2008.

[20] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, 1985.

[21] H. A. David and H. N. Nagarja, *Order Statistics*, 3rd ed. Wiley, 2003.

[22] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie, "Online aggregation for large mapreduce jobs," *PVLDB*, 2011.

[23] B. Babcock and C. Olston, "Distributed top-k monitoring," in *SIGMOD*, 2003, pp. 28–39.

[24] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison-Wesley, 1994.