CHOPPER: Optimizing Data Partitioning for In-Memory Data Analytics Frameworks

Arnab Kumar Paul†, Wenjie Zhuang†, Luna Xu†, Min Li‡, M. Mustafa Rafique⋆, Ali R. Butt†
†*Virginia Tech*, ‡*IBM Almaden Research*, ⋆*IBM Research - Ireland*
{akpaul, kaito, xuluna, butta}@cs.vt.edu, minli@us.ibm.com, mustafa.rafique@ie.ibm.com

***Abstract*—The performance of in-memory based data analytic frameworks such as Spark is significantly affected by how data is partitioned. This is because the partitioning effectively determines task granularity and parallelism. Moreover, different phases of a workload execution can have different optimal partitions. However, in the current implementations, the tuning knobs controlling the partitioning are either configured statically or involve a cumbersome programmatic process for affecting changes at runtime.**

In this paper, we propose CHOPPER, a system for automatically determining the optimal number of partitions for each phase of a workload and dynamically changing the partition scheme during workload execution. CHOPPER monitors the task execution and DAG scheduling information to determine the optimal level of parallelism. CHOPPER repartitions data as needed to ensure efficient task granularity, avoids data skew, and reduces shuffle traffic. Thus, CHOPPER allows users to write applications without having to hand-tune for optimal parallelism. Experimental results show that CHOPPER effectively improves workload performance by up to 35.2% compared to standard Spark setup.

I. INTRODUCTION

Large scale in-memory data analytic platforms, such as Spark [30], [31], are being increasingly adopted by both academia and industry for processing data for a myriad of applications and data sources. These platforms are able to greatly reduce the amount of disk I/Os by caching the intermediate application data in-memory, and leverage more powerful and flexible direct acyclic graphs (DAG) based task scheduling. Thus, in-memory platforms outperform widely-used MapReduce [4]. The main advantage of a DAG-based programming paradigm is the flexibility it offers to the users in expressing their application requirements. However, the downside is that complicated task scheduling makes identifying application bottlenecks and performance tuning increasingly challenging.

There is a plethora of application-specific parameters that impact runtime performance in Spark, such as tasks parallelism, data compression and executor resource configuration. In typical data processing systems, the input (or intermediate) data is divided into logical subsets, called partitions. Specifically, in Spark, a partition can not be divided between multiple compute nodes for execution, and each compute node in the cluster processes one or more partitions. Moreover, a user can configure the number of partitions and how the data should be partitioned (i.e., hash or range partitioning schemes) for each Spark job. Suboptimal task partitioning or selecting a non-optimal partition scheme can significantly increase workload execution time. For instance, if a partition strategy launches too many tasks within a computation phase, this would lead to CPU and memory resource contention, and thus lose performance. Conversely, if too few tasks are launched, the system would have low resource utilization and would again result in reduced performance.

Moreover, inferior partitioning can lead to serious data skew across tasks, which would eventually result in some tasks taking significantly longer time to complete than others. As data processing frameworks usually employ a global barrier between computation phases, it is critical to have all the tasks in the same phase finish approximately at the same time, so as to avoid stragglers that can hold back otherwise fast-running tasks. The right scheme for data partitioning is the key for extracting high performance from the underlying hardware resources. However, finding a data partitioning scheme that gives the best or highest performance is non-trivial. This is because, data analytic workflows typically involve complex algorithms, e.g., machine learning and graph processing. Thus, the resulting task execution plan can become extremely complicated with increasing number of multiple computation phases. Moreover, given that each computation phase is different, the optimal number of partitions for each phase can also be different, further complicating the problem.

Spark provides two methods for users to control task parallelism. One method is to use a workload specific configuration parameter, $default.parallelism$, which serves as the default number of tasks to use for when the number of partitions is not specified. The second method is to use repartitioning APIs, which allow the users to repartition the data. Spark does not support changing of data parallelism between different computation phases except via manual partitioning within a user program through repartitioning APIs. This is problematic because the optimal number of partitions can be affected by the size of the data. Users would have to change and recompile the program every time they process a different data set. Thus, a clear opportunity for optimization is lost due to the rigid partitioning approach.

In this paper, we propose CHOPPER, an auto-partitioning system for Spark[1] that automatically determines the opti-

[1]We use Spark as our evaluation DAG-based data processing framework to implement and showcase the effectiveness of CHOPPER. We note that the proposed system can be applied to other DAG-based data processing framework, such as Dryad [10].

2168-9253/16 $31.00 © 2016 IEEE
DOI 10.1109/CLUSTER.2016.41

mal number of partitions for each computation phase during workload execution. CHOPPER alleviates the need for users to manually tune their workloads to mitigate data skewness and suboptimal task parallelism. Our proposed dynamic data partitioning is challenging due to several reasons. First, since Spark does not support changing tasks parallelism parameters for each computation phase, CHOPPER would need to design a new interface to enable the envisioned dynamic tuning of task parallelism. Second, the optimal data partitions differ across different computation phases of workloads. CHOPPER needs to understand application characteristics that affect the task parallelism in order to select an appropriate partitioning strategy. Finally, to adjust the number of tasks, CHOPPER may introduce additional data repartitioning, which may incur extra data shuffling overhead that has to be mitigated or amortized. Thus, a careful orchestration of the parameters is needed to ensure that CHOPPER's benefits outweigh the costs.

To address the above challenges, CHOPPER first modifies Spark to support dynamically changing data partitions through an application specific configuration file. CHOPPER checks different numbers of data partitions before scheduling the next computation phase. Information gathering about the application execution is achieved via several lightweight test runs, which are then analyzed to identify task profiles, data skewness, and optimization opportunities. CHOPPER uses this information along with a heuristic to compute a data repartitioning scheme, which minimizes the data skew, determines the right tasks parallelism for each computation phase, while minimizing the repartitioning overhead.

Specifically, this paper makes the following contributions.

1) We enable dynamic tuning of task parallelism for each computation phase in DAG-based in-memory analytics platforms such as Spark.
2) We design a heuristic to carefully compute suitable data repartitioning schemes with low repartitioning overhead. Our approach successfully identifies the data skewness and optimization opportunities and adjusts task parallelism automatically to yield higher performance compared to the default static approach.
3) We implement CHOPPER on top of Spark and evaluate the system to demonstrate its effectiveness. Our experiments demonstrate that CHOPPER can significantly outperform vanilla Spark by up to 35.2% for the studied workloads.

II. BACKGROUND AND MOTIVATION

In this section, we first discuss current data partitioning methodologies in Spark. Next, we present the motivation for our work by studying the performance impact of different number of data partitions on a representative workload, *KMeans* [7].

A. Spark Data Partitioning

In Spark, data is managed as an easy-to-use memory abstraction called resilient distributed datasets (RDDs) [30]. To process large data in parallel, Spark partitions an RDD into a collection of immutable partitions (*blocks*) across a set of machines. Each machine retains several blocks of an RDD. Spark tasks, with one-to-one relationship with the partitions, are launched on the machine that stores the partitions. Computation is done in the form of RDD actions and transformations, which can be used to capture the lineage of a dataset as a DAG of RDDs, and help in the recreation of an RDD in case of a failure. Such DAGs of RDDs are maintained in a specialized component, DAGScheduler, which schedules the tasks for execution.

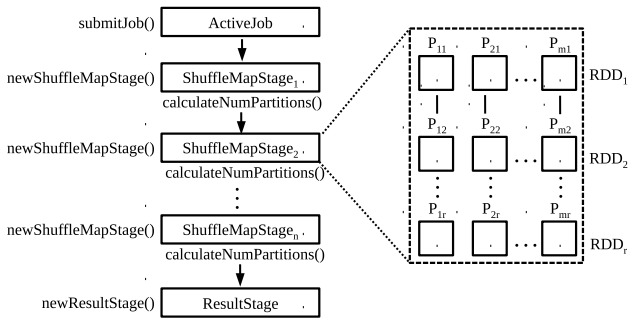

Fig. 1. Overview of Spark DAGScheduler.

Fig. 1 shows an overview of the Spark DAGScheduler. The input to DAGScheduler are called jobs (shown as ActiveJob in the figure). Jobs are submitted to the scheduler using a *submitJob* method. Every job requires computation of multiple stages to produce the final result. Stages are created by shuffle boundaries in the dependency graph, and constitute a set of tasks where each task is a single unit of work executed on a single machine. The narrow dependencies, e.g., *map* and *filter*, allow operations to be executed in parallel and are combined in a single stage. The wide dependencies, e.g., *reduceByKey*, require results to be combined from multiple tasks and cannot be confined to a single stage. Thus, there are two types of stages: ShuffleMapStage (shown in Fig. 1 as $ShuffleMapStage_1$, $ShuffleMapStage_2$, ..., $ShuffleMapStage_n$), which writes map output files for a shuffle operation, and ResultStage, i.e., the final stage in a job. ShuffleMapStage and ResultStage are created in the scheduler using *newShuffleMapStage* and *newResultStage* methods, respectively.

In Spark, tasks are generated based on the number of partitions of an input RDD at a particular stage. The same function is executed on every partition of an RDD by each task. In Fig. 1, $ShuffleMapStage_2$ is expanded to show the operations involved in a particular stage. Different operations in a stage form different RDDs (RDD_1, RDD_2, ... , RDD_r). Each RDD consists of a number of tasks that can be operated in parallel. In the figure, $P_{\alpha\beta}$ represents $partition_\alpha$ of RDD_β. There are m partitions for an RDD. Each RDD in a stage consists of a narrow dependency on the previous RDD, which enables parallel execution of multiple tasks. Thus, the number of partitions at each stage determins the level of parallelism.

Currently, Spark automatically determines the number of partitions based on the dataset size and the cluster setups. However, the number of partitions can also be configured

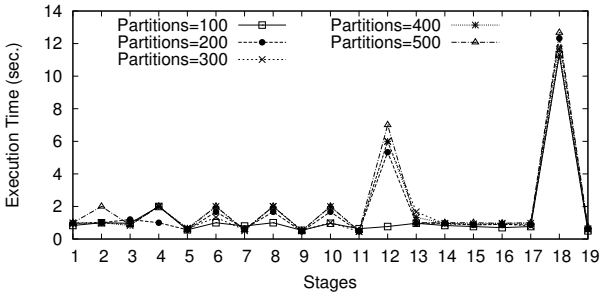

Fig. 2. Execution time per stage under different number of partitions.

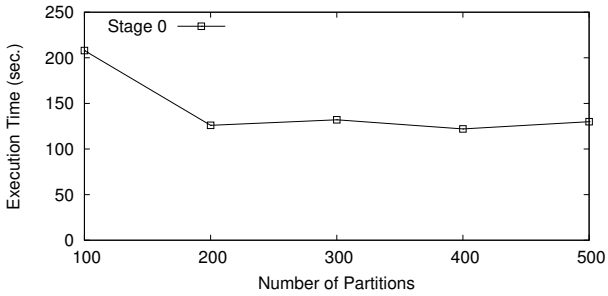

Fig. 3. Execution time of stage 0 under different partition numbers.

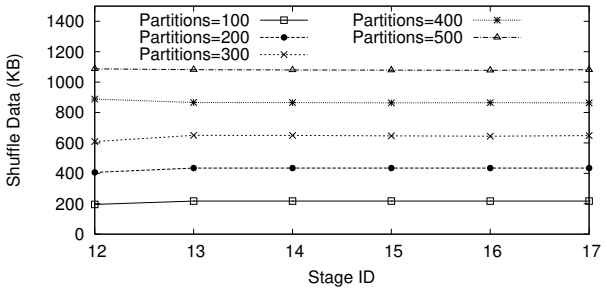

Fig. 4. Shuffle data per stage under different partition numbers.

manually using `spark.default.parallelism` parameter. In order to partition the data, Spark provides two data partitioning schemes, namely hash partitioner and range partitioner. Hash partitioner divides RDD based on the hash values of keys in each record. Data with the same hash keys are assigned to the same partition. Conversely, range partitioner divides a dataset into approximately equal-sized partitions, each of which contains data with keys within a specific range. Range partitioner provides better workload balance, while hash partitioner ensures that the related records are in the same partition. Hash partitioner is the default partitioner, however, users can opt to use their own partitioner by extending the *Partitioner* interface.

Although Spark provides mechanisms to automatically determine the number of partitions for a given RDD, it lacks application related knowledge to determine the best parallelism for a specific job. Moreover, the default hash partitioner is prone to creating workload imbalance for some input data. Spark provides the flexibility to tailor the configurations for workloads, however, it is not easy for users to determine the best configurations for each stage of a workload, especially when workloads may contain hundreds of stages.

B. Workload Study

To show the impact of data partitioning on application performance, we conduct a study using *KMeans* workload from SparkBench [14] and the latest release of Spark (version 1.6.1), with Hadoop (version 2.6) providing the HDFS [22] storage layer. Our experiments execute on a 6-node heterogeneous cluster. Three nodes (A, B, C) have 32 cores, 2.0 GHz AMD processors, 64 GB memory, and are connected through a 10 Gbps Ethernet interconnect. Two nodes (D, E) have 8 cores, 2.3 GHz Intel processors, 48 GB memory. The the remaining node (F) has 8 cores, 2.5 GHz Intel processor, and 64 GB memory. Nodes D, E and F are connected via a 1 Gbps Ethernet interconnect. Node F is configured to be the master node, while nodes A to E are worker nodes for both HDFS and Spark. Every worker node has one executor with 40 GB memory, and the remaining memory can be used for the OS buffer and HDFS data node operations. While our cluster hardware is heterogeneous, we configure each executor with the same amount of resources, essentially providing same resources to each component to better match with Spark's needs, and alleviating the performance impact due to the heterogeneity of the hardware. Note that, given the increasing heterogeneity in modern clusters, we do take the heterogeneity of cluster resource into account when designing CHOPPER. We repeated each experiment 3 times, and report the average results in the following.

First, we study the performance impact of different number of partitions. For this test, we use KMeans workload with 7.3 GB input data size. KMeans has 20 stages in total, and we change the number of partitions from 100 to 500 and record the execution time for each stage. Fig. 2 shows the results. For every stage, the number of partitions that yields minimum execution time varies. This shows that different stages have different characteristics and that the execution time for each stage can vary even under the same configuration. To further investigate the impact on performance, we study stage-0 in more detail. As shown in Fig. 3, the execution time of a stage changes with the number of partitions, and we see the worst performance when the number of partitions is set to 100. From this study, we observed that the number of partitions has an impact on the overall performance of a workload. Furthermore, different stages inside a workload can have different optimal number of partitions. In this example, specifying 100 partitions may be an optimal configuration for overall execution (Fig. 2), but clearly it is not optimal for stage-0 (Fig. 3).

To better understand the above observed performance impact, we investigate the amount of shuffle data produced at each stage with different number of partitions—since shuffle has a big impact on workload performance. For this test, we record the maximum of shuffle read or write data as representative of shuffle data per stage. For KMeans, only stages 12-17 involve data shuffle. As shown in Fig. 4, any increase in the number of partitions also increases the shuffle data

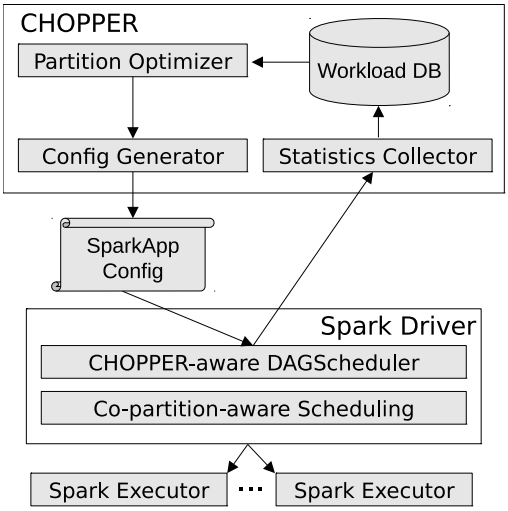

Fig. 5. System architecture of CHOPPER.

at each stage. A shuffle stage usually involves repartitioning RDD data. For an equivalent execution time, if repartitioning is not involved, then the amount of shuffle data increases from $434.83\ KB$ for 200 partitions to $1081.6\ KB$ for 500 partitions for stage-17, compared to $217.33\ KB$ of shuffle data when repartitioning is done. Also, when compared to a large number of partitions, e.g., 2000, there is a significant increase in the execution time as well as increase in the amount of shuffle data. For 2000 partitions, the execution time is 4.53 minutes and the amount of shuffle data for stage-17 is $4300.8\ KB$. We observe 38.8% improvement in execution time from repartitioning for similar amount of shuffle data. Also there is 46.1% improvement in execution time, and 94.9% reduction in the amount of shuffle data per stage when compared to large (i.e., 2000) number of partitions.

These experiments show that the number of partitions is an important configuration parameter in Spark and can help improve the performance of a workload. The optimal number of partitions not only varies with workload characteristics, but is also different among different stages of a workload. We leverage these findings in designing the auto-partitioning scheme of CHOPPER.

III. SYSTEM DESIGN

In this section, we present the design of CHOPPER, and how it achieves automatic repartitioning of RDDs for improved performance and system efficiency.

Fig. 5 illustrates the overall architecture of CHOPPER. We design and implement CHOPPER as an independent component outside of Spark. As Spark is a fast evolving system, we keep the changes to Spark for enabling our dynamic partitioning to a minimum. This reduces code maintenance overhead while ensuring adoption of CHOPPER for real-world use cases. CHOPPER consists of a partition optimizer, a configuration generator, a statistics collector, and a workload database. In addition, CHOPPER extends the Spark's DAGScheduler to support dynamic partitioning configuration and employs a

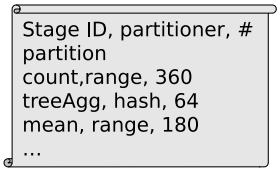

Fig. 6. An example generated Spark workload configuration in CHOPPER.

co-partition-aware scheduling mechanism to reduce network traffic whenever possible. *Statistics collector* communicates with Spark to gather runtime information and statistics of Spark applications. The collector can be easily extended to gather additional information as needed. *Workload DB* stores the observed information including the input and intermediate data size, the number of stages, the number of tasks per stage, and the resource utilization information. *Partition optimizer* retrieves application statistics, trains models, and computes an optimized partition scheme based on the current statistics and the trained models for the workload to be optimized. This information is also stored in *Workload DB* for future use. *Partition optimizer* then generates a workload specific configuration file. The extended dynamic partitioning DAGScheduler changes the number of partitions and the partition scheme per stage according to the generated Spark configuration file. Finally, the co-partitioning-aware component schedules partitions that are in the same key range on the same machine if possible to decrease the amount of shuffle data. The partition optimizer does not need to consider data locality because the input of the repartitioning phase is the local output of the previous Map phase, and the destinations of the output of the repartitioning phase depend on the designated shuffle scheme. Thus, existing locality is automatically preserved.

Our system allows dynamic updates to the Spark configuration file whenever more runtime information is obtained. CHOPPER modifies Spark to allow applications to recognize, read, and adopt the new partition scheme.

A. Enable Auto-Partitioning

An example application configuration file produced by CHOPPER is shown in Fig. 6. It consists of multiple tuples each containing a signature; the partitioner, and the number of partitions for a particular stage. We use stage signatures to identify stages that invoke identical transformations and actions. This is helpful when the number of iterations within a machine learning workload is unknown, where we can: (a) use the same partition scheme for all the iterations; or (b) use previously trained models to dynamically determine the number of partitions to use if the intermediate data size changes across iterations.

When an application is submitted to a Spark cluster, a Spark driver program is launched in which a SparkContext is instantiated. SparkContext then initiates our auto-partitioning aware DAGScheduler. The scheduler checks the Spark configuration file before a stage is executed. If the partition scheme is different from the current one, the scheduler changes the scheme based on the one specified in the configuration file.

Each RDD has five internal properties, namely, partition list, function to compute for every partition's dependency list on other RDDs, partitioner, and a list of locations to compute every partition. CHOPPER changes the partitioner properties to enable repartitioning across stages.

CHOPPER also supports dynamic updates to the Spark application configuration file based on the runtime information of current running workload. In particular, DAGScheduler periodically checks the updated configuration file and uses the updated partitioning scheme if available. This improves the partitioning efficiency and overall performance.

B. Determine Stage-Level Partition Scheme

The partition optimizer is responsible for computing a desirable partition scheme for each stage of a workload, given the collected workload history, and current input data and size. The optimizer not only considers the execution time and shuffle data size of a stage but also the shuffle dependencies between RDDs. In the following, we describe how this is achieved for the stage-level information. The use of global DAG information is discussed in Section III-C.

Spark provides two types of partitioners, namely range partitioner and hash partitioner. Different data characteristics and data distributions require different partitioners to achieve optimal performance. Range partitioner creates data partitions with approximately same-sized ranges. RDD tuples that have keys in the same range are allocated to the same partition. Spark determines the ranges through sampling of the content of RDDs passed to it when creating a range partitioner. Thus, the effectiveness of a range partitioner highly depends on the data contents. A range partition scheme that distributes a RDD evenly is likely to partition another RDD into a highly-skewed distribution. In contrast, a hash partitioner allocates tuples using a hash function modulo of the number of partitions. The hash partitioner attempts to partition data evenly based on a hash function and is less sensitive to the data contents, and can produce even distributions. However, if the dataset has hot keys, a partition can become skewed in terms of load, as identical keys are mapped to the same partition. Consequently, the appropriate choice between the range partitioner or hash partitioner depends on the dataset characteristics and DAG execution patterns.

To compute the stage level partition scheme, we aim to minimize both the stage execution time and the amount of shuffle data. Considering stage execution time, and shuffle data that directly affects the execution time, enables us to capture the right task granularity. This prevents the partitions from both growing unexpectedly large—creating resource contention—or becoming trivially small—under-utilizing resources and incurring extra task scheduling overhead. The approach also implicitly alleviates task skew by filtering out inferior partition schemes.

Equations 1, 2, 3 and 4 describe the model learned and the objective function used to determine the optimal number of partitions. In particular, *D* denotes the size of input data for the stage, *P* denotes the number of partitions, t_{exe} represents the execution time of the stage, and $s_{shuffle}$ is the amount of shuffle data in the stage. t'_{exe} and $s'_{shuffle}$ denote the stage execution time and amount of shuffle data obtained using default parallelism, respectively. Given input data size, Equation 4 enables CHOPPER to determine the optimal number of partitions minimizing both execution time and the amount of shuffle data. By normalizing the execution time and the amount of shuffle data with respect to the respective values under default parallelism, we are able to capture both of our constraints into the same objective function. Constants α and β can be used to adjust the weights between the two factors. In our implementation, we set the constants to a default value of 0.5, making them equally important.

We model the execution time and the amount of shuffle data based on the input data size of the current stage and the number of partitions as shown in Equations 1 and 2. This is a coarse grained model, since it is independent of Spark execution details and focuses on capturing the relationship between input size, parallelism, execution time and shuffle data. In particular, we posit that the execution time increases with the input data size, obeying a combination of cube, square, linear, and sub-linear curves. The amount of shuffle data also increases or decreases with the number of partitions according to a combination of cube, square, linear, and sub-linear curves. Note that our model can capture most applications observed in the real-world use cases for Spark. However, it may not be able to model corner cases such as those with radically different behavior, e.g., workloads for which execution time grows with D^4. In general, we observe that such a model is simple and computationally efficient, yet powerful enough to capture applications with different characteristics via various coefficients of the model. When the cluster resources and other configuration parameters are fixed, the model fits the actual execution time and amount of shuffle data well with varying size of input data and number of partitions. The data points needed to train the models are gathered by the statistics collector. If the collected data points are not sufficient, CHOPPER can initiate a few test runs by varying the sampled input data size and the number of partitions and record the execution time and the amount of shuffle data produced. CHOPPER also remembers the statistics from the user workload execution in a production environment, which can be further leveraged to better train and model the current application behavior.

$$t_{exe} = a_1 D^3 + b_1 D^2 + c_1 D + d_1 D^{1/2} + e_1 P^3 + f_1 P^2 + g_1 P + h_1 P^{1/2}, \quad (1)$$

$$s_{shuffle} = a_2 D^3 + b_2 D^2 + c_2 D + d_2 D^{1/2} + e_2 P^3 + f_2 P^2 + g_2 P + h_2 P^{1/2}, \quad (2)$$

$$cost = \alpha t_{exe}/t'_{exe} + \beta s_{shuffle}/s'_{shuffle} \quad (3)$$

$$\min\ cost \quad (4)$$

Algorithm 1: Get Stage level Partition Scheme *getStagePar*.

Input: workload w, stage s, input size d
Output: ($partitioner$, $numPar$, $cost$)
begin
  $rModel$=getRangeParitionModel(w,s)
  $hModel$=getHashPartitionModel(w,s)
  ($numRangePar$,$rCost$)=getMinPar($rModel$,d)
  ($numHashPar$,$hCost$)=getMinPar($hModel$,d)
  if $rCost < hCost$ **then**
    return ($RangePartitioner$,$numRangePar$,$rCost$)
  else
    return ($HashPartitioner$,$numHashPar$,$hCost$)

Algorithm 2: Get workload partition scheme *getWorkloadPar*.

Input: workload w, DAG dag, input size D
Output: $parList$
begin
  if $dag == null$ **then**
    dag = getDAG(w)
  for *stage* s *in dag* **do**
    d=getStageInput(w, s, D)
    ($partitoner$, $numPar$, $cost$) = getStagePar(w, s, d)
    $parList$.add(s,$partitioner$,$numPar$,$cost$)
  return parList

CHOPPER trains two models using Equations 1 and 2 for every stage of a workload, one for range partitioning and the other for hash partitioning. Algorithm 1 presents how CHOPPER calculates the optimized stage level partition scheme given workload w, stage s and input size d for the stage. The algorithm returns the partitioner, the optimal number of partitions used for stage s and the cost. Specifically, CHOPPER retrieves the trained models of stages for both range partitioning and hash partitioning from the workload database. After this, Algorithm 1 computes the optimal numbers of partitions with minimal cost for both range partitioning and hash partitioning using Equation 4. Finally, CHOPPER returns the partitioner that would incur the lowest cost along with the number of partitions to use.

C. Globally-Optimized Partition Scheme

After we compute the stage level partition scheme, a naive solution is to compute the optimal partition scheme for each stage independently and generate the Spark configuration file. This is shown in Algorithm 2. It gets the DAG information from workload database, iterates through the DAG, computes the desirable partition scheme for each stage, and adds to a list of partition scheme. Lastly, the algorithm returns the list of partition schemes, which is then used to generate the Spark configuration file for the current workload.

Although Algorithm 2 optimizes the partition scheme per stage, it misses the opportunities to reduce shuffle traffic because of the dependencies between stages and RDDs. For example, if stage-C joins the RDDs from stage-A and stage-B, the shuffle traffic introduced by join can be completely eliminated if the two use the same partition scheme and the joined partitions are allocated on the same machine. However, this cannot be achieved using Algorithm 2. If the computed optimal scheme of stage-A is $(Range, 100)$ and the optimal scheme of stage-B is $(hash, 200)$, the shuffle data cannot be eliminated, as the partition schemes of stage-A and stage-B are different. Even though stage-A and stage-B are optimally partitioned, the shuffle data of stage-C is sub-optimal. Since join and co-group operation are two of the most commonly used operations in Spark applications, poor partitioning will typically introduce significant shuffle overhead. Consequently, it is critical to optimize the join and co-group operation to decrease the amount of shuffle data as much as possible.

Algorithm 3: Get globally optimized partition scheme.

Input: workload w, input size D
Output: $parList$
begin
  dag=getReGroupedDAG(w)
  for *node* s *in dag* **do**
    d=getStageInput(w, s, D)
    if s *isInstanceOf Stage* **then**
      ($partitoner$, $numPar$, $cost$) = getStagePar(w, s, d)
    else
      ($partitoner$, $numPar$, $cost$) = getSubGraphPar(w, s, d)
    if s *isFixed* **then**
      $curCost$ = getCost(w, s, getPartitioner(w, s), getNumPar(w, s))
      $optCost$ = $cost$ + getRepartitionCost(w, s, $partioner$, $numPar$)
      if $optCost < curCost$ **then**
        s' = s + "$repartitionstage$"
        $parList$.add(s', $partitioner$, $numPar$, $optCost$)
    else
      $parList$.add(s, $partitioner$, $numPar$, $cost$)
  return parList
Function *getSubGraphPar*
  Input: workload w, DAG dag, input size D
  Output: $paritioner, numPar, cost$
  $parList$ = getWorkloadPar(w, dag, D)
  min = $parList(0)$
  for s *in* $parList$ **do**
    $cost$ = getCost(w, dag, $s.partitioner$, $s.numPar$)
    if $cost < min.cost$ **then**
      min = s
  return ($min.partitioiner, min.numPar, min.cost$)
Function *getCost*
  Input: workload w,DAG dag, $partitioner$, $numPar$
  Output: $cost$
  for *stage* s *in dag* **do**
    if $partitioner == range$ **then**
      $rModel$ = getRangePartitionModel(w, s)
      $cost$ += Equation 3
    else
      $hModel$ = getHashParitionModel(w, s)
      $cost$ += Equation 3
  return $cost$

Workload	KMeans	PCA	SQL
Input Size (GB)	21.8	27.6	34.5

TABLE I: Workloads and input data sizes.

Another issue is that since the users are allowed to tune and specify customized partition scheme on their own, CHOPPER leaves the user optimization intact even when the computed optimal scheme disagrees with the user specified partition scheme. However, CHOPPER can choose to add an additional partition operation if the benefit of introducing the partition operation significantly outweighs the overhead incurred. For instance, consider a case where stage-B blows up the number of tasks to by a power of two from its previous stage-A (i.e., 100^2 tasks) due to the user-fixed partition scheme of stage-A. If CHOPPER coalesces the number of tasks of stage-A from 100 to 10, it would significantly reduce the number of tasks in stage-B from 10000 to 100.

To remedy this, CHOPPER determines the partition scheme by globally considering the entire DAG execution. As described in Algorithm 3, CHOPPER first groups the DAG graph based on the stage dependencies. The grouping of DAG graph is started from the end stages of the graph and iterated towards the source stages. The grouping is based on the join operations or partition dependencies. The stages with join operations are grouped into a subgraph. The partition dependencies refer to the cases where the number of stages is determined by the previous stage, thus CHOPPER cannot change the partition scheme. After the DAG of the workload is regrouped, the node within the new DAG can either be a stage or a subgraph that consists of multiple stages. If the node is a stage, the optimal partition scheme is computed using Algorithm 1. Otherwise, the node is a subgraph, where the optimal partition scheme is computed differently. Specifically, we iterate through all the nodes within the subgraph and get the optimal partition scheme for each node, we then compute the cost of applying each partition scheme to all the applicable nodes in the graph and return the partition scheme that has the minimal cost.

Finally, after we compute the globally optimal partition scheme, we check whether the stage partition is allowed to be changed. If not, and the partition scheme is different, we then check whether it is beneficial to insert a new repartitioning phase by comparing the cost using original partitioning to the cost of the new repartitioning phase together with the cost of optimized partition scheme. If the benefit outweighs the cost by a factor of γ, we choose to insert a new partition phase into the DAG graph. We empirically set γ to 1.5 to tolerate the model estimation error.

IV. EVALUATION

In this section, we evaluate CHOPPER and demonstrate its effectiveness on the cluster described earlier in Section II. We use three representative workloads from SparkBench: KMeans, PCA, and SQL. KMeans [7] is a popular clustering algorithm that partitions and clusters n data points into k clusters in which each data point is assigned to the nearest center

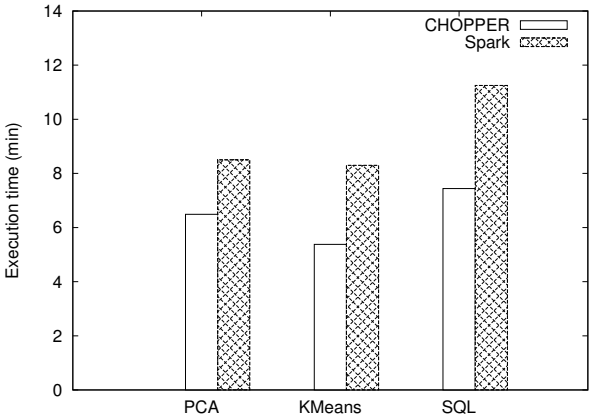

Fig. 7. Execution time of Spark and CHOPPER.

point. The computation requirement of this workload change according to the number of clusters, the number of data points, and the machine learning workload that exhibits different resource utilization demand for different stages during the process of iteratively calculating k clusters. PCA [11] is a commonly used technique to reduce the number of features in various data mining algorithms such as SVM [3] and logistic regression [8]. It is both computation and network-intensive machine learning workload that involves multiple iterations to compute a linearly uncorrelated set of vectors from a set of possibly correlated ones. SQL is a workload that performs typical query operations that count, aggregate, and join the data sets. Thus, SQL represents a common real world scenario. SQL is compute intensive for count and aggregation operations and shuffle intensive in the join phase. The input data is generated by the corresponding data generator within SparkBench. The input data size for each workload is shown in Table I. The experiments for vanilla Spark are conducted with the default configuration, which is set to 300 partitions for all the workloads. We run all of our experiments three times, and the numbers reported here are from the average of these runs. Moreoer, we clear the OS cache between runs to preclude the impact of such caching on observed times.

A. Overall Performance of CHOPPER

Our first test evaluates the overall performance impact of CHOPPER. Fig. 7 illustrates the total execution time of three workloads comparing CHOPPER against standard vanilla Spark. The reported execution time includes the overhead of repartitioning introduced by CHOPPER. We can see that CHOPPER achieves overall improvement in the execution time by 23.6%, 35.2% and 33.9% for PCA, KMeans and SQL, respectively. This is because CHOPPER effectively detects optimal partitioner and the number of partitions for all stages within each workload. CHOPPER also performs global optimization to further reduce network traffic by intelligently co-partitioning dependent RDDs and inserting repartition operations when the benefits outweigh the cost. We also observe that CHOPPER is effective for all types of workloads that exhibit

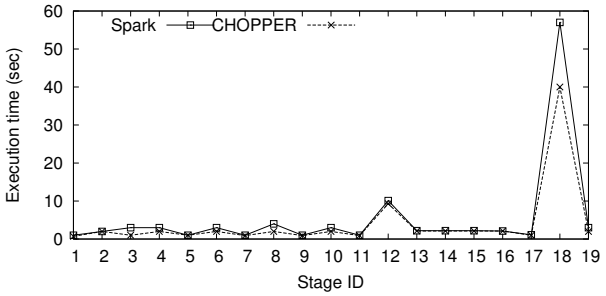

Fig. 8. Execution time per stage breakdown of KMeans.

	Chopper	Spark
Execution Time (sec)	250	372

TABLE II: Execution time for stage 0 in KMeans.

different resource utilization characteristics. The repartition method of CHOPPER implicitly reduces the potential data skew and determines the right task granularity for each workload, thus improving the cluster resource utilization and workload performance. Thus, CHOPPER shows significant reduction in the overall execution time for all the three workloads. The model training of CHOPPER is conducted offline, and thus is not in the critical path of workload execution. Moreover, the overhead of repartitioning is negligible as it involves solving a simple linear programming problem.

B. Timing Breakdown of Execution Stages

To better understand how dynamic partitioning of CHOPPER helps to improve overall performance, in our next test, we examine the detailed timing breakdown of individual workload stages. Fig. 8 depicts the execution time per stage for KMeans. We show the execution time of stage-0 separately in Table II since the execution time of stage-0 and that of other stages differs significantly. We see that CHOPPER reduces execution time of each stage for KMeans compared to vanilla Spark, as CHOPPER is able to customize partition schemes for each stage according to associated history and runtime characteristics. Table III shows the number of partitions used by CHOPPER for different stages compared to vanilla Spark. Stages 12 to 17 are iterative, and thus are assigned the same number of partitions. We see that CHOPPER effectively detects and changes to the correct number of partitions for this workload rather than using a fixed (default) value throughout the execution.

C. Impact on Shuffle Stages

In our next test, we use a shuffle-intensive workload, SQL, to study how CHOPPER reduces the shuffle traffic by automatically recognizing and co-partitioning dependent RDDs. Fig. 9 shows that the shuffle data for all four stages is less under CHOPPER compared to vanilla Spark. Stage-4 (not shown in the figure) has the same amount of shuffle data for SQL workload using CHOPPER or Spark (i.e., 4.7 GB). However, as seen in Fig. 10, stage-4 takes comparatively shorter time to

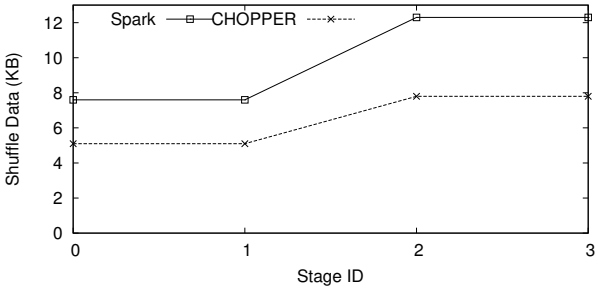

Fig. 9. Shuffle data per stage for SQL.

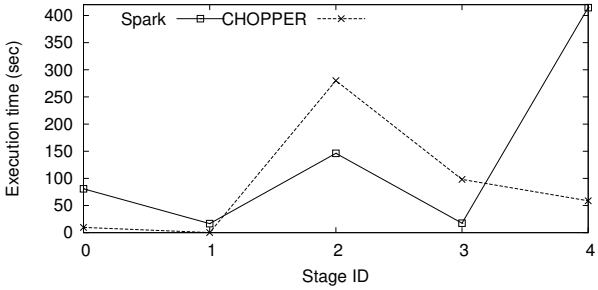

Fig. 10. Execution time per stage breakdown of SQL.

execute using CHOPPER versus Spark. This is because, stage 4 is divided into 4 sub-stages where the first two sub-stages have a shuffle write data of 1.9 GB and 2.8 GB. CHOPPER combines these two sub-stages for shuffle write and provides the third sub-stage for the shuffle data to be read. This greatly reduces the execution time for stage 4 as seen in Fig. 10. Thus, we demonstrate that CHOPPER can effectively detect dependent RDDs and co-partition them to reduce the shuffle traffic and improve the overall workload performance.

D. Impact on System Utilization

In our next experiment, we investigate how CHOPPER impacts the resource utilization of all the studied workloads. Fig. 11, 12, 13 and 14 depict the CPU utilization, memory utilization, total number of transmitted and received packets per second, and the total number of read and write transactions per second, respectively, during the execution of the workloads under CHOPPER and vanilla Spark. The numbers show the average of the statistics collected from the six nodes in our cluster setup. We observe that the performance of CHOPPER is either equivalent or in most of the cases better than the performance of vanilla Spark for the studied workloads. In some cases, CHOPPER shows improved transactions per seconds as compared to vanilla Spark because of the high throughput and improved system performance.

These experiments show that the performance (computed on the basis of execution time and shuffle data) improves under CHOPPER compared to vanilla Spark. Also, these improvements in CHOPPER yield comparable or better system utilization compared to vanilla Spark.

StageID	0	1	2	3	4	5	6	7	8	9	10	11	12 - 17	18	19
CHOPPER	210	210	300	720	300	720	300	720	300	720	300	720	210	380	210
Spark	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300

TABLE III: Repartition of stages using CHOPPER.

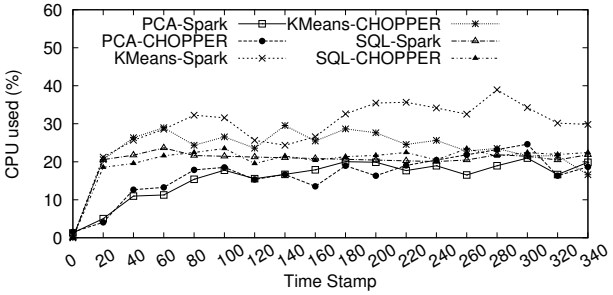

Fig. 11. CPU utilization.

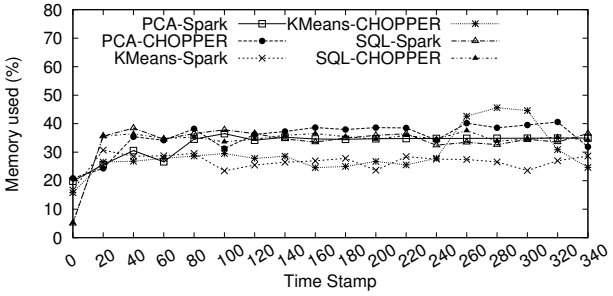

Fig. 12. Memory utilization.

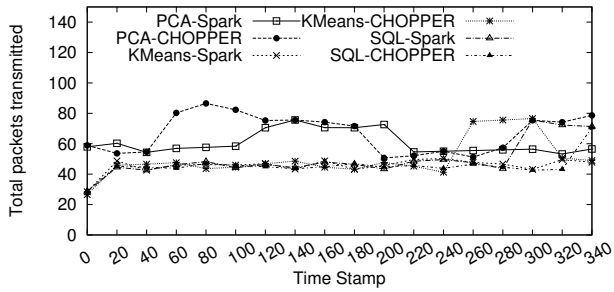

Fig. 13. Total transmitted and received packets per second.

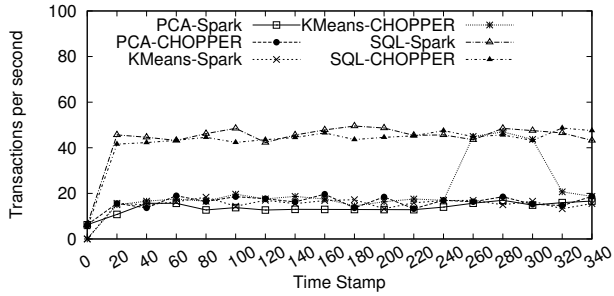

Fig. 14. Transactions per second.

V. RELATED WORK

A number of recent works have focused on improving the performance of big data processing frameworks by designing better built-in data partitioning. In the following, we discuss projects that are closely related to CHOPPER.

Shark [28] supports a column-oriented in-memory storage, using RDDs [30], to efficiently run SQL queries and iterative machine learning functions. This is achieved by re-planning query execution mid-query if needed. Although Shark optimizes execution plan of a workload while the workload is running, unlike CHOPPER, Shark does not alter the number of partitions at each Spark compute phase, which can help optimize data movement between cluster nodes and load balance between tasks.

Spartan [9] automatically partitions the data to improve data locality in processing large multi-dimensional arrays in a distributed setting. It transforms the user code into an expression graph based on high-level operators, namely map, fold, filter, scan and join_update, to determine the communication costs of data distribution between the compute nodes. Although Spark can be used to implement Spartan, in contrast to CHOPPER, the partitioning approach proposed in Spartan can not be applied directly to determine the optimal number of partitioning in Spark at each stage (where a stage may present unique execution characteristics that may not be represented by high-level operators).

Repartitioning MapReduce tasks has been actively studied [4], [5], [13], [15]. PIKACHU [5] improves load balancing of MapReduce [4] workloads on clusters with heterogeneous compute capabilities. It specifically targets the reduce phase of MapReduce execution and schedules jobs on slow and fast nodes such that jobs completion times are evened out across all nodes. On the other hand, Stubby [15] optimizes the given MapReduce workflow by combining multiple MapReduce jobs into a single job. It searches through the plan space of a given workflow and applies multiple optimizations, such as vertical packing to combine map and reduce operations from multiple jobs for reducing the network traffic. Similarly, SkewTune [13] mitigates the skewness in MapReduce applications by first detecting if a node has become idle in a cluster and then by scheduling the longest job on the idle node. The optimizations proposed in these systems share the goal of repartitioning with CHOPPER, but are specific to MapReduce programming model, and can not be applied directly to improve the partitioning scheme in a Spark application. This is because, the data partitioning for Spark need to consider characteristics across multiple compute phases within a workflow, and not just Map and Reduce phase. This concept can also help in virtual machine management in cloud computing [17].

The Hardware Accelerated Range Partitioning (HARP) [26]

technique leverages specialized processing elements to improve the balance between memory throughput and energy efficiency by eliminating the compute bottlenecks of the data partitioning process. HARP makes the case that using dedicated hardware for data partitioning outperforms its software counterparts and achieves higher parallelism. The approach proposed in HARP is orthogonal to our work, and can be used in conjunction with CHOPPER.

Several other works also propose solutions to optimize the data partitioning problem in order to improve the processing and storage performance of multi-processor systems [2], [12], [24], [32], cloud storage systems [20], [25], [29], database systems [16], [18], [19], [27], and graph processing systems [1], [6], [21], [23]. While not directly applicable to the context of Spark and CHOPPER, the techniques proposed in these works can be leveraged in CHOPPER to further improve the data partitioning approaches inline with other system-level constraints, e.g., storage optimization.

VI. CONCLUSION

In this paper, we design CHOPPER, a dynamic partitioning approach for in-memory data analytic platforms. CHOPPER determines the optimal number of partitions and the partitioner for each stage of a running workload with the goal of minimizing the stage execution time and shuffle traffic. CHOPPER also considers the dependencies between stages, including join and cogroup operations, to further reduce shuffle traffic. By minimizing the stage execution time and shuffle traffic, CHOPPER implicitly alleviates the task data skew using different partitioners and improves the task resource utilization through optimal number of partitions. Experimental results demonstrate that CHOPPER effectively improves overall performance by up to 35.2% for representative workloads compared to standard vanilla Spark.

Our current implementation of CHOPPER has to re-train its models whenever the available resources are changed. In future, we plan to explore the per-stage performance models that can work across different resource configurations, i.e., clusters. We will also explore how CHOPPER behaves under failures. These will further improve the applicability of CHOPPER in a cloud environment, where compute resources are failure-prone and scaled as needed.

VII. ACKNOWLEDGMENT

This work is sponsored in part by the NSF under the grants: CNS-1405697, CNS-1422788, and CNS-1615411.

REFERENCES

[1] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. ACM EuroSys*, 2015.

[2] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *Proc. ACM Data Management on New Hardware*, 2008.

[3] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*, 2004.

[5] R. Gandhi, D. Xie, and Y. C. Hu. Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proc. USENIX ATC*, 2013.

[6] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. USENIX OSDI*, 2014.

[7] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[8] D. W. Hosmer Jr and S. Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[9] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao. Spartan: A distributed array framework with smart tiling. In *Proc. USENIX ATC*, 2015.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. ACM Eurosys*, 2007.

[11] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[12] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *Proc. ACM ASPLOS*, 2008.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proc. ACM SIGMOD*, 2012.

[14] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proc. ACM International Conference on Computing Frontiers*, 2015.

[15] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. In *Proc. VLDB*, 2012.

[16] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proc. ACM SIGMOD*, 2011.

[17] A. K. Paul, S. K. Addya, B. Sahoo, and A. K. Turuk. Application of greedy algorithms to virtual machine distribution across data centers. In *Annual IEEE India Conference (INDICON)*, 2014.

[18] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. ACM SIGMOD*, 2012.

[19] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proc. ACM International Conference on Extending Database Technology*, 2013.

[20] C. Selvakumar, G. J. Rathanam, and M. R. Sumalatha. Pdds - improving cloud data storage security using data partitioning technique. In *Proc. IEEE International Advance Computing Conference*, 2013.

[21] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proc. ACM SIGMOD*, 2013.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. IEEE MSST*, 2010.

[23] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proc. ACM International Conference on Web Search and Data Mining*, 2014.

[24] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve. Automated data partitioning for highly scalable and strongly consistent transactions. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):106–118, 2016.

[25] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud. In *Proc. ACM VLDB*, 2012.

[26] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proc. ACM ISCA*, 2013.

[27] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proc. ACM SoCC*, 2011.

[28] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proc. ACM SIGMOD*, 2013.

[29] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Perform. Eval. Rev.*, 40(4):23–32, 2013.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX NSDI*, 2012.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. USENIX HotCloud*, 2010.

[32] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on multicore and multi-gpu platforms using functional performance models. *IEEE Transactions on Computers*, 64(9):2506–2518, 2015.