End-to-End Optimization for Geo-Distributed MapReduce

Benjamin Heintz, *Student Member, IEEE*, Abhishek Chandra, *Member, IEEE*, Ramesh K. Sitaraman, *Member, IEEE*, and Jon Weissman, *Senior Member, IEEE*

Abstract—MapReduce has proven remarkably effective for a wide variety of data-intensive applications, but it was designed to run on large single-site homogeneous clusters. Researchers have begun to explore the extent to which the original MapReduce assumptions can be relaxed, including skewed workloads, iterative applications, and heterogeneous computing environments. This paper continues this exploration by applying MapReduce across geo-distributed data over geo-distributed computation resources. Using Hadoop, we show that network and node heterogeneity and the lack of data locality lead to poor performance, because the interaction of MapReduce phases becomes pronounced in the presence of heterogeneous network behavior. To address these problems, we take a two-pronged approach: We first develop a *model-driven optimization* that serves as an oracle, providing high-level insights. We then apply these insights to design *cross-phase* optimization techniques that we implement and demonstrate in a real-world MapReduce implementation. Experimental results in both Amazon EC2 and PlanetLab show the potential of these techniques as performance is improved by 7-18 percent depending on the execution environment and application.

Index Terms—Batch processing systems, distributed systems, parallel systems

1 Introduction

Recent years have seen increasing amounts of data generated and stored in a geographically distributed manner for a large variety of application domains. Examples include social networking, Web and Internet service providers, and content delivery networks (CDNs) that serve the content for many of these services. For instance, Facebook has more than one billion users, more than 80 percent of whom are outside the US or Canada,[1] and Google has developed storage systems [1], [2] to manage data partitioned across globally distributed data centers.

As another example, consider a large content delivery network such as Akamai, which uses over 100,000 servers deployed in over 1,000 locations to serve 15-30 percent of the global web traffic [3]. Content providers use CDNs to deliver web content, live and on-demand videos, downloads, and web applications to users around the world. The servers of a CDN, deployed in clusters in hundreds or thousands of geo-distributed data centers, each log detailed data about each user they serve. In aggregate, the servers produce tens of billions of lines of geo-distributed log data every day.

Many modern applications need to *efficiently* process such *geo-distributed data* on a *geo-distributed platform*. As an example, a CDN analytics application must extract detailed information about who is accessing the content, from which networks and from which geographies, as well as the quality of experience for users, including page download speeds, video startup times, and application transaction times. This processing must complete quickly so that content providers can understand and act upon these key indicators.

A critical question for efficient processing of such distributed data is *where* to carry out the computation. As Jim Gray noted, "you can either move your questions or the data" [4]. These two options, however, represent two extreme possibilities. At one extreme, sending massive data from its diverse origin locations to a centralized data center may lead to excessive latency [5], [6]. Further, the bandwidth cost may be prohibitive and it may be infeasible to fit all of the data in a single central data center. A centralized approach is also fundamentally less fault-tolerant than a distributed one; failure of the single centralized data center can cause a complete outage.

At the other extreme, if we map computation onto each input datum *in situ*, the results of these subcomputations comprise intermediate data that must be aggregated to generate final results. If such intermediate data are large, then aggregating them may be more costly than moving input data to a centralized location in the first place. Further, the various locations may differ in their compute capacities, leading to imbalanced resource utilization and task execution times.

In this paper, we use MapReduce as a vehicle for exploration, and show that between these two extremes lies a spectrum of possibilities that are often more efficient. Because MapReduce was originally designed [7] for the relatively homogeneous single-datacenter setting, however, it is natural to ask whether it is well suited for environments where both data and compute resources are

1. http://newsroom.fb.com

- B. Heintz, A. Chandra, and J. Weissman are with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455. E-mail: {heintz, chandra, jon}@cs.umn.edu.
- R.K. Sitaraman is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003, and Akamai Technologies. E-mail: ramesh@cs.umass.edu.

Manuscript received 13 Nov. 2013; revised 6 Mar. 2014; accepted 25 Aug. 2014. Date of publication 5 Sept. 2014; date of current version 7 Sept. 2016. Recommended for acceptance by G. Agrawal.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2014.2355225

2168-7161 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

geo-distributed. Indeed, as our prior work has shown [8], the Hadoop MapReduce implementation often performs very poorly in geo-distributed environments with high network heterogeneity. We find that the main reason for this poor performance is the heavy dependency across different MapReduce phases. This happens because the data placement and task execution are tightly coupled in MapReduce, as tasks are usually assigned to nodes that already host their input data. As a result, the choice of mapper nodes to which inputs are pushed impacts both how long the data push takes, as well as where the intermediate data are generated. This in turn impacts the performance of the data shuffle to the reducers. This problem is particularly severe for wide-area environments, where heterogeneity of both node and link capacities prevails. Therefore, it is important to optimize the end-to-end computation as a whole while taking into account the platform and application characteristics.

In spite of the limitations of popular MapReduce implementations, however, the MapReduce *abstraction* is remarkably powerful, and implementing it in geo-distributed settings is a worthy objective. MapReduce has been applied to a surprising variety of data-intensive computing applications, and many data scientists have gained MapReduce application development expertise. Further, a rich ecosystem has been built on top of MapReduce, or more specifically the open-source Hadoop [9] implementation. We see promise in expanding the reach of this ecosystem and expertise into geo-distributed environments.

1.1 Research Contributions

In this paper, we take a two-pronged approach, first focusing on a model-driven optimization that serves as an oracle, and then applying the high-level insights from this oracle to develop techniques that we apply in a real-world MapReduce implementation. We make the following research contributions.

1.1.1 Model-Driven Optimization

We develop a framework for modeling the performance of MapReduce in geo-distributed settings (Section 3). We use this model to formulate an optimization problem whose solution is an *optimal execution plan* describing the best placement of data and computation within a MapReduce job. Our optimization minimizes *end-to-end* makespan and controls multiple phases of execution, unlike existing approaches which may optimize in a *myopic* manner or control only a single phase. Further, optimizations using our model are efficiently solvable in practice as mixed integer programs (MIP) using powerful solver libraries.

Second, we modify Hadoop to implement our optimization outputs and use this modified Hadoop implementation along with network and node measurements from the PlanetLab [10] testbed to validate our model.

Third, we evaluate our model-driven optimization and demonstrate that its end-to-end objective and multi-phase control both contribute significantly to improving task placement in geo-distributed settings. Concretely, our model results show that for a geo-distributed compute environment, our end-to-end, multi-phase optimization can provide nearly 82 and 64 percent reduction in execution time over myopic and the best single-phase optimizations, respectively.

1.1.2 Systems Implementation

We apply the results from our model-driven optimization toward a practical implementation in Hadoop. The key idea behind our techniques is to consider not only the execution cost of an individual task or computational phase, but also its impact on the performance of subsequent phases. We develop two techniques in particular.

- *Map-aware Push (Section 5).* We propose making the data push aware of the cost of map execution, based on the source-to-mapper link capacities as well as mapper node computation speeds. We achieve this by overlapping the data push with map execution, which provides us with two benefits. The first benefit is a pipelining effect which hides the latency of data push with the map execution. The second benefit is a dynamic feedback between the map and push that enables nodes with higher speeds and faster links to process more data at runtime.
- *Shuffle-aware Map (Section 6).* In MapReduce, the shuffling of intermediate data from mappers to reducers is an all-to-all operation. In a heterogeneous environment, a mapper with a slow outgoing link can therefore become a bottleneck in the shuffle phase, slowing down the downstream reducers. We propose map task scheduling based on the estimated shuffle cost from each mapper to enable faster shuffle and reduce.

We implement these techniques in the Hadoop framework, and evaluate their benefits on both Amazon EC2 and PlanetLab (Section 7). Experimental results show the potential of these techniques, as performance is improved by 7-18 percent depending on the execution environment and application characteristics.

2 A MapReduce Optimization Example

A typical MapReduce job executed in a cluster environment comprises three main phases: (i) *Map*, where map tasks execute on their input data; (ii) *Shuffle*, where the output of map tasks (intermediate key-value pairs) are disseminated to reduce tasks; and (iii) *Reduce*, where reduce tasks are executed on the intermediate data to produce the final outputs. It is typically assumed that the input data are already available on the compute nodes before the job execution begins. Such *data push* is usually achieved through file system mechanisms such as those provided by HDFS. In geo-distributed settings, however, the process of pushing data from sources to compute nodes may itself be costly, and therefore must be considered as a separate phase of the overall computation.

Before delving into the details of our research contributions, we further illustrate the challenges of task placement in geo-distributed MapReduce with a simple example, demonstrating that the best placement depends on both platform and application characteristics.

Consider the example MapReduce platform shown in Fig. 1. Assume that the data sources S_1 and S_2 have 150 and

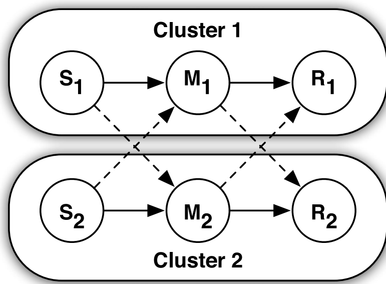


Fig. 1. An example of a two-cluster distributed environment, with one data source, one mapper, and one reducer in each cluster. The intra-cluster “local” communication links are solid lines, while the inter-cluster “non-local” communication links are the dashed lines.

50 GB of data, respectively. Let us model a parameter α that represents the ratio of the amount of data output by a mapper to its input; i.e., α is the expansion (or reduction) of the data in the map phase and it is specific to the application.

First, consider a situation where $\alpha = 1$ and the network is perfectly homogeneous. That is, all links have the same bandwidth—say 100 MBps each—whether they are local or non-local. Assume also that the computational resources at each mapper and reducer are homogeneous; say, each can process data at the rate of 100 MBps. Clearly, in this case, a uniform data placement, where each data source (resp., mapper) pushes (resp., shuffles) an equal amount of data to each mapper (resp., reducer), produces the minimum execution time (formally, *makespan*).

Now, consider a slight modification to the above scenario, where non-local links are much slower, transmitting only at 10 MBps, while all other parameters remain unchanged. A uniform data placement no longer produces the best makespan. Since the non-local links are much slower, it makes sense to avoid non-local links when possible. In fact, a plan where each data source pushes all of its data to its own local mapper completes the push phase in $150\,\text{GB}/100\,\text{MBps} = 1{,}500$ seconds, while the uniform push would have taken $75\,\text{GB}/10\,\text{MBps} = 7{,}500$ seconds. Although the map phase for the uniform placement would be smaller by $50\,\text{GB}/100\,\text{MBps} = 500$ seconds, the local push approach makes up for this through a more efficient data push.

Finally, consider the same network parameters (100 MBps local links and 10 MBps non-local links), but now with $\alpha = 10$, implying that there is 10 times more data in the shuffle phase than in the push phase. The local push no longer performs well, since it does not alleviate the communication bottleneck in the shuffle phase. To avoid non-local communication in the shuffle phase, it makes sense for data source S_2 to push all of its 50 GB of data to mapper M_1 so that all the reduce can happen within cluster 1 without having to use non-local links in the communication-heavy shuffle phase.

From this example, we can see that an effective optimization must control *multiple* phases with an *end-to-end* objective to minimize makespan.

3 Model-Driven Optimization

Our model-driven optimization framework allows us to explore the spectrum between purely distributed and purely centralized computation. In particular, we focus on how to find the best location along this spectrum assuming *perfect knowledge* under *static conditions*. In a real-world MapReduce implementation, the applicability of such a model may be limited by the need to gather all inputs ahead of time. For example, the number of network links to measure may be large, the computation time required by a new application may not be known *a priori*, or conditions may vary over time. In spite of these practical limitations, however, such a model is highly valuable, as it provides insights that inspire the design of pragmatic optimization mechanisms, as we demonstrate later in this paper (Sections 5, 6, and 7).

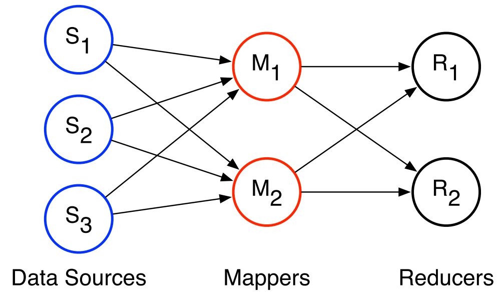


Fig. 2. A tripartite graph model for geo-distributed MapReduce with three data sources, two mappers, and two reducers.

3.1 Model

We present our model in terms of the distributed platform, the MapReduce application, and the placement of data and computation (formally, the *execution plan*).

3.1.1 Distributed Platform

We model the distributed platform available for executing the MapReduce application as a tripartite graph with vertex set $V = S \cup M \cup R$, where S is the set of data sources, M is the set of mappers, and R is the set of reducers (see Fig. 2). The edge set E is the complete set of edges, $(S \times M) \cup (M \times R)$.

Each *node* represents a physical resource: either a data source providing inputs, or a computation resource available for executing map or reduce processes. A node can therefore represent a single physical machine or even a single map or reduce *slot* in a small Hadoop cluster, or it can represent an entire rack, cluster, or data center of machines in a much larger deployment. Each edge represents the communication link connecting a pair of such nodes. The *capacity of a node* $i \in M \cup R$, denoted by C_i, captures the computational resources available at that node in units of bits of incoming data that it can process per second. Note that C_i is also application-dependent as different MapReduce applications require different amounts of computing resources to process the same amount of data. Likewise, the *capacity of an edge* $(i, j) \in E$, denoted by B_{ij}, represents the bandwidth (in bits/second) that can be sustained on the communication link that the edge represents.

3.1.2 MapReduce Application

Application characteristics are captured by two key parameters: the *amount of data* D_i (in bits) originating at data source i, for each $i \in S$; and the *expansion factor* α that

represents the ratio of the size of the output of the map phase to the size of its input. Note that α can take values less than, greater than, or equal to 1, depending on whether the output of the map operation is smaller than, larger than, or equal in size to the input, respectively. Many applications perform extensive aggregation in the map phase, for example by filtering records according to a predicate, or by projecting only a subset of fields from complex records. These applications have α much less than 1. On the other hand, some applications augment the input data in the map phase (e.g., relational join), or they emit multiple copies of intermediate records (e.g., to compute an aggregate at city, state, and national levels), yielding $\alpha > 1$. This parameter has a strong impact on optimal placement decisions, as observed in our prior work [8]. The value of α can be determined by profiling the MapReduce application on a sample of inputs [11], [12].

3.1.3 Execution Plans

We define the notion of an *execution plan* to capture the manner in which data and computation of a MapReduce application are scheduled on a distributed platform. Intuitively, an execution plan determines how the input data are distributed from the sources to the mappers and how the intermediate key-value pairs produced by the mappers are distributed across the reducers. Thus, the execution plan determines which nodes and which communication links are used and to what degree. An execution plan is represented by variables x_{ij}, for $(i, j) \in E$, that represent the fraction of the outgoing data from node i that is sent to node j.

We now mathematically express sufficient conditions for an execution plan to be *valid* in a MapReduce implementation while obeying the MapReduce application semantics. First, for each i, the x_{ij} values must be fractions that sum to 1, as enforced by (1) and (2). Second, the semantics of MapReduce requires that each intermediate key be shuffled to a single reducer, which we enforce with (3), where y_k may be interpreted as the fraction of the intermediate key-value data reduced at reducer k.

$$\forall (i, j) \in E\colon 0 \leq x_{ij} \leq 1, \tag{1}$$

$$\forall i \in V\colon \sum_{(i,j)\in E} x_{ij} = 1, \tag{2}$$

$$\forall j \in M, k \in R\colon x_{jk} = y_k. \tag{3}$$

We define an execution plan to be *valid* if (1), (2), and (3) hold.

3.1.4 Makespan of Valid Execution Plans

The push, map, shuffle, and reduce phases are executed in sequence, but there are three possible assumptions we can make regarding the boundaries between these phases. The simplest is that a *global barrier* exists, requiring that all nodes complete one phase before any node begins the next. This clearly satisfies data dependencies, but limits concurrency. An alternative is a *local barrier*, where each individual node can move from one phase to the next without regard to the progress of other nodes. This increases concurrency, allowing for faster execution. Finally, phases may be *pipelined*, where each individual node can start a phase as soon as *any* data is available rather than waiting for *all* of its data to arrive. Such pipelined concurrency allows for even lower makespan.

Which of these three options is allowable, however, depends on the application; the typical MapReduce assumption is that shuffle and reduce are separated by at least a local barrier, though with some system and application changes [13], this assumption can be relaxed. Other phase boundaries are more flexible. Due to space limitations, this paper presents a model for the makespan of a valid execution plan only for the case where all barriers are global. For details of how local barriers and pipelining are modeled in our framework, we refer readers to our Technical Report [14].[2]

In addition to assuming global barriers at each phase boundary, we also assume that inputs are available at all data sources at time zero when the push phase begins. For each mapper $j \in M$, the time for the push phase to end, $\mathrm{push_end}_j$, is then equal to the time at which the last of its input data arrives; i.e.,

$$\mathrm{push_end}_j = \max_{i\in S} \frac{D_i x_{ij}}{B_{ij}}. \tag{4}$$

Since we assume a global barrier after the push phase, all mappers must receive their data before the push phase ends and the map phase begins. The time when the map phase starts, denoted by $\mathrm{map_start}$, thus obeys the following:

$$\mathrm{map_start} = \max_{j\in M} \mathrm{push_end}_j. \tag{5}$$

The computation at each mapper takes time proportional to the data pushed to that mapper. Thus, the time $\mathrm{map_end}_j$ for mapper $j \in M$ to complete its computation obeys the following:

$$\mathrm{map_end}_j = \mathrm{map_start} + \frac{\sum_{i\in S} D_i x_{ij}}{C_j}. \tag{6}$$

As a result of the global barrier, the shuffle phase begins when all mappers have finished their respective computations. Thus, the time $\mathrm{shuffle_start}$ when the shuffle phase starts obeys the following:

$$\mathrm{shuffle_start} = \max_{j\in M} \mathrm{map_end}_j. \tag{7}$$

The shuffle time for each reducer is governed by the slowest shuffle link into that reducer. Thus the time when shuffle ends at reducer $k \in R$, denoted by $\mathrm{shuffle_end}_k$, obeys the following:

$$\mathrm{shuffle_end}_k = \mathrm{shuffle_start} + \max_{j\in M} \frac{\alpha \sum_{i\in S} D_i x_{ij} x_{jk}}{B_{jk}}. \tag{8}$$

2. In short, modeling local barriers and pipelining involves (a) removing the constraints in (5)-(7); (b) defining an abstract addition operator that adds for local barriers, and takes the maximum for global barriers; and (c) rewriting the constraints in (6), (8), and (10) in terms of this abstract addition operator.

TABLE 1
Measured Bandwidth (KBps) of the Slowest/Fastest Links between Nodes in Each Continent

	US	EU	Asia
US	216/9,405	110/2,267	61/3,305
EU	794/2,734	4,475/11,053	1,502/1,593
Asia	401/3,610	290/1,071	23,762/23,875

Following the global barrier assumption, the reduce phase computation begins only after all reducers have received all of their data. Let reduce_start be the time when the reduce phase starts. Then

$$\text{reduce_start} = \max_{k \in R} \text{shuffle_end}_k. \quad (9)$$

Reduce computation at a given node takes time proportional to the amount of data shuffled to that node. Thus, the time when reduce ends at node k denoted by reduce_end_k obeys the following:

$$\text{reduce_end}_k = \text{reduce_start} + \frac{\alpha \sum_{i \in S} \sum_{j \in M} D_i x_{ij} x_{jk}}{C_k}. \quad (10)$$

Finally, the makespan equals the time when the last reducer finishes. Thus

$$\text{Makespan} = \max_{k \in R} \text{reduce_end}_k. \quad (11)$$

3.2 Model Validation

To validate our model, we modify the Hadoop MapReduce implementation to follow a user-specified execution plan. In particular, we modify two aspects of Hadoop: how map and reduce tasks are *defined*, as well as how they are *scheduled*. Changes to task definition lie exclusively at the application level, in the `InputSplit`, `InputFormat`, and `Partitioner` classes. Map tasks are defined to combine inputs from several data sources to match the distribution dictated by the execution plan. Reduce tasks are defined to contain larger or smaller fractions of the intermediate key space as based on the execution plan.

Our changes to task scheduling lie largely at the application level and only partially at the system-level, specifically to the `JobInProgress` class. Map and reduce tasks are assigned only to specified hosts, except when executed as speculative backups, or re-executed for fault tolerance purposes. Of course, this is only a brief summary of our implementation. For much deeper detail, readers are referred to our Technical Report [14].

For validation purposes, we estimate parameters for our model based on the PlanetLab globally distributed testbed [10]. We base these measurements on a set of eight physical PlanetLab nodes distributed across eight sites, including four in the United States, two in Europe, and two in Asia. Table 1 shows the intra-continental and inter-continental bandwidths (averaged over several measurements) and highlights the heterogeneity that characterizes such geo-distributed networks. Measured compute rates for a sample computation range from as low as 9 MBps to as high as about 90 MBps. We provide these parameters to our model to compute a predicted makespan, and we also execute a corresponding job on our modified Hadoop implementation on a local cluster emulating the measured PlanetLab heterogeneity [14] and measure the actual makespan.

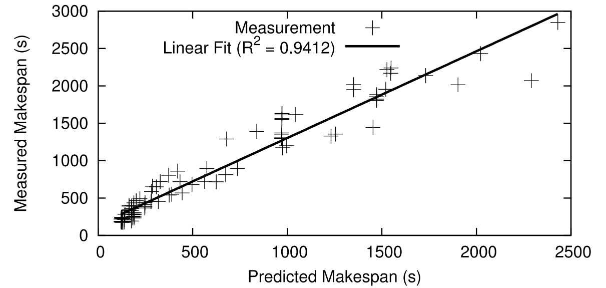


Fig. 3. Measured versus model-predicted makespan.

The results of the validation, for 96 combinations of α values, network heterogeneity, barrier configurations, and execution plans, are shown in Fig. 3, where the vertical axis shows the measured makespan and the horizontal axis shows the model prediction based on the C_i, B_{ij}, and α parameters, and (1)-(11). We observe a strong correlation (R^2 value of 0.9412) between predicted makespan and measured makespan. In addition, the linear fit to the data points has a slope of 1.1464, which shows there is also a strong correspondence between the absolute values of the predicted and measured makespan.

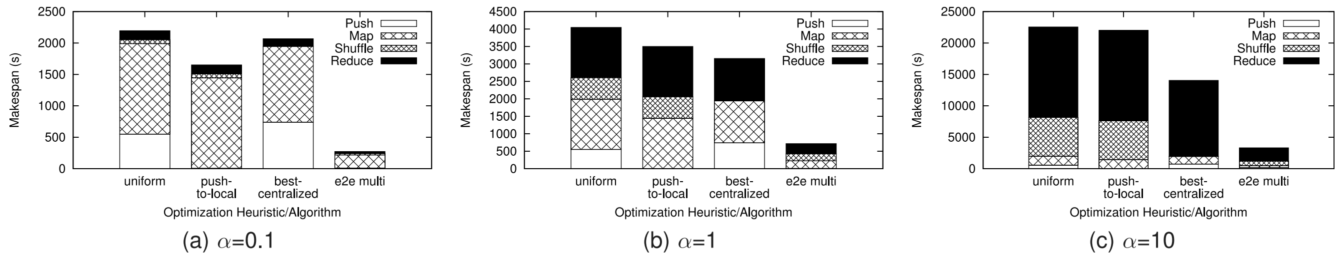
3.3 Optimization Algorithm

Having defined and validated our model, we now use it to find the execution plan—the x_{ij} values—to minimize the makespan of a MapReduce job. To do so, we formulate an optimization problem with two key properties. First, it minimizes the *end-to-end* makespan of the MapReduce job, including the time from the initial data push to the final reducer execution. Thus, though its decisions may be suboptimal for individual phases, they will be optimal for the overall execution of the job. Second, our optimization is *multi-phase* in that it controls the data dissemination across both the push and shuffle phases; i.e., it outputs the best way to disseminate both the input and intermediate data so as to minimize makespan.

Viewing (1)-(11) as constraints, we need to minimize an objective function that equals the variable Makespan. To perform this optimization efficiently, we rewrite the constraints in linear form to obtain a mixed integer program. (See our Technical Report [14] for the details of this transformation.) Writing it as MIP opens up the possibility of using powerful off-the-shelf solvers such as Gurobi 5.0.0 as we use here.

4 Model-Driven Insights

We now apply our model-driven, end-to-end, multi-phase optimization as an oracle, deriving high-level insights that inspire the design of pragmatic implementation techniques described in later sections. To realistically model the geo-distributed environments that motivate our work, we use actual measurements of the compute speeds and link bandwidths from PlanetLab nodes distributed around the world, including four in the US, two in Europe, and two in Asia



(a) α=0.1 (b) α=1 (c) α=10

Fig. 4. A comparison of three optimization heuristics and our end-to-end multi-phase optimization (e2e multi).

with compute rates and interconnection bandwidths as described in Section 3.2.

4.1 Heuristics versus Model-Driven Optimization

We first compare our optimal execution plans to three baseline heuristics, which we refer to as *uniform*, *best-centralized* and *push-to-local*. In the *uniform* heuristic, sources push uniformly to all mappers, and mappers shuffle uniformly to all reducers. This reflects the expected distribution if data sources and mappers were to randomly choose where to send data. We model this heuristic by adding the following additional constraints:

$$\text{Uniform Push:} \quad \forall i \in S, j \in M\colon x_{ij} = \frac{1}{|M|}, \tag{12}$$

$$\text{Uniform Shuffle:} \quad \forall j \in M, k \in R\colon x_{jk} = \frac{1}{|R|}. \tag{13}$$

The *best-centralized* heuristic pushes data to a single mapper node and performs all map computation there. Likewise, all reduce computation is performed at a single reducer node at the same location. We can model this heuristic by simply excluding all mappers and reducers other than the selected centralized node, and then using the model as usual. We compute the makespan for each possible centralized location, and report the minimum among these values, hence the name *best-centralized*.

The *push-to-local* heuristic pushes data from each source to the most local mapper (in this case, to a mapper on the same node) and uses a uniform shuffle; it corresponds to the purely distributed extreme. We model this heuristic by adding the uniform shuffle constraint from Equation (13) as well as a new local push constraint:

$$\forall i \in S, j \in M\colon x_{ij} = \begin{cases} 1 & \text{if } j \text{ is closest to source } i, \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

Fig. 4 shows the makespan achieved by each of these heuristics, as well as by our end-to-end multi-phase optimization algorithm (e2e multi) for three values of α. We make several observations from these results. First, both the *uniform* and *best-centralized* heuristics have poor push performance relative to the *push-to-local* heuristic and our end-to-end optimization. The reason is that the *uniform* and *best-centralized* heuristics place heavy traffic on slow wide-area links while the *push-to-local* heuristic uses fast local links for the push. Our optimization approach intelligently decides how to utilize links to minimize end-to-end makespan.

Second, we see that the *best-centralized* heuristic becomes more favorable relative to the *push-to-local* heuristic as α increases, demonstrating that neither is universally better than the other. The reason is that, as α increases, the shuffle phase becomes heavier relative to the push phase. The *best-centralized* heuristic avoids wide-area links in the shuffle, and this becomes increasingly valuable as the shuffle becomes dominant.

Finally, we see that our end-to-end multi-phase optimization approach is significantly better than any of the three heuristics, reducing makespan over the best heuristic by 83.5, 77.3, and 76.4 percent for $\alpha = 0.1$, 1, and 10 respectively. This demonstrates that the best data placement is one that considers resource and application characteristics.

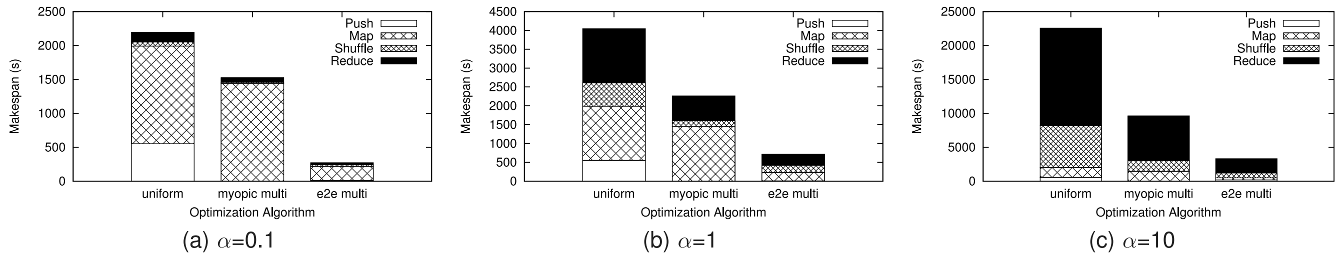
4.2 End-to-End versus Myopic

Next, we assess the benefit of optimizing with an *end-to-end* objective as opposed to *myopically* minimizing the time for a single phase, and we find that an effective optimization should pursue an end-to-end objective.

The distinction between end-to-end and myopic alternatives lies in the *objective function* that is being minimized. With end-to-end, the objective function is the overall makespan of the MapReduce job, whereas a myopic optimizer uses the time for a single phase (or subset of phases) as its objective function. Myopic optimization is a localized optimization, e.g., pushing data from data sources to mappers in a manner that minimizes the data push time. Such local optimization might result in suboptimal global execution times by creating bottlenecks in other phases of execution. Note that myopic optimization can be applied to multiple phases in succession. For example, the push phase might first be optimized to minimize push time and then the shuffle phase could be optimized to minimize shuffle time assuming that the input data were pushed to mappers according to the first optimization. Such myopic optimizations can be modeled by replacing (11) with alternate objective functions. For example, in this section we will model myopically optimizing to minimize the push time (minimize $\max_{j \in M} \text{push_end}_j$) followed by myopically optimizing for minimum shuffle time (minimize $\max_{k \in R} \text{shuffle_end}_k$.) Note that computing such a myopic multi-phase plan requires solving several optimizations in sequence.

As an additional point of comparison, we also consider the *uniform* heuristic from the previous subsection.

In Fig. 5, we show the makespan achieved in three different cases: (i) a uniform data placement; (ii) a myopic, multi-phase optimization, where the push and shuffle phases are optimized myopically in succession; and (iii) our end-to-end, multi-phase optimization that minimizes the total job makespan. Note that since both (ii) and (iii) are multi-phase, the primary difference between them is that one is myopic



(a) α=0.1 (b) α=1 (c) α=10

Fig. 5. Our end-to-end multi-phase optimization (e2e multi) compared to a uniform heuristic and a myopic optimization approach.

and the other is end-to-end. We see that for each of three values of α, the myopic optimization reduces the makespan over the uniform data placement approach (by 30, 44, and 57 percent for $\alpha = 0.1$, 1, and 10 respectively), but is significantly outperformed by the end-to-end optimization (which reduces makespan by 87, 82, and 85 percent). This is because, although the myopic approach makes locally optimal decisions at each phase, these decisions may be globally suboptimal. Our end-to-end optimization makes globally optimal decisions. As an example, for $\alpha = 0.1$, while both the myopic and end-to-end approaches dramatically reduce the push time over the uniform approach, the end-to-end approach is also able to reduce the map time substantially whereas the myopic approach makes no improvement to the map time. A similar trend is evident for $\alpha = 10$, where the end-to-end approach is able to lower the reduce time significantly over the myopic approach. These results show the benefit of an end-to-end, globally optimal approach over a myopic, locally optimal but globally suboptimal approach. Clearly, an effective optimization approach should pursue and end-to-end rather than a myopic optimization objective.

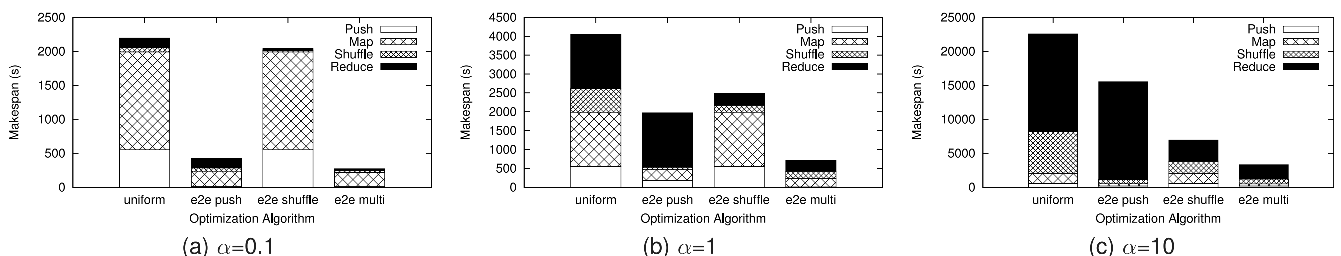
4.3 Single-Phase versus Multi-Phase

Finally, we assess the importance of optimizing multiple phases rather than only a single phase, and we find that an effective optimization should jointly control multiple phases.

The distinction between single-phase and multi-phase alternatives lies in which phase (push, shuffle, or both) is *controlled* by the optimization, so this distinction is orthogonal to the end-to-end versus myopic distinction. A single-phase optimization controls the data distribution of one phase—e.g., the push phase—alone, while using a fixed data distribution for the other communication phase. This is a subtle but important distinction; note that a single-phase optimization can also be end-to-end if it controls the single phase so as to achieve the minimum end-to-end makespan. We model a single-phase optimization by using one of the uniform push or shuffle constraints (12) or (13) to constrain the data placement for one of the phases, while allowing the other phase to be optimized.

In Fig. 6, we compare (i) a uniform data placement, (ii) an end-to-end single-phase push optimization that assumes a uniform shuffle, (iii) an end-to-end single-phase shuffle optimization that assumes a uniform push, and (iv) our end-to-end multi-phase optimization. Note that both the single-phase optimizations here are end-to-end optimizations in that they attempt to minimize the total makespan of the MapReduce job. The primary difference between (ii) and (iii) on the one hand and (iv) on the other is that the former are single-phase and the latter multi-phase, allowing us to evaluate the relative benefit of single- versus multi-phase optimization.

We observe that across three α values, the multi-phase optimization outperforms the best single-phase optimization (by 37, 64, and 52 percent for $\alpha = 0.1$, 1, and 10 respectively). This shows the benefit of being able to control the data placement across multiple phases. Further, for each α value, optimizing the bottleneck phase brings greater reduction in makespan than optimizing the non-bottleneck phase. For instance, for $\alpha = 0.1$, the push and map phases dominate the makespan in the baseline (25 and 66 percent of the total runtime for uniform, respectively) and push optimization alone is able to reduce the makespan over uniform by 80 percent by lowering the runtime of these two phases. On the other hand, for $\alpha = 10$, the shuffle and reduce phases dominate (27 and 64 percent of total runtime for uniform, respectively) and optimizing these phases via shuffle optimization reduces makespan by 69 percent over uniform. By influencing the data placement across multiple phases, however, our multi-phase optimization improves both the bottleneck as well as non-bottleneck phases. When there is no prominent bottleneck phase ($\alpha = 1$), the multi-phase optimization outperforms the best single-phase optimization substantially (by 64 percent).



(a) α=0.1 (b) α=1 (c) α=10

Fig. 6. Our end-to-end multi-phase optimization (e2e multi) compared to a uniform heuristic as well as end-to-end push (e2e push) and end-to-end shuffle (e2e shuffle) optimization.

These results show that the multi-phase optimization is able to automatically optimize the execution independent of the application characteristics.

Interestingly, Figs. 6b and 6c show that optimizing earlier phases can have a positive impact on the performance of later phases. In particular, for $\alpha = 10$, push optimization also reduces the shuffle time, even though the push and map phases themselves contribute little to the makespan. This is because the decision of where to push inputs affects where the mappers will produce outputs, in turn impacting how the intermediate data are shuffle to reducers.

We therefore conclude that an effective optimization approach should jointly control multiple phases rather than controlling only a single phase in isolation.

4.4 Summary

Overall, our model-driven optimization leads to the following high-level insights:

- Neither a purely distributed approach nor a purely centralized approach is the best for all situations. Instead, the best placement typically lies between either of these extremes.
- Optimizing with an end-to-end *objective* yields significantly lower makespan than optimizing with a myopic objective. The reason is that end-to-end optimization tolerates local suboptimality in order to achieve global optimality.
- Jointly *controlling* multiple phases lowers the makespan compared to controlling only a single phase. Although optimizing the bottleneck phase alone can be beneficial, each phase contributes to the overall makespan, so controlling multiple phases has the best overall benefit, independent of application characteristics.

These high-level insights inspire the design of *cross-phase* optimization techniques that are suitable to a real-world MapReduce implementation, where decisions must be made under imperfect knowledge, and the dynamic nature of network, compute, and application characteristics might render a static optimization approach impractical.

Based on the need for an end-to-end optimization objective, our *Map-aware Push* technique makes push decisions aware of their impact not just on the push phase, but also on the map phase. In the spirit of multi-phase optimization, we also control communication in the shuffle phase through our *Shuffle-aware Map* technique, which factors in predicted shuffle costs while placing map tasks. This technique also contributes to a more nearly end-to-end optimization, as it allows both push and map placement to be influenced by downstream costs in the shuffle phase.

5 Map-Aware Push

The first opportunity for cross-phase optimization in MapReduce lies at the boundary between push and map. A typical practice is what we call a *push-then-map* approach, where input data are imported in one step, and computation begins only after the data import completes. This approach has two major problems. First, by forcing all computation to wait for the slowest communication link, it introduces waste. Second, separating the push and map phases deprives map tasks of a way to demand more or less work based on their compute capacity. This makes scheduling the push in a map-aware manner more difficult.

To overcome these challenges, we propose two changes: first, overlapping—or pipelining—the push and map phases; and second, inferring locality information at runtime and driving scheduling decisions based on this knowledge. Dynamically inferring locality information at runtime makes this approach adaptable to changing network conditions, and pipelining brings two main benefits: hiding latency, and opening a new feedback channel. First, by pipelining push and map, latency of the push phase can be partially hidden by the map phase, as pipelining allows map computation to begin at each mapper as soon as data begin to arrive there, without the need to wait for the full push phase to complete. Second, overlapping the two phases addresses the lack of feedback from the map phase back to the push phase. Without such feedback, and absent any *a priori* knowledge of the map phase performance, we are left with few options other than simply optimizing the push phase in isolation. Such a single-phase optimization favors pushing more data to mappers with faster incoming network links, but these may not be the best mappers from a computational perspective. For better overall performance, we need to weigh the two factors of network bandwidth and computation capacity and trade off between faster push and faster map. By overlapping push and map, mappers can pull work from data sources on demand, thereby influencing the push phase.

5.1 Map-Aware Push Scheduling

With overlapped push and map, the distribution of computation across mapper nodes can be demand-driven. Specifically, whereas push-then-map first pushes data from sources, our approach logically *pulls* data from sources on-demand. Using existing Hadoop mechanisms, this on-demand pull is initiated when a mapper becomes idle and requests more work, so faster mappers can perform more work. This is how our proposed approach addresses map computation heterogeneity.

To address network heterogeneity, our *Map-aware Push* technique departs from the traditional Hadoop approach of explicitly modeling network topology as a set of racks and switches, and instead infers locality information at runtime. It does this by monitoring source-mapper link bandwidth at runtime and estimating the push time for each source-mapper pair. Specifically, let d be the size of a task in bytes (assume for ease of presentation that all task sizes are equal) and let $L_{s,m}$ be the link speed between source node s and mapper node m in bytes per second. Then we estimate the push time $T_{s,m}$ in seconds from source s to mapper m as

$$T_{s,m} = \frac{d}{L_{s,m}}. \tag{15}$$

Let S denote the set of all sources that have not yet completed their push. Then when mapper node m requests work, we grant it a task from source $s^* = \arg\min_{s \in S} T_{s,m}$. Intuitively, this is equivalent to selecting the closest task in terms of network bandwidth. This policy is similar to

Hadoop's default approach of preferring *data-local* tasks, but our overall approach is distinguished in two ways. First, rather than *reacting* to data movement decisions that have already been made in a separate push phase, it *proactively* optimizes data movement and task placement in concert. Second, it discovers locality information dynamically and automatically rather than relying on an explicit user-specified model.

5.2 Implementation in Hadoop

We have implemented this technique in Hadoop 1.0.1. The overlapping itself is possible using existing Hadoop mechanisms, but a more creative deployment: We set up a Hadoop Distributed File System (HDFS) instance comprising the data source nodes, which we refer to as the "remote" HDFS instance and use directly as the input to a Hadoop MapReduce job. Map tasks in Hadoop typically read their inputs from HDFS, so this allows us to directly employ existing Hadoop mechanisms.[3]

Our scheduling enhancements, on the other hand, require modification to the Hadoop task scheduler. To gather the bandwidth information mentioned earlier, we add a simple network monitoring module which records actual source-to-mapper link performance and makes this information accessible to the task scheduler. For Hadoop jobs that read HDFS files as input, each map task corresponds to an `InputSplit` which in turn corresponds to an HDFS file block. HDFS provides an interface to determine physical block locations, so the task scheduler can determine the source associated with a task and compute its $T_{s,m}$ based on bandwidth information from the monitoring module. If there are multiple replicas of the file block, then $T_{s,m}$ can be computed for each replica, and the system can use the replica that minimizes this value. The task scheduler then assigns tasks from the closest source s^* as described earlier.

6 Shuffle-Aware Map

We have shown how the map phase can influence the push phase, in terms of both the volume of data each mapper receives as well as the sources from which each mapper receives its data. In turn, the push determines, in part, when a map slot becomes available for a mapper. Thus, from the perspective of the push and map phases, a set of mappers and their data sources are decided. This decision, however, ignores the downstream cost of the shuffle and reduce as we will show. In this section, we show how the set of mappers can be adjusted to account for the downstream shuffle cost. This was also motivated in Section 2 as we illustrated the importance of shuffling and merging intermediate results close to mappers, particularly for shuffle-heavy jobs.

In traditional MapReduce, intermediate map outputs are shuffled to reducers in an all-to-all communication. In Hadoop, one can (with some effort) control the granularity of reduce tasks and the amount of work each reducer will obtain. However, these decisions ignore the possibility that a mapper-reducer link may be very poor. For example, in Fig. 1, if the link between mapper M_1 and reducer R_1 or R_2 is poor, then performing too much map computation at M_1 may lead to a bottleneck in the shuffle phase. For applications in which shuffle is dominant and the network is heterogeneous, this phenomenon can greatly impact performance.

3. To improve fault tolerance, we have also added an option to cache and replicate inputs at the compute nodes. This reduces the need to re-fetch remote data after task failures or for speculative execution.

Two solutions are possible: changing the reducer nodes, or reducing the amount of work done by mapper M_1 and in turn reducing the volume of data traversing the bottleneck link(s). We present an algorithm that takes the latter approach. In this way, the downstream shuffle can impact the map. This is similar to the *Map-aware Push* technique where the map influenced the push.

As in typical MapReduce, our Shuffle-aware Map technique assumes that the reducer nodes are known *a priori*. We also assume that we know the approximate distribution of reducer tasks: i.e., we know the fraction of intermediate data allocated to each reducer node. This allows our algorithm to determine how much data must traverse each mapper-to-reducer link. To achieve a true multi-phase optimization, our model-driven optimization can be used in conjunction with pre-profiled mapper-reducer link speeds and reducer node execution power to determine an appropriate allocation of the intermediate key space to reducers.

6.1 Shuffle-Aware Map Scheduling

To estimate the impact of a mapper node upon the reduce phase, we first estimate the time taken by the mapper to obtain a task, execute it, and deliver intermediate data to all reducers (assuming parallel transport). The intuition is that if the shuffle cost is high then the mapper node should be throttled to allow the map task to be allocated to a mapper with better shuffle performance. We estimate the finish time T_m for a mapper m to execute a map task as follows: $T_m = T_m^{map} + T_m^{shuffle}$, where T_m^{map} is the estimated time for the mapper m to execute the map task, including the time to read the task input from a source (using the *Map-aware Push* approach), and $T_m^{shuffle}$ is the estimated time to shuffle the accumulated intermediate data D_m up to the current task, from mapper m to all reducer nodes. Let $D_{m,r}$ be the portion of D_m destined for reducer r, and $L_{m,r}$ be the link speed between mapper node m and reducer node r. Then, we can compute

$$T_m^{shuffle} = \max_{r \in R} \left(\frac{D_{m,r}}{L_{m,r}} \right). \quad (16)$$

The *Shuffle-aware Map* scheduling algorithm uses these T_m estimates to determine a set of *eligible mappers* to which to assign tasks. The intuition is to throttle those mappers that would have an adverse impact on the performance of the downstream reducers. The set of eligible mappers M_{Elig} is based on the most recent T_m values and a tolerance parameter β:

$$M_{Elig} = \left\{ m \in M \,|\, T_m \leq \min_{m \in M} T_m + \beta \right\}, \quad (17)$$

where M is the set of all mapper nodes.

The intuition is that if the execution time for a mapper (including its shuffle time) is too high, then it should not be assigned more work at present. The value of the tolerance parameter β controls the aggressiveness of the algorithm in excluding slower mappers from being assigned work. At

TABLE 2
Measured Bandwidths in EC2

From	To	Bandwidth (MB/s)
Source EU	Worker EU	8
Source EU	Worker US	3
Source US	Worker EU	3
Source US	Worker US	4
Worker EU	Worker EU	16
Worker EU	Worker US	2
Worker US	Worker EU	5
Worker US	Worker US	2

one extreme, $\beta = 0$ would enforce assigning work only to the mapper with the earliest estimated finish time, intuitively achieving good load balancing, but leaving all other mappers idle for long periods of time. At the other extreme, a high value of $\beta > (\max_{m \in M} T_m - \min_{m \in M} T_m)$ would allow all mapper nodes to be eligible irrespective of their shuffle performance, and would thus reduce to the default MapReduce map scheduling. We select an intermediate value:

$$\beta = \frac{(\max_{m \in M} T_m - \min_{m \in M} T_m)}{2}. \quad (18)$$

The intuition behind this value is that it biases towards mappers with better shuffle performance. This is just one possible threshold; many others are possible.

We note that the algorithm makes its decisions dynamically, so that over time, a mapper may become eligible or ineligible depending on the relation between its T_m value and the current value of $\min_{m \in M} T_m$. As a result, this algorithm allows an ineligible mapper node to become eligible later should other nodes begin to offer worse performance. Similarly, a mapper may be throttled if its performance degrades over time.

6.2 Implementation in Hadoop

We have implemented this *Shuffle-aware Map* scheduling algorithm by modifying the task scheduler in Hadoop. The task scheduler now maintains a list of estimates T_m for all mapper nodes m, and updates these estimates as map tasks finish. It also uses the mapper-to-reducer node pair bandwidth information obtained by the network monitoring module to update the estimates of shuffle times from each mapper node. Every time a map task finishes, the task tracker on that node asks the task scheduler for a new map task. At that point, the scheduler uses (17) to determine the eligibility of the node to receive a new task. If the node is eligible, then it is assigned a task from the best source determined by the *Map-aware Push* algorithm described in Section 5. On the other hand, if the node is not eligible, then it is not assigned a task. However, it can request for work again periodically by piggybacking on heartbeat messages, when its eligibility will be checked again.

7 Experimental Results

To evaluate the performance of our techniques, we carry out experiments on globally distributed testbeds on both

TABLE 3
Measured Bandwidths in PlanetLab

From	To	Bandwidth (MB/s)
All sources	All workers	1-3
Workers A-C	Workers A-C	4-9
Workers A-C	Worker D	2
Worker D	Workers A-C	0.2-0.4

Amazon EC2 and PlanetLab using three applications: `WordCount`, `InvertedIndex`, and `Sessionization`.

Our EC2 testbed uses eight nodes in total, all of the m1.small instance type. These nodes are distributed evenly across two EC2 regions: four in the US and the other four in Europe. Each node hosts one map slot and one reduce slot. Two PlanetLab nodes, one in the US and one in Europe, serve as distributed data sources. Table 2 shows the bandwidths measured between the multiple nodes in this setup.

For our PlanetLab testbed, we continue to use two nodes as distributed data sources, and we use four other globally distributed nodes as compute nodes, each hosting one map slot and one reduce slot. Table 3 shows the bandwidths measured between the multiple nodes in this setup.

Our `WordCount` job takes as input random text data generated by the Hadoop example `randomtextwriter` generator, and computes the number of occurrences of each word in the input data. This is a map-heavy application, producing a relatively small volume of intermediate data.

The `InvertedIndex` application takes as input a set of eBooks from Project Gutenberg [15] and produces, for each word in its input, the complete list of positions where that word appears. This application shuffles a large volume of intermediate data, so it is an interesting application for evaluating our *Shuffle-aware Map* scheduling technique.

Our `Sessionization` application takes as input a set of web server logs from the WorldCup98 trace [16], and sorts these records first by client and then by time. The sorted records for each client are then grouped into a set of "sessions" based on the gap between consecutive records. This is also a shuffle-heavy application, so we expect it to benefit from our *Shuffle-aware Map* technique.

Using these testbeds and applications, we first study the *Map-aware Push* and *Shuffle-aware Map* techniques in isolation, and then show results where both techniques are applied.

7.1 Map-Aware Push

We are interested in the performance of *Map-aware Push*, which overlaps push and map and infers locality at runtime, compared to a baseline push-then-map approach. To implement the push-then-map approach, we also run an HDFS instance comprising the compute nodes (call this the "local" HDFS instance). We first run a Hadoop DistCP job to copy from the remote HDFS to this local HDFS, and then run a MapReduce job directly from the local HDFS. We compare application execution time using these two approaches. We run this experiment on both our Amazon

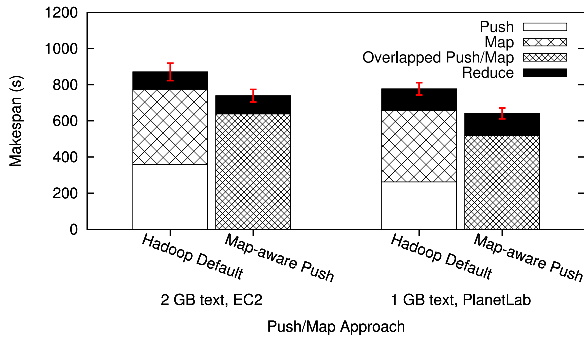


Fig. 7. Makespan of a Hadoop `WordCount` job on text data for the push-then-map approach and the *Map-aware Push* approach on globally distributed environments.

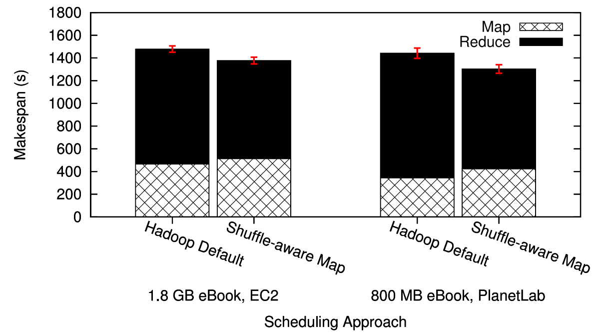


Fig. 8. Makespan of the `InvertedIndex` job on eBook data for the default Hadoop scheduler and our *Shuffle-aware Map* scheduler. Both approaches use an overlapped push and map in these experiments.

EC2 and PlanetLab testbeds. Because we are concerned primarily with push and map performance at this point, we run the map-heavy `WordCount` application.

The left cluster of Fig. 7 shows the makespan[4] of the `WordCount` job on 2 GB of input data on our EC2 testbed, and it shows that our approach to overlapping push and map reduces the total runtime of the push and map phases by 17.7 percent, and the total end-to-end makespan by 15.2 percent.

Next, we run the same experiment on our PlanetLab testbed, using 1 GB of text input data due to the smaller cluster size. The right cluster of Fig. 7 shows that push-map overlap can reduce runtime of the push and map phases by 21.3 percent and the whole job by 17.5 percent in this environment. We see a slightly greater benefit from push-map overlap on PlanetLab than on EC2 due to the increased heterogeneity of the PlanetLab testbed.

7.2 Shuffle-Aware Map

We now present some results that show the benefit of *Shuffle-aware Map*. Here we run our `InvertedIndex` application, which shuffles a large volume of intermediate data and is therefore an interesting application for evaluating our *Shuffle-aware Map* scheduling technique.

First, we run this application on our EC2 testbed. In this environment, we use 1.8 GB of eBook data as input, which yields about 4 GB of intermediate data to be shuffled to reducers. The left cluster of Fig. 8 shows the runtime for a Hadoop baseline with push and map overlapped, as well as the runtime of our *Shuffle-aware Map* scheduling technique, also with push and map overlapped.

The reduce time shown includes shuffle cost. Note that in *Shuffle-aware Map* the shuffle and reduce time (labeled "reduce" in the figure) are smaller than in stock Hadoop. Also observe that in *Shuffle-aware Map* the map times go up slightly—our algorithm has decided to make this tradeoff to achieve better performance.

On our wider-area PlanetLab testbed we use 800 MB of eBook data and see a similar pattern, as the right cluster of Fig. 8 shows. Again, an increase in map time is tolerated to reduce shuffle cost. This may mean that a slower mapper is given more work since it has faster links to downstream reducers. For this application, we see performance improvements of 6.8 and 9.6 percent on EC2 and PlanetLab, respectively.

7.3 Putting It All Together

To determine the complete end-to-end benefit of our proposed techniques, we run experiments comparing a traditional Hadoop baseline, which uses a push-then-map approach, to an alternative that uses both *Map-aware Push* and *Shuffle-aware Map*. Taken together, we will refer to our techniques as the *End-to-end* approach. We focus here on the `InvertedIndex` and `Sessionization` applications, both of which are relatively shuffle-heavy, representing a class of applications that can benefit from our *Shuffle-aware Map* technique.

7.3.1 Amazon EC2

First, we explore the combined benefit of our techniques on our EC2 testbed. The left cluster of Fig. 9 shows results for the `InvertedIndex` application, where we see that our approaches reduce the total makespan by about 9.7 percent over the traditional Hadoop approach. There is little difference in total push and map time, so most of this reduction in runtime comes from a faster shuffle and reduce (labeled "reduce" in the figure). This demonstrates the effectiveness of our *Shuffle-aware Map* scheduling approach, as well as the ability of our techniques to automatically determine how to tradeoff between faster push and map phases or faster shuffle and reduce phases.

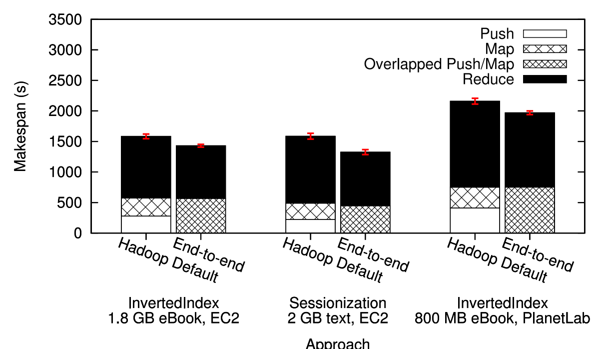


Fig. 9. Makespan for traditional Hadoop compared with our proposed *Map-aware Push* and *Shuffle-aware Map* techniques (together, *End-to-end*) for `InvertedIndex` and `Sessionization` applications on globally distributed environments.

4. Throughout this paper, error bars indicate 95 percent confidence intervals.

TABLE 4
Number of Map Tasks Assigned to Each Mapper Node in Our PlanetLab Testbed

Scheduler	Node A	Node B	Node C	Node D
Hadoop Default	5	4	5	3
End-to-end	5	5	6	1

Now consider the `Sessionization` application, which has a slightly lighter shuffle and slightly heavier reduce than does the `InvertedIndex` application. The center cluster of Fig. 9 shows that for this application on our EC2 testbed, our approaches can reduce makespan by 8.8 percent. Again most of the reduction in makespan comes from more efficient shuffle and reduce phases. Because this application has a slightly lighter shuffle than does the `InvertedIndex` application, we would expect a slightly smaller performance improvement, and our experiments confirm this.

7.3.2 PlanetLab

Now we move to the PlanetLab testbed, which exhibits more extreme heterogeneity than EC2. For this environment, we consider only the `InvertedIndex` application, and the right cluster of Fig. 9 shows that our approaches can reduce makespan by about 16.4 percent. Although we see a slight improvement in total push and map time using our approach, we can again attribute the majority of the performance improvement to a more efficient shuffle and reduce.

To more deeply understand how our techniques achieve this improvement, we record the number of map tasks assigned to each mapper node, as shown in Table 4. We see that both Hadoop and our techniques assign fewer map tasks to Mapper D, but that our techniques do so in a much more pronounced manner.

Network bandwidth measurements reveal that this node has much slower outgoing network links than do the other mapper nodes; only about 200-400 $\mathrm{KB/s}$ compared to about 4-9 $\mathrm{MB/s}$ for the other nodes (see Table 3). By scheduling three map tasks there, Hadoop has effectively "trapped" intermediate data, resulting in a prolonged shuffle phase. Our *Shuffle-aware Map* technique, on the other hand, has the foresight to avoid this problem by denying Mapper D additional tasks even when it becomes idle and requests more work.

8 Related Work

Several recent works consider data and task placement in MapReduce. Purlieus [17] categorizes jobs by the relative size of inputs to their map and reduce phases and applies different optimization approaches for each category; this is similar to our use of α as a key application parameter. CoGRS [18] considers partition skew and locality and places reduce tasks as close to their intermediate data as possible. Unlike our approach, however, it does not consider reduce task locality while placing map tasks.

Gadre et al. [19] and Kim et al. [20] focus on geo-distributed data and compute resources, respectively. Kim et al. focus on reaching the end of the shuffle phase as early as possible. Mattess et al. [21] dynamically provision remote resources to improve map-phase completion time. Unlike these works, our multi-phase optimization controls both map and reduce phases, and we use an end-to-end optimization objective.

Other work considers wide-area MapReduce deployments from an architectural standpoint. Luo et al. [22] propose running multiple local MapReduce jobs and aggregating their results using a new "Global Reduce" phase. They assume that communication is a small part of the total runtime, whereas our approach can find the best execution plan for a broad range of application and system characteristics.

Heterogeneity in tightly coupled local clusters has been addressed by several recent works. For example Zaharia et al. [23] propose the LATE scheduler to better detect straggler tasks due to hardware heterogeneity. Mantri [24] and Tarazu [25] take *proactive* approaches to recognizing stragglers and dynamically balancing workloads, respectively. While these works focus on tightly coupled local clusters, they may also apply at the level of a single data center in a geo-distributed deployment.

Sandholm and Lai [26], Quincy [27], and Delay Scheduling [28] focus on multi-job scheduling in MapReduce clusters. While this paper focuses on a single job running in isolation, extending our work to multiple concurrent jobs in a geo-distributed setting is an interesting area for future work.

Our model-driven optimization requires knowledge of network bandwidth and compute speeds. The network weather service (NWS) [29] and OPEN [30] demonstrate systems for gathering and disseminating such information in a scalable manner. He et al. [31] investigate formula- and history-based techniques for predicting the throughput of large TCP transfers, and show that history-based approaches with even a small number of samples can lead to accurate prediction.

9 Conclusion

Many emerging data-intensive applications are geo-distributed, due to the distributed generation and storage of their input data. MapReduce performance suffers in these geo-distributed settings, as the impact of one phase upon another can lead to severe bottlenecks. Existing placement and scheduling mechanisms do not take such cross-phase interactions into account, and while they may try to optimize individual phases, they can results in globally poor decisions, resulting in poor overall performance. To overcome these limitations, we have presented a model-driven optimization framework, as well as cross-phase optimization algorithms suitable for a real-world MapReduce implementation. Our model-driven optimization has two key features: (i) it optimizes end-to-end makespan unlike myopic optimizations which make locally optimal but globally suboptimal decisions, and (ii) it controls multiple MapReduce phases to minimize makespan, unlike single-phase optimizations which control only individual phases. We have used our model to show how our end-to-end multi-phase optimization can significantly reduce execution time compared to myopic or single-phase baselines.

Applying these insights, we have developed techniques to implement cross-phase optimization in a real-world MapReduce system. The key idea behind these techniques is to consider not only the execution cost of an individual

task or computational phase, but also the impact on downstream phases. *Map-aware Push* enables push-map overlap to hide latency and enable dynamic feedback between the map and push phases, allowing nodes with higher speeds and faster links to process more data at runtime. *Shuffle-aware Map* enables a shuffle-aware scheduler to feed back the cost of a downstream shuffle into the map process and affect the map phase. Mappers with poor outgoing links to reducers are throttled, eliminating the impact of mapper-reducer bottleneck links. For a range of heterogeneous environments (multi-region Amazon EC2 and PlanetLab) and diverse data-intensive applications (`WordCount`, `InvertedIndex`, and `Sessionization`) we have shown the performance potential of our techniques, as runtime is reduced by 7-18 percent depending on the execution environment and application.

ACKNOWLEDGMENTS

The authors would like to acknowledge National Science Foundation grants CNS-0643505, CNS-0519894, and IIS-0916425, which supported this research, and Chenyu Wang for his contributions to the cross-phase optimization techniques and experiments.

REFERENCES

[1] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. Conf. Innovative Data Syst. Res.*, 2011, pp. 223–234.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proc. 10th USENIX Conf. Operating Syst. Design Implementation*, 2012, pp. 251–264.

[3] E. Nygren, R. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 3, pp. 2–19, 2010.

[4] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, WA, USA: Microsoft Research, 2009.

[5] D. Patterson and J. Gray, "A conversation with Jim Gray," *Queue*, vol. 1, no. 4, pp. 8–17, Jun. 2003.

[6] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. J. Freedman, "Making every bit count in wide-area analytics," in *Proc. 14th USENIX Conf. Hot Topics Oper. Syst.*, 2013, p. 6.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating Syst. Des. Implementation*, 2004, pp. 137–149.

[8] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman, "Exploring MapReduce efficiency with highly-distributed data," in *Proc. 2nd Int. Workshop MapReduce Its Appl.*, 2011, pp. 27–33.

[9] Hadoop [Online]. Available: http://hadoop.apache.org

[10] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: An overlay testbed for broad-coverage services," *ACM SIGCOMM*, vol. 33, no. 3, pp. 3–12, 2003.

[11] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proc. ACM 2nd ACM Symp. Cloud Comput.*, 2011, pp. 18:1–18:14.

[12] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Bridging the tenant-provider gap in cloud services," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 6:1–6:14.

[13] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell, "Breaking the MapReduce stage barrier," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2010, pp. 235–244.

[14] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing MapReduce for highly distributed environments," Dept. Comput. Sci. Eng., Univ. Minnesota, Minneapolis, MN, USA, Tech. Rep. TR 12-003, Feb. 2012.

[15] Project Gutenberg [Online]. Available: http://www.gutenberg.org/

[16] M. Arlitt and T. Jin, "Workload characterization of the 1998 World Cup web site," HP Labs, Palo Alto, CA, USA, Tech. Rep. HPL-1999-35R1, Sep. 1999.

[17] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: locality-aware resource allocation for MapReduce in a cloud," in *Proc. Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2011, pp. 58:1–58:11.

[18] M. Hammoud, M. S. Rehman, and M. F. Sakr, "Center-of-Gravity reduce task scheduling to lower MapReduce network traffic," in *Proc. IEEE Cloud*, 2012, pp. 49–58.

[19] H. Gadre, I. Rodero, and M. Parashar, "Investigating MapReduce framework extensions for efficient processing of geographically scattered datasets," in *Proc. ACM SIGMETRICS*, 2011, pp. 116–118.

[20] S. Kim, J. Won, H. Han, H. Eom, and H. Y. Yeom, "Improving Hadoop performance in intercloud environments," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 3, pp. 107–109, 2011.

[21] M. Mattess, R. Calheiros, and R. Buyya, "Scaling MapReduce applications across hybrid clouds to meet soft deadlines," in *Proc. IEEE 27th Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2013, pp. 629–636.

[22] Y. Luo, Z. Guo, Y. Sun, B. Plale, J. Qiu, and W. W. Li, "A hierarchical framework for cross-domain MapReduce execution," in *Proc. 2nd Int. Workshop Emerging Comput. Methods Life Sci.*, 2011, pp. 15–22.

[23] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 29–42.

[24] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, and B. Saha, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 265–278.

[25] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing MapReduce on heterogeneous clusters," in *Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 61–74.

[26] T. Sandholm and K. Lai, "MapReduce optimization using dynamic regulated prioritization," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst.*, 2009, pp. 299–310.

[27] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 261–276.

[28] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[29] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: A distributed resource performance forecasting service for metacomputing," *Future Generation Comput. Syst.*, vol. 15, pp. 757–768, 1999.

[30] J. Kim, A. Chandra, and J. Weissman, "Passive network performance estimation for large-scale, data-intensive computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 8, pp. 1365–1373, Aug. 2011.

[31] Q. He, C. Dovrolis, and M. Ammar, "On the predictability of large transfer tcp throughput," in *Proc. ACM SIGCOMM*, 2005, pp. 145–156.

Benjamin Heintz received the BS degree in mechanical engineering and economics from Northwestern University, and the MS degree in computer science from the University of Minnesota. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at the University of Minnesota. His research interests include distributed systems and data-intensive computing. He is a student member of the IEEE.

Abhishek Chandra received the BTech degree in computer science and engineering from the Indian Institute of Technology Kanpur, and the MS and PhD degrees in computer science from the University of Massachusetts Amherst. He is an associate professor in the Department of Computer Science and Engineering at the University of Minnesota. His research interests are in the areas of operating systems and distributed systems. He is a member of the IEEE.

Ramesh K. Sitaraman received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, and the PhD degree in computer science from Princeton University. He is currently in the School of Computer Science at the University of Massachusetts at Amherst. His research spans all aspects of Internet-scale distributed systems, including algorithms, architectures, performance, energy efficiency, user behavior, and economics. As a principal architect, he helped create the Akamai network and is an Akamai fellow. He is best known for his pioneering role in helping build the first large content delivery networks (CDNs) that currently deliver much of the world's web content, streaming videos, and online applications. He received the National Science Foundation (NSF) CAREER Award and a Lilly fellowship. He is a member of the IEEE.

Jon Weissman received the PhD degree in computer science from the University of Virginia. He is a professor of computer science at the University of Minnesota. His current research interests are in distributed systems, high-performance computing, and resource management. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.