Optimizing Shuffle in Wide-Area Data Analytics

Shuhao Liu, Hao Wang, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
{shuhao, haowang, bli}@ece.toronto.edu

***Abstract*—As increasingly large volumes of raw data are generated at geographically distributed datacenters, they need to be efficiently processed by data analytic jobs spanning multiple datacenters across wide-area networks. Designed for a single datacenter, existing data processing frameworks, such as Apache Spark, are not able to deliver satisfactory performance when these wide-area analytic jobs are executed. As wide-area networks interconnecting datacenters may not be congestion free, there is a compelling need for a new system framework that is optimized for wide-area data analytics.**

In this paper, we design and implement a new proactive data aggregation framework based on Apache Spark, with a focus on optimizing the network traffic incurred in shuffle stages of data analytic jobs. The objective of this framework is to strategically and proactively aggregate the output data of mapper tasks to a subset of worker datacenters, as a replacement to Spark's original passive fetch mechanism across datacenters. It improves the performance of wide-area analytic jobs by avoiding repetitive data transfers, which improves the utilization of inter-datacenter links. Our extensive experimental results using standard benchmarks across six Amazon EC2 regions have shown that our proposed framework is able to reduce job completion times by up to 73%, as compared to the existing baseline implementation in Spark.

***Keywords*-MapReduce; shuffle; inter-datacenter network.**

I. Introduction

Modern data processing frameworks, such as Apache Hadoop [1] and Spark [2], are routinely used as the foundation for running large-scale data analytic applications. These frameworks are designed to work effectively within a single datacenter; yet as the volume of raw data to be analyzed increases exponentially, it is increasingly necessary to run large jobs across multiple datacenters that are geographically distributed around the world.

Intuitively, due to the time and bandwidth cost for moving data across datacenters, it would be more efficient to process data locally as much as possible. However, most existing data processing frameworks are designed to operate within a single cluster, and oblivious to *data locality* at the datacenter level. As a result, excessive inter-datacenter data transfers are likely to occur. The challenge, known as *wide-area data analytics* [3], is how to maximize the performance of a data analytic job when its tasks are distributed across multiple geo-distributed datacenters.

Existing approaches to address this challenge attempted to make better resource scheduling decisions for the sake of improving performance. They optimize the execution of a wide-area data analytic job by intelligently assigning individual tasks to datacenters, such that the overhead of moving data across datacenters can be minimized. For example, Geode [4], WANAnalytics [5] and Pixida [3] have proposed various task placement strategies, reducing the volume of inter-datacenter traffic. Iridium [6] achieves shorter job completion times by leveraging a redistributed input dataset, along with mechanisms for making optimal task assignment decisions.

Despite their promising outlook, even the best scheduling strategies may not achieve optimality due to the level of abstraction needed to solve the problem. As an example, in Spark, resource schedulers can only operate at the granularity of computation tasks. The induced inter-datacenter data transfers under the hood, which impact performance directly, are hidden from these schedulers. Consequently, all existing proposals have to make certain assumptions, which make them less practical. For example, existing work [3]–[5] assumed that intermediate data sizes are known beforehand, even though this is rarely the case in practice.

In this paper, we propose to take a systems-oriented approach to reduce the volume of data transfers across datacenters. Our new system framework is first and foremost designed to be *practical*: it has been implemented in Apache Spark to optimize the *runtime performance* of wide-area analytic jobs in a variety of real-world benchmarks. To achieve this objective, our framework focuses on the shuffle phase, and strategically aggregate the output data of mapper tasks in each shuffle phase to a subset of datacenters. In our proposed solution, the output of mapper tasks is proactively and automatically *pushed* to be stored in the destination datacenters, without requiring any intervention from a resource scheduler. Our solution is orthogonal and complementary to existing task assignment mechanisms proposed in the literature, and it remains effective even with the simplest task assignment strategy.

Compared to existing task assignment mechanisms, the design philosophy in our proposed solution is remarkably different. The essence of traditional task assignment is to move a computation task to be closer to its input data to ex-

ploit data locality; in contrast, by proactively moving data in the shuffle phase from mapper to reducer tasks, our solution improves data locality even further. As the core of our system framework, we have implemented a new method, called `transferTo()`, on Resilient Distributed Datasets (RDDs), which is a basic data abstraction in Spark. This new method proactively sends data in the shuffle phase to a specific datacenter that minimizes inter-datacenter traffic. It can be either used explicitly by application developers or embedded implicitly by the job scheduler. With the implementation of this method, the semantics of aggregating the output data of mapper tasks can be captured in a simple and intuitive fashion, making it straightforward for our system framework to be used by existing Spark jobs.

With the new `transferTo()` method at its core, our new system framework enjoys a number of salient performance advantages. *First*, it pipelines inter-datacenter transfers with the preceding mappers. Starting data transfers early can help improve the utilization of inter-datacenter links. *Second*, when task execution fails at the reducers, repetitive transfers of the same datasets across datacenters can be avoided, since they are already stored at the destination datacenter by our new framework. *Finally*, the application programming interface (API) in our system framework is intentionally exposed to the application developers, who are free to use this mechanism explicitly to optimize their job performance.

We have deployed our new system framework in a Spark cluster across six Amazon EC2 regions. By running workloads from the HiBench [7] benchmark suite, we have conducted a comprehensive set of experimental evaluations. Our experimental results have shown that our framework speeds up the completion time of general analytic jobs by 14% to 73%. Also, with our implementation, the impact of bandwidth and delay jitters in wide-area networks is minimized, resulting in a lower degree of performance variations over time.

II. Background and Motivation

In a typical MapReduce job, the input datasets are split into *partitions* that can be processed in parallel. Logical computation is organized in several consecutive `map()` and `reduce()` operations: `map()` operates on each individual partition to filter or sort, while `reduce()` collects the summary of results. An all-to-all communication pattern will usually be triggered between mappers and reducers, which is called a *shuffle* phase. These intermediate data shuffles are well known as costly operations in data analytic jobs, since they incur intensive traffic across worker nodes.

A. Fetch-based Shuffle

Both Apache Hadoop and Spark are designed to be deployed in a single datacenter. Since datacenter networks typically have abundant bandwidth, network transfers are considered even less expensive than local disk I/O in Spark [2]. With this assumption in mind, the shuffle phase is implemented with a *fetch*-based mechanism by default. To understand the basic idea in our proposed solution, we need to provide a brief explanation of the fetch-based shuffle in Spark.

In Spark, a data analytic job is divided into several *stages*, and launched in a stage-by-stage manner. A typical stage starts with a shuffle, when all the output data from the previous stages is already available. The workers of the new stage, *i.e.*, reducers in this shuffle, will *fetch* the output data from the previous stages, which constitutes the shuffle input, stored as a collection of local files on the mappers. Because the reducers are launched at the same time, shuffle input is fetched concurrently, resulting in a concurrent all-to-all communication pattern. For better fault tolerance, the shuffle input will not be deleted until the next stage finishes. When failures occur on the reducer side, the related files will be fetched from the mappers again, without the need to re-run them.

B. Problems with Fetch in Wide-Area Data Analytics

Though effective within a single datacenter, it is quite a different story when it comes to wide-area data analytics across geographically distributed datacenters, due to limited bandwidth availability on wide-area links between datacenters [8]. Given the potential bottlenecks on inter-datacenter links, there are two major problems with fetch-based shuffles.

First, as a shuffle will only begin when all mappers are finished — a barrier-like synchronization — inter-datacenter links are usually well *under-utilized* most of the time, but likely to be *congested* with bursty traffic when the shuffle begins. The links are under-utilized, because when some mappers finish their tasks earlier, their output cannot be transmitted immediately to the reducers. Yet, when the shuffle is started by all the reducers at the same time, they initiate concurrent network flows to fetch their corresponding shuffle input, leading to bursty traffic that may contend for the limited inter-datacenter bandwidth, resulting in potential congestion.

Second, when failures occur on reducers with the traditional fetch-based shuffle mechanism, data must be *fetched again* from the mappers over slower inter-datacenter network links. Since a stage will not be considered complete until all its tasks are executed successfully, The slowest tasks, called the *stragglers*, will directly affect the overall stage completion time. Re-fetching shuffle input over inter-datacenter links will slow down these stragglers even further, and negatively affects the overall job completion times.

III. Transferring Shuffle Input across Datacenters

To improve the performance of shuffle in wide-area data analytics, we will need to answer two important questions:

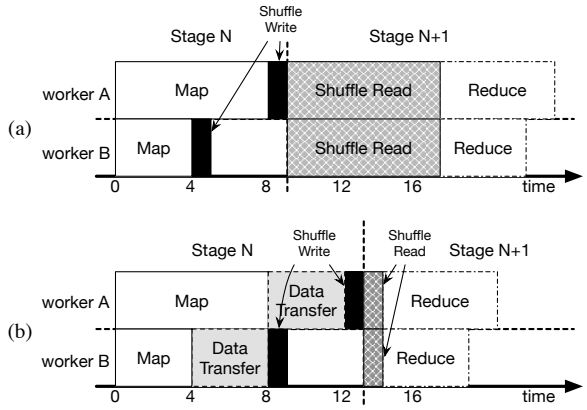

Figure 1: Mappers typically cannot finish their work at the same time. In this case, if we proactively push the shuffle input to the datacenter where the reducer is located (Fig. 1(b)), the inter-datacenter link will be better utilized as compared to leaving it on the mappers (Fig. 1(a)).

when and *where* should we transfer the shuffle input from mappers to reducers? The approach we have taken in our system framework is simple and quite intuitive: we should proactively *push* shuffle input as soon as any data partition is ready, and *aggregate* it to a subset of worker datacenters.

A. Transferring Shuffle Input: Timing

Both problems of fetch-based shuffle stem from the fact that the shuffle input is co-located with mappers in different datacenters. Therefore, they can be solved if, rather than asking reducers to fetch the shuffle input from the mappers, we can *proactively push* the shuffle input to those datacenters where the reducers are located, as soon as such input data has been produced by each mapper.

As an example, consider a job illustrated in Fig. 1. Reducers in stage $N+1$ need to fetch the shuffle input from mappers A and B, located in another datacenter. We assume that the available bandwidth across datacenters is $\frac{1}{4}$ of a single datacenter network link, which is an optimistic estimate. Fig. 1(a) shows what happens with the fetch-based shuffle mechanism, where shuffle input is stored on A and B, respectively, and transferred as soon as stage $N+1$ starts at $t = 10$. Two flows share the inter-datacenter link, allowing both reducers start at $t = 18$. In contrast, in Fig. 1(b), shuffle input is *pushed* to the datacenter hosting the reducer immediately after it is computed by each mapper. Inter-datacenter transfers are allowed to start at $t = 4$ and $t = 8$, respectively, without the need for sharing link bandwidth. As a result, reducers will be able to start at $t = 14$.

Fig. 2 shows an example of the case of reducer failures. With the traditional fetch-based shuffle mechanism, the failed reducer will need to fetch its input data again from another datacenter, if such data is stored with the mappers, shown in Fig. 2(a). In contrast, if the shuffle input is stored with the reducer instead when it fails, the reducer can read from the local datacenter, which is much more efficient.

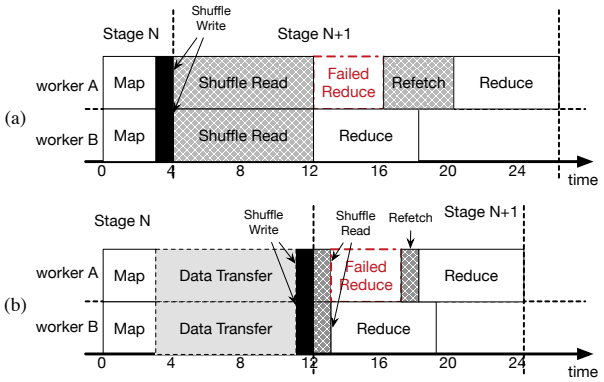

Figure 2: In the case of reducer failures, if we proactively push the shuffle input to the datacenter where the reducer is located (Fig. 2(b)), data re-fetching across datacenters can be eliminated, reducing the time needed for failure recovery as compared to the case where shuffle input is located on the mappers (Fig. 2(a)).

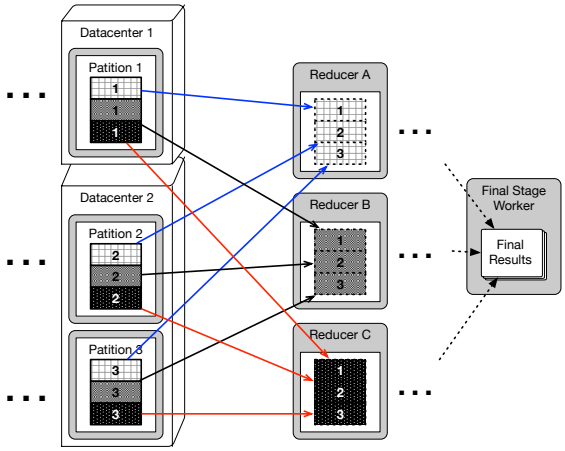

Figure 3: A snippet of a sample execution graph of a data analytic job.

B. Transferring Shuffle Input: Choosing Destinations

Apparently, proactively pushing shuffle input to be co-located with reducers is beneficial, but a new problem arises with this new mechanism: since the reduce tasks will not be placed until the map stage finishes, how can we decide the destination hosts of the proactive pushes?

It is indeed a tough question, because the placement of reducers is actually decided by the shuffle input distribution at the start of each stage. In other words, our choice of push destinations will in turn impact the reducer placement. Although it seems a cycle of unpredictability, but we think there already exist enough hints to give a valid answer in wide-area data analytics. Specifically, for the sake of minimizing cross-datacenter traffic, there is a tendency for both task and shuffle input placement at the datacenter level. We can exploit this tendency as a vital clue.

Our analysis starts by gaining a detailed understanding of shuffle behaviors in MapReduce. Fig. 3 shows a snippet of abstracted job execution graph and data transfers. The depicted shuffle involves 3 partitions of shuffle input, which will then be dispatched to 3 reducers. In this case, each

partition of the shuffle input is saved as 3 shards based on specific user-defined rules, *e.g.,* the keys in the key-value pairs. During data shuffle, each shard will be fetched by the corresponding reducer, forming an all-to-all traffic pattern. In other words, every reducer will access all partitions of the shuffle input, fetching the assigned shards from each.

We assume that shuffle input is placed in M datacenters. The sizes of the partitions stored in these datacenters are $s_1, s_2, \ldots, s_M$, respectively. Without loss of generality, let the sizes be sorted in the non-ascending order, *i.e.,* $s_1 \geq s_2 \geq \ldots \geq s_M$. Also, each partition is divided into N shards, with respect to N reducers, $R_1, R_2, \ldots, R_N$. Though being different in sizes in practice, all shards of a particular partition tend to be about the same size for the sake of load balancing [9]. Thus, we assume the shards in a partition are equal in size.

If a reducer R_k is placed in Datacenter i_k, the total volume of its data fetched from non-local datacenters will be

$$d_{i_k}^{(k)} = \sum_{\substack{1 \leq j \leq M \\ j \neq i_k}} d_{i_k,j}^{(k)} = \sum_{\substack{1 \leq j \leq M \\ j \neq i_k}} \frac{1}{N} s_j.$$

Each term in the summation, $d_{i_k,j}^{(k)}$, denotes the size of data to be fetched from Datacenter j.

Let S be the total size of the shuffle input, *i.e.,* $S = \sum_{j=1}^{M} s_j$, we have

$$d_{i_k}^{(k)} = \sum_{j=1}^{M} \frac{1}{N} s_j - \frac{1}{N} s_{i_k} = \frac{1}{N}(S - s_{i_k}) \geq \frac{1}{N}(S - s_1). \quad (1)$$

The equality holds if and only if $i_k = 1$. In other words, the minimum cross-datacenter traffic can be achieved when the reducer is placed in the datacenter which stores the most shuffle input.

The inequality Eq. (1) holds for every reducer R_k ($k \in \{1, 2, \ldots, N\}$). Then, the total volume of cross-datacenter traffic incurred by this shuffle satisfies

$$D = \sum_{k=1}^{N} d_{i_k}^{(k)} \geq N \cdot \frac{1}{N}(S - s_1) = S - s_1. \quad (2)$$

Again, the equality holds iff. $i_1 = i_2 = \ldots = i_N = 1$.

Without any prior knowledge on application workflow, we reach two conclusions to optimize a general wide-area data analytic job.

First, given a shuffle input distribution, the datacenter with the largest fraction of shuffle input will be favored by the reducer placement. This is a direct corollary of Eq. (2).

Second, shuffle input should be aggregated to a subset of datacenters as much as possible. The minimum volume of data to be fetched across datacenters is $S - s_1$. Therefore, in order to further reduce cross-datacenter traffic in shuffle, we should improve $\frac{s_1}{S}$ which is the fraction of shuffle input placed in Datacenter 1. As an extreme case, if all shuffle input is aggregated in Datacenter 1, there is no need for cross-datacenter traffic in future stages.

In all, compared to scattered placement, a better placement decision would be *aggregating all shuffle input into a subset of datacenters which store the largest fractions*. Without loss of generality, in the subsequent sections of this paper, we will aggregate to a single datacenter as an example.

C. Summary and Discussion

According to the analysis throughout this section, we learn that the strategy of *Push/Aggregate*, *i.e.,* proactively pushing the shuffle input to be aggregated in a subset of worker datacenters, can be beneficial in wide-area data analytics. It can reduce both stage completion time and traffic tension, because of higher utilization of the inter-datacenter links. Also, duplicated inter-datacenter data transfers can be avoided in case of task failures, further reducing the pressure on the bottleneck links.

One may argue that even with the aggregated shuffle input, a good task placement decision is still required. Indeed, the problem itself sounds like a dilemma, where task placement and shuffle input placement depend on each other. However, with the above hints, we are able to break the dilemma by placing the shuffle input first. After that, a better task placement decision can be made by even the default resource schedulers, which has a simple straight-forward strategy to exploit host-level data locality.

Conceptually speaking, the ultimate goal of Push/Aggregate is to *proactively improve the best data locality* that a possible task placement decision can achieve. Thus, the Push/Aggregate operations are completely *orthogonal* and *complementary* to the task placement decisions.

IV. Implementation on Spark

In this section, we present our implementation of the Push/Aggregate mechanism on Apache Spark. We take Spark as an example in this paper due to its better performances [2] and better support for machine learning algorithms with MLlib [10]. However, the idea can be applied to Hadoop as well.

A. Overview

In order to implement the Push/Aggregate shuffle mechanism, we are required to modify two default behaviors in Spark: i) Spark should be allowed to directly *push* the output of an individual map task to a remote worker node, rather than storing on the local disk; and ii) the receivers of the output of map tasks should be selected automatically within the specific *aggregator datacenters*.

One may think a possible implementation would be replacing the default shuffle mechanism completely, by enabling remote disks, which locate in the aggregator datacenters, to be the potential storage in addition to the local

disk on a mapper. Though this approach is straight-forward and simple, there are two major issues.

On the one hand, although the aggregator datacenters are specified, it is hard for mappers to decide the exact destination worker nodes to place the map output. In Spark, it is the Task Scheduler's responsibility to make *centralized* decisions on task and data placement, considering both data locality and load balance among workers. However, a mapper by itself, without synchronization with the global Task Scheduler, can hardly have sufficient information to make the decision in a *distributed* manner, while still keeping the Spark cluster load-balanced. On the other hand, the push will not start until the entire map output is ready in the mapper memory, which introduces unnecessary buffering time.

Both problems are tough to solve, requiring undesirable changes to other Spark components such as the Task Scheduler. To tackle the former issues, it is natural to ask: *rather than implementing a new mechanism to select the storage of map output, is it possible to leave the decisions to the Task Scheduler?*

Because the Task Scheduler can have knowledge of computation tasks, we need to generate additional tasks in the aggregator datacenter, whose computation is as simple as receiving the output of mappers.

In this paper, we add a new transformation on RDDs, `transferTo()`, to achieve this goal. From a high level, `transferTo()` provides a means to explicitly transfer a dataset to be stored in a specified datacenters, while the host-level data placement decisions are made by the Spark framework itself for the sake of load balance. In addition, we implement an optional mechanism in Spark to automatically enforce `transferTo()` before a shuffle. This way, if this option is enabled, the developers are allowed to use the Push/Aggregate mechanism in all shuffles without changing a single line of code in their applications.

B. `transferTo()`*: Enforced Data Transfer in Spark*

`transferTo()` is implemented as a method of the base RDD class, the abstraction of datasets in Spark. It takes one optional parameter, which gives all worker hosts in the aggregator datacenter. However, in most cases, the parameter can be omitted, such that all data will be transferred to a datacenter that is *likely* to store the largest fraction of the parent RDD, as is suggested in Sec. III-C. It returns a new RDD, `TransferredRDD`, which represents the dataset after the transfer operation. Therefore, `transferTo()` can be used in the same way as other native RDD transformations, including chaining with other transformations.

When an analytic application is submitted, Spark will interpret the transformation by launching an additional set of receiver tasks, only to receive all data in the parent RDD. Then, the Task Scheduler can place them in the same manner as other computation tasks, thus to achieve automatic host-level load balance. Because the `preferredLocations` attributes of these receiver tasks are set to be in the aggregator datacenters, the default Task Scheduler will satisfy these placement requirements as long as the datacenters have workers available. This way, from the application's perspective, the parent RDD is thus explicitly *pushed* to the aggregator datacenters, without violating any default host-level scheduling policies.

Besides, a bonus point of `transferTo()` is that, since the receiver tasks require no shuffle from the parent RDD, they can be *pipelined* with the preceding computation tasks. In other words, if `transferTo()` is called upon the output dataset of a `map` task, the actual data transfer will start as soon as there is a fraction of data available, without waiting until the entire output dataset is ready. This pipelining feature is enabled by Spark without any further change, which automatically solves the second issue mentioned in Sec. IV-A.

It is worth noting that `transferTo()` can be directly used as a developer API. For developers, it provides the missing function that allows *explicit* data migration across worker nodes. `transferTo()` enjoy the following graceful features:

Non-Intrusiveness and Compatibility. The introduction of the new API modifies no original behavior of the Spark framework, maximizing the compatibility with existing Spark applications. In other words, changes made on the Spark codebase regarding `transferTo()` are completely incremental, rather than being intrusive. Thus, our patched version of Spark maintains 100% backward compatibility with the legacy code.

Consistency. The principle programming concept remains consistent. In Spark, RDD is the abstraction of datasets. The APIs allow developers to process a dataset by applying *transformations* on the corresponding RDD instance. The implementation of `transferTo()` inherits the same principle.

Minimum overhead. `transferTo()` strives to eliminate unnecessary overhead introduced by enforced data transfers. For example, if a partition of dataset already locates in our specified datacenter, no cross-node transfer is made. Also, unnecessary disk I/O is avoided.

C. Implementation Details of `tranferTo()`

As a framework for building big data analytic applications, Spark strives to serve the developers. By letting the framework itself make tons of miscellaneous decisions automatically, the developers are no longer burdened by the common problems in distributed computing, *e.g.,* communication and synchronization among nodes. Spark thus provides such a high-level abstraction that developers are allowed to program as if the cluster was a single machine.

An easier life comes at the price of less control. The details of distributed computing, including communications and data transfers among worker nodes, are completely hidden from the developers. In the implementation of

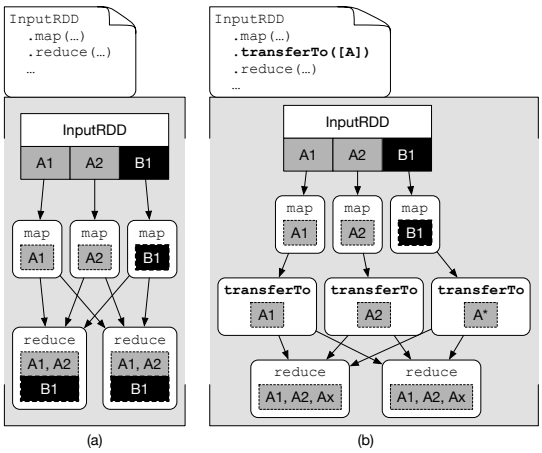

Figure 4: An example to show how `preferredLocations` attribute works without(a) or with(b) `transferTo()` transformation. `A*` represents all available hosts in datacenter A, while `Ax` represents the host which is selected as the storage of third map output partition.

`transferTo()` where we intend to explicitly specify the cross-node transfer of intermediate data, Spark does not expose such a functionality to the application developers.

Here we close this gap, by leveraging the internal `preferredLocations` attribute of an RDD.

1) `preferredLocations` *in Spark:* It is a native attribute in each partition of all RDDs, being used to specify the host-level data locality preferences. While the Task Scheduler is trying to place the corresponding computation on individual worker nodes, it plays an important role. In other words, the Task Scheduler takes `preferenceLocations` as a list of higher priority hosts, and strives to satisfy the placement preferences whenever possible.

A simple example is illustrated in Fig. 4 (a), where the input dataset is transformed by a `map()` and a `reduce()`. The input RDD has 3 partitions, located on two hosts in Datacenter A and one host in Datacenter B, respectively. Thus, 3 corresponding map tasks are generated, with `preferredLocations` the same as input data placement. Since the output of map tasks is stored locally, the `preferredLocations` of all reducers will be the union of the mapper hosts.

This way, the Task Scheduler can have enough hints to place tasks to maximize host-level data locality and minimize network traffic.

2) Specifying the Preferred Locations for `transferTo()` *Tasks:* In our implementation of `transferTo()`, we generate an additional computation task right after each map task, whose `preferredLocations` attribute filters out all hosts that are not in the aggregator datacenters.

Why do we launch new tasks, rather than directly changing the `preferredLocations` of mappers? The reason is simple: if mappers are directly placed in the aggregator datacenter, it will be the raw input data that is transferred across datacenters. In most cases, it is undesirable because map output is very likely to have a smaller size as compared to the raw input data.

If the parent mapper already locates in the aggregator datacenter, the generated task will do nothing; however, if not, thus the parent partition of map output requires being transferred, the corresponding task will provide a list of all worker nodes in the aggregator datacenter as the `preferredLocations`. In the latter case, the Task Scheduler will select one worker node from the list to place the task, which simply receives output from the corresponding mapper.

As another example, Fig. 4 (b) shows how `transferTo()` can impact the `preferredLocations` of all tasks in a simple job. As compared to Fig. 4 (a), the map output is explicitly transferred to Datacenter A. Because the first two partitions are already placed in Datacenter A, the two corresponding `transferTo()` tasks are completely transparent. On the other hand, since the third partition originated in Datacenter B, the subsequent `transferTo()` task should prefer any hosts in Datacenter A. As a result of task execution, the map output partition will be eventually transferred to a random host in Datacenter A, which is selected by the Task Scheduler. Finally, since all input of reducers is in Datacenter A, the shuffle can happen within a single datacenter, realizing the Push/Aggregate mechanism.

Note that we can omit the destination datacenter of `transferTo()`. If no parameter is provided, `transferTo()` will automatically decide the aggregator datacenter, by selecting the one with the most partitions of map output.

3) Optimized Transfers in the case of Map-side Combine: There is a special case in some transformations, *e.g.,* `reduceByKey()`, which require `MapSideCombine` before a shuffle. Strictly speaking, `MapSideCombine` is a part of reduce task, but it allows the output of map tasks to be combined on the mappers before being sent through network, in order to reduce the traffic.

In wide-area data analytics, it is critical to reduce cross-datacenter traffic for the sake of performance. Therefore, our implementation of `transferTo()` makes smart decisions, by performing `MapSideCombines` before transfer whenever possible. In `transferTo()`, we pipeline any `MapSideCombine` operations with the preceding map task, and avoid the repetitive computation on the receivers before writing the shuffle input to disks.

D. Automatic Push/Aggregate

Even though `transferTo()` is enough to serve as the fundamental building block of Push/Aggregate, a mechanism is required to enforce `transferTo()` automatically, without the explicit intervention from the application developers. To this end, we modified the default `DAGScheduler` component in Spark, to add an optional feature that automatically inserts `transferTo()` before all potential shuffles in the application.

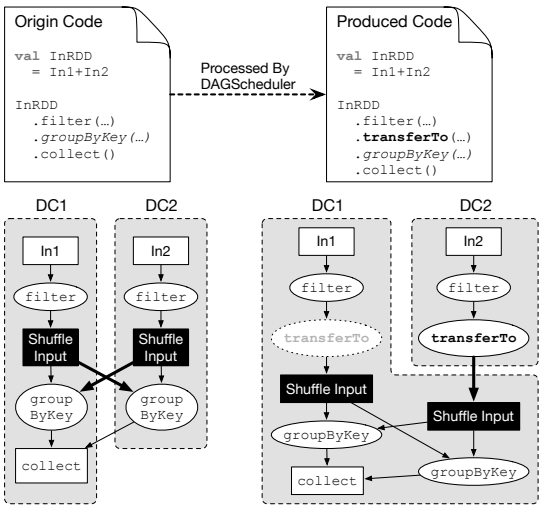

Figure 5: An example of implicit embedding of the `transferTo()` transformation. `transferTo()` aggregates all shuffle input in DC1, before the `groupByKey()` transformation starts. For the partition natively stored in DC1, `transferTo()` simply does nothing.

The programmers can enable this feature by setting a property option, `spark.shuffle.aggregation`, to `true` in their Spark cluster configuration file or in their code. We did not enable this feature by default for backward compatibility considerations. Once enabled, the `transferTo()` method will be embedded *implicitly* and automatically to the code before each shuffle, such that the shuffle inputs can be pushed to the aggregator datacenters.

Specifically, when a data analytic job is submitted, we use `DAGScheduler` to embed the necessary `transferTo()` transformations into the origin submitted code. In Spark, `DAGScheduler` is responsible for rebuilding the entire workflow of a job based on consecutive RDD transformations. Also, it decomposes the data analytic job into several shuffle-separated stages.

Since `DAGScheduler` natively identifies all data shuffles, we propose to add a `transferTo()` transformation ahead of each shuffle, such that the shuffle input can be aggregated. Fig. 5 illustrates an example of implicit `transferTo()` embedding. Since `groupByKey()` triggers a shuffle, the `transferTo()` transformation is embedded automatically right before that to start proactive transfers of the shuffle input.

Note that because `transferTo()` is inserted automatically, none parameter is provided to the method. Therefore, it works the same way as if the aggregator datacenter is omitted, *i.e.,* the datacenter that generates the largest fraction of shuffle input should be chosen. We approximate the optimal selection by choosing the datacenter storing the largest amount of map input, which is a known piece of information in `MapOutputTracker` at the beginning of the map task.

E. Discussion

In addition to the basic feature that enforces aggregation of shuffle input, the implementation of `transferTo()` can trigger interesting discussions in wide-area data analytics.

Reliability of Proactive Data Transfers. One possible concern is that, since the push mechanism for shuffle input is a new feature in Spark, the reliability of computation, *e.g.,* fault tolerance, might be compromised. However, it is not true.

Because `transferTo()` is implemented by creating additional receiver tasks rather than changing any internal implementations, all native features provided by Spark are inherited. The introduced proactive data transfers, from the Spark framework's perspective, are the same as regular data exchanges between a pair of worker nodes. Therefore, in case of failure, built-in recovery mechanisms, such as retries or relaunches, will be triggered automatically in the same manner.

Expressing Cross-region Data Transfers as Computation. Essentially, `transferTo()` provides a new interpretation for inter-datacenter transfers. In particular, they can be expressed in a form of computation, since `transferTo()` is implemented as a transformation. It conforms with our intuition, in which moving a large volume of data across datacenters consumes both computation and network resources that are comparable to a normal computing task.

This concept can help in several ways. For example, inter-datacenter data transfers can be shown from the Spark WebUI. It can be helpful in terms of debugging the wide-area data analytic jobs, by visualizing the critical inter-datacenter traffic.

Implicit vs. Explicit Embedding. Instead of implicit embedding `transferTo()` using `DAGScheduler`, the developers are allowed to explicitly control the data placement at the granularity of datacenters. In some real-world data analytic applications, this is meaningful because the developers always know better about their data.

For example, it is possible in production that the shuffle input has a larger size than the raw data. In this case, to minimize inter-datacenter traffic, it is the raw data rather than the shuffle input that should be aggregated. The developers can be fully aware of this situation; however, it is difficult for the Spark framework itself to make this call, resulting in an unnecessary waste of bandwidth.

Another example is the cached datasets. In Spark, the developers are allowed to call `cache()` on any intermediate RDD, in order to persist the represented dataset in memory. These cached datasets will not be garbage collected until the application exits. In practice, the intermediate datasets that will be used several times in an application should be cached to avoid repetitive computation. In wide-area data analytics, caching these datasets across multiple datacenters is extremely expensive, since reusing them will induce repetitive inter-datacenter traffic. Fortunately, with the help

Workload	Specification
WordCount	The total size of generated input files is 3.2 GB.
Sort	The total size of generated input data is 320 MB.
TeraSort	The input has 32 million records. Each record is 100 bytes in size.
PageRank	The input has 500,000 pages. The maximum number of iterations is 3.
NaiveBayes	The input has 100,000 pages, with 100 classes.

Table I: The specifications of four workloads used in the evaluation.

of `transferTo()`, the developers are allowed to cache after all data is aggregated in a single datacenter, avoiding the duplicated cross-datacenter traffic.

Limitations. Even though the Push/Aggregate shuffle enjoys many graceful features, it does have limitations that users should be aware of. The effectiveness of `transferTo()` relies on the sufficient computation resources in the aggregator datacenter. It will launch additional tasks in the aggregator datacenters, in which more computation resources will be consumed. If the chosen aggregator datacenter cannot complete all reduce tasks because of insufficient resources, the reducers will be eventually placed in other datacenters, which would be less effective.

We think this limitation is acceptable in wide-area data analytics for two reasons. On the one hand, Push/Aggregate basically trades more computation resources for lower job completion times and less cross-datacenter traffic. Because the cross-datacenter network resources are the bottleneck in wide-area data analytics, the trade-off is reasonable. On the other hand, in practice, it is common that a Spark cluster is shared by multiple jobs, such that the available resources within one datacenter is more than enough for a single job. Besides, when the cluster is multiplexed by many concurrent jobs, it is very likely that the workload can be rebalanced across-datacenters, keeping the utilization high.

V. Experimental Evaluation

In this section, we present a comprehensive evaluation of our proposed implementation. Our experiments are deployed across multiple Amazon Elastic Compute Cloud (EC2) regions. Selective workloads from HiBench [7] are used as benchmarks to evaluate both the job-level and the stage-level performances.

The highlights of our evaluation results are as follows:

1) Our implementation speeds up workloads from the HiBench benchmark suite, reducing the average job completion time by 14% $\sim$ 73%.
2) The performances are more predictive and stable, despite the bandwidth jitters on inter-datacenter links.
3) The volume of cross-datacenter traffic can be reduced by about 16% $\sim$ 90%.

A. Cluster Configurations

Amazon EC2 is one of the most popular cloud service providers today. It provides computing resources that are hosted in their datacenters around the globe. Since EC2 is a production environment for a great number of big data analytic applications, we decide to run our experimental evaluation by leasing instances across regions.

Figure 6: The geographical location and the number of instances in our Spark cluster. Instances in 6 different Amazon EC2 regions are employed. Each region has 4 instances running, except N. Virginia where two extra special nodes deployed.

Cluster Resources. We set up a Spark cluster with 26 nodes in total, spanning 6 different geographically distributed regions on different continents, as is shown in Fig. 6. Four worker nodes are leased in each datacenter. The Spark master node and the HaDoop File System (HDFS) NameNode are deployed on 2 dedicated instances in the N. Virginia region, respectively.

All instances in use are of the type `m3.large`, which has 2 vCPUs, 7.5 GB of memory, and a 32 GB Solid-State Drive (SSD) as disk storage. The network performance of the instances is reported as "moderate". Our measurement shows that there is approximately 1 Gbps of bandwidth capacity between a pair of instances within a region. However, the cross-region network capacity varies over time. Our preliminary investigation is consistent with previous empirical studies [8], [11]. The available bandwidth of inter-datacenter links fluctuates greatly. Some links can have as low as 80 Mbps of capacity, while other links may have up to 300 Mbps bandwidth.

Software Settings. The instances in our cluster are running a Linux Operating System, Ubuntu 14.04 LTS 64-bit (HVM). To set up a distributed file system, we use HDFS from Apache Hadoop 2.6.4. Our implementation is developed based on Apache Spark 1.6.1, built with Java 1.8 and Scala 2.11.8. Spark cluster is started in the standalone mode, without the intervention of external resource managers. This way, we leave the Spark's internal data locality mechanism to make the task placement decisions in a coarse-grained and greedy manner.

Workload Specifications. Within the cluster, we run five selected workloads of the HiBench benchmark suite, `WordCount`, `Sort`, `TeraSort`, `PageRank`, and `NaiveBayes`. These workloads are good candidates for testing the efficiency of the data analytic frameworks, with an increasing complexity. Among the workloads, `WordCount` is the sim-

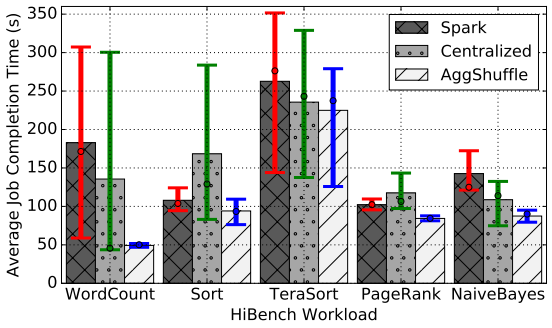

Figure 7: The average job completion time under different HiBench workload. For each workload, we present a 10% trimmed mean over 10 runs, with an error bar representing the interquartile range as well as the median value.

plest, involving one shuffle only. `PageRank` and `NaiveBayes` are relatively more complex and require several iterations at runtime, with multiple consecutive shuffles. They are two representative workloads of the machine learning algorithms. The workloads are configured to run at "large scale," which is one of the default options in HiBench. The specifications of their settings are listed in Table I. The maximum parallelism of both map and reduce is set to 8, as there are 8 cores available within each datacenter.

Baselines. We use two naive solutions, referred as "*Spark*" and "*Centralized*", in wide-area data analytics as the baselines to compare with our proposed shuffle mechanism. *"Spark"* represents the deployment of Spark across geo-distributed datacenters, without any optimization in terms of the wide-area network. The job execution will be completely blind about the network bottleneck. The *"Centralized"* scheme refers to the naive and greedy solution in which all raw data is sent to a single datacenter before being processed. After all data is centralized within a cluster, Spark works within a datacenter to process data.

As a comparison, the Spark patched with our proposed shuffle mechanism is referred to as "*AggShuffle*" in the remainder of this section, meaning the shuffle input is aggregated in a single datacenter. Note that we *do not* use implicitly embedded `transferTo()` transformation. Only are the implicit transformations involved in the experiments, leaving the benchmark source code unchanged.

B. Job Completion Time

The completion time of a data analytic job is the primary performance metric. Here, we report the measurement results from HiBench, which records the duration of running each workload. With 10 iterative runs on the 5 different workloads, the mean and the distribution of completion times are depicted in Fig. 7. Note that running Spark applications across EC2 regions is prone to the unpredictable network performances, as the available bandwidth and network latency fluctuates dramatically over time. As a result, running the same workload with the same execution plan at different times may result in distinct performances. To eliminate the incurred randomness as much as possible, we introduce the following statistical methods to process the data.

Trimmed average of the job completion time. The bars in Fig. 7 reports the 10% trimmed mean value of job completion time measurements over 10 runs. In particular, the maximum and the minimum values are invalidated before we compute the average. This methodology, in a sense, eliminates the impact of its long-tail distribution on the mean.

According to Fig. 7, *AggShuffle* offers the best performances in all three schemes in evaluation. For example, *AggShuffle* shows as much as 73% and 63% reduction in job completion time, as compared to *Spark* and *Centralized*, respectively. Under other workloads, using *Spark* as the baseline, our mechanism achieves at least 15% performance gain in terms of job durations.

As compared to the *Centralized* mechanism, we can easily find that *AggShuffle* is still beneficial, except `TeraSort`. Under the TeraSort workload, the average job completion time in *Centralized* is only 4% higher, which is a minor improvement in practice. The reason is hidden behind the `TeraSort` algorithm. In the HiBench implementations, there is a map transformation before all shuffles, which actually bloats the input data size. In other words, the input of the first shuffle is even larger in size as compared to the raw input. Consequently, extra data will be transferred to the destination datacenter, incurring unnecessary overhead. Looking ahead, the analysis is supported by the cross-datacenter traffic measurement shown in Fig. 8. `TeraSort` turns out to be a perfect example to show the necessity of developers' interventions. Only can the application developers tell the increase of data size beforehand. This problem can be resolved by *explicitly* calling `transferTo()` before the map, and we can expect further improvement from *AggShuffle*.

Interquartile range and the median of the job completion time. In addition to the average job completion time, we think the distribution of durations of the same job matters in wide-area data analytics. To provide further the distribution information in Fig. 7, according to our measurements over 10 iterative runs. To this end, we add the interquartile range and the median into the figure as error bars. The interquartile range shows the range from the 25-th percentile and the 75-th percentile in distribution. Also, the median value is shown as a dot in the middle of an error bar.

Fig. 7 clearly shows that *AggShuffle* outperforms both other schemes in terms of minimizing the variance. In other words, it can provide wide-area data analytic applications with more stable performances, making it more predictive. It is an important feature, as is suggested in the experimental results, even running in the same environment settings, the completion time of a wide-area analytic job varies significantly over time. We argue the ability to limit the variance

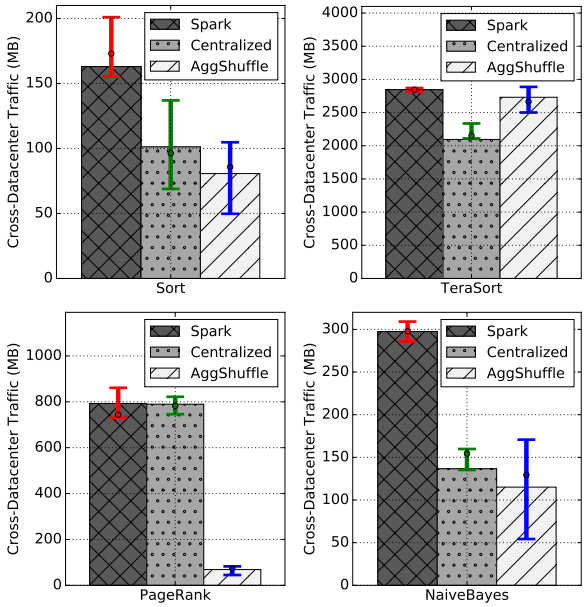

Figure 8: Total volume of cross-datacenter traffic under different workloads.

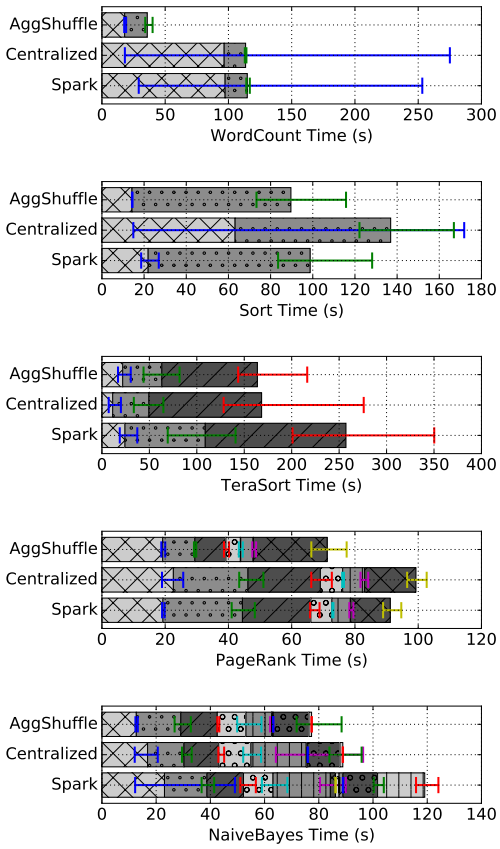

Figure 9: Stage execution time breakdown under each workload. In the graph, each segment in the stacked bars represents the life span of a stage.

of data analytics frameworks is a performance metric that has been overlooked in the literature.

The reason of *AggShuffle*'s stability is two-fold. On the one hand, the major source of performance fluctuation is the network performance. As the wide-area links interconnecting datacenters, unlike the datacenter network, are highly unstable with no performance guarantees. Flash congestion and temporarily connections lost are common, whose impact will be magnified in the job completion times. On the other hand, since *AggShuffle* initiates early data transfers without waiting for the reducers to start. This way, concurrent bulk traffic on bottleneck links will be smoothed over time, with less link sharing and a better chance for data transfer to complete quickly.

As for *TeraSort*, rather than offering help, our proposed aggregation of shuffle input actually burdens the cross-datacenter network. Again, it can be resolved by explicitly invoking `transferTo()` for optimality.

C. Cross-Region Traffic

The volume of cross-datacenter traffic incurred by wide-area analytic applications is another effective metric for evaluation. During our experiments on EC2, we tracked the cross-datacenter traffic among Spark worker nodes. The average of our measurement is shown in Fig. 8. Note in this figure, the "Centralized" scheme indicates the cross-region traffic to aggregate all data into the centralized datacenter.

Except for `TeraSort`, whose `transferTo()` is automatically called on a bloated dataset, all other workloads in the evaluation can enjoy much less bandwidth usage in `AggShuffle`. As shuffle input is proactively aggregated in early stages and all further computation is likely to be scheduled within one datacenter, cross-datacenter traffic will be reduced significantly on average. In particular, it is worth noting that under the `PageRank` workload, the required cross-datacenter traffic can be reduced by 91.3%, which is pretty impressive.

Fig. 8 shows that the "Centralized" scheme requires the least cross-datacenter traffic in `TeraSort` among the three. It is consistent with the conclusion in our previous discussions.

D. Stage Execution Time

In Fig. 9, we breakdown the execution of different workloads by putting them under the microscope. Specifically, we show the detailed average stage completion time in our evaluation. Particularly, the length of stacked bars represents the trimmed average execution time for each stage under specific workloads. Again, the error bars read the interquartile regions and median values.

Inferred from the large variances, any stages in *Spark* may suffer from degraded performances, most likely due to poor data locality. As a comparison, the *Collective* strategy usually performs well in late stages, while has the longest average completion time in early stages. It is supposed to be the result of collecting all raw data in early stages. However,

AggShuffle can finish both early and late stages quickly. Similar to the *Collective* scheme, it offers an exceptionally low variance in the completion time of late stages.

Although different stages under different workloads have specific features and patterns, we are still able to provide some useful insights. The "magic" behind *AggShuffle* is that it proactively improves the data locality during shuffle phases, without the need to transfer excessive data. Then, as shuffle input is aggregated in a smaller number of datacenters, the achievable data locality is high enough to guarantee a better performance.

Note that in Fig. 9, the total completion time of all stages is not necessarily equivalent to the job completion time presented in Fig. 7. First of all, though being stacked together in Fig. 9, some of the stages may overlap with each other at runtime. The summation does not directly contribute to the total job completion time. Second, stage completion time is measured and reported by Spark, while the measurement of job completion time is implemented by HiBench, with different concepts. Third, the cross-stage delays such as scheduling and queuing are not covered by the stage completion time measurements.

VI. Related Work

Wide-area Data Analytics. Running analytic jobs whose input data originates from multiple geographically distributed datacenters is commonly known as *wide area data analytics* in the research literature. Recent work on wide area data analytics focuses mostly on task placement.

Geode [4], WANalytics [5] and Pixida [3] propose task assignment strategies aiming at reducing the total volume of traffic among datacenters. Iridium [6], on the other hand, argues that less cross-datacenter traffic does not necessarily result in shorter job completion time, which is a better metric for analytic job performance. Thus, they propose an online heuristic to make joint decisions on both input data and task placement across datacenters. Our work, as is mentioned in previous sections (Sec. III-C), is complementary to these proposals on task placement. We focus on the placement of shuffle input, the intermediate data, to help task placement algorithms achieve even better performances.

Heintz *et al.* [12] propose an algorithm to produce an entire job execution plan, including both data and task placement, for geo-distributed MapReduce. However, their model requires too much prior knowledge such as intermediate data sizes, which makes it far from practical. Other existing work in the literature tries to solve a different problem in wide-area data analytics, which is beyond the scope of this paper. For example, Hung *et al.* [13] propose a greedy scheduling heuristic to schedule multiple concurrent analytic jobs, with an objective of reducing the average job completion time. Instead of caring about the cross-job performance, we aim to optimize the execution of a single job. There is work on wide-area streaming analytics [14], [15] and SQL queries [16] in the literature, which is a different story from the batch MapReduce jobs that we are focusing on.

Shuffle Input Placement in MapReduce. When it comes to running MapReduce jobs within a datacenter, there exist several proposals on optimizing shuffle input data placement. iShuffle [17] proposes to shuffle before the reducer tasks are launched in Hadoop. Shards in the shuffle input are pushed to the predicted reducers, respectively, during shuffle write, that is, a "shuffle-on-write" service. However, it is not practical to predict the reducer placement before hand, especially in Spark. MapReduce Online [18] also proposes a push-based shuffle mechanism, in order to optimize the performance under continuous queries. Unfortunately, general analytic jobs that are submitted randomly will not benefit.

VII. Concluding Remarks

In this paper, we have designed and implemented a new system framework that optimizes network transfers in the shuffle stages of wide-area data analytic jobs. The gist of our new framework lies in the design philosophy that the output data from mapper tasks should be proactively aggregated to a subset of datacenters, rather than passively fetched as they are needed by reducer tasks in the shuffle stage. The upshot of such proactive aggregation is that data transfers can be pipelined and started as soon as computation finishes, and do not need to be repeated when reducer tasks fail. The core of our new framework is a simple `transferTo` transformation on Spark RDDs, which allows it to be implicitly embedded by the Spark DAG scheduler, or explicitly added by application developers. Our extensive experimental evaluations with the HiBench benchmark suite on Amazon EC2 have clearly demonstrated the effectiveness of our new framework, which is complementary to existing task assignment algorithms in the literature.

Acknowledgments

This work is supported by a research contract with Huawei Corp. and an NSERC Collaborative Research and Development (CRD) grant. S. Liu is partially supported by the China Scholarship Council.

References

[1] "Apache hadoop official website." http://hadoop.apache.org/, [Online; accessed 1-May-2016].

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[3] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics," *VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.

[4] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global Analytics in the Face of Bandwidth and Regulatory Constraints," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[5] A. Vulimiri, C. Curino, P. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for A Geo-Distributed Data-Intensive World," in *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2015.

[6] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low Latency Geo-Distributed Data Analytics," in *Proc. ACM SIGCOMM*, 2015.

[7] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," in *Proc. International Conference on Data Engineering Workshops (ICDEW)*, 2010.

[8] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling Tasks Closer to Data Across Geo-Distributed Datacenters," in *Proc. IEEE INFOCOM*, 2016.

[9] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing Reducer Skew in MapReduce Workloads Using Progressive Sampling," in *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2012.

[10] "MLlib: apache spark website." http://spark.apache.org/mllib/, [Online; accessed 1-May-2016].

[11] Z. Liu, Y. Feng, and B. Li, "Bellini: Ferrying application traffic flows through geo-distributed datacenters in the cloud," in *Proc. IEEE Globecom*, 2013.

[12] B. Heintz, A. Chandra, R. Sitaraman, and J. Weissman, "End-to-End Optimization for Geo-Distributed MapReduce," *IEEE Transactions on Cloud Computing*, 2015.

[13] C. Hung, L. Golubchik, and M. Yu, "Scheduling Jobs Across Geo-Distributed Datacenters," in *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2015.

[14] B. Heintz, A. Chandra, and R. Sitaraman, "Optimizing Grouped Aggregation in Geo-Distributed Streaming Analytics," in *Proc. ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2015.

[15] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. Freedman, "Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[16] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: Wan-aware optimization for analytics queries," in *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[17] Y. Guo, J. Rao, and X. Zhou, "iShuffle: Improving Hadoop Performance with Shuffle-on-Write," in *Proc. USENIX International Conference on Autonomic Computing (ICAC)*, 2013.

[18] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *Proc. USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.