Network Cost-aware Geo-distributed Data Analytics System

Kwangsung Oh, Minmin Zhang, Abhishek Chandra, and Jon Weissman

Abstract—Many geo-distributed data analytics (GDA) systems have focused on the network performance-bottleneck: inter-data center network bandwidth to improve performance. Unfortunately, these systems may encounter a *cost-bottleneck* ($) because they have not considered data transfer cost ($), one of the most expensive and heterogeneous resources in a multi-cloud environment. In this paper, we present *Kimchi*, a network cost-aware GDA system to meet the cost-performance tradeoff by exploiting data transfer cost heterogeneity to avoid the cost-bottleneck. Kimchi determines cost-aware task placement decisions for scheduling tasks given inputs including data transfer cost, network bandwidth, input data size and locations, and desired cost-performance tradeoff preference. In addition, Kimchi is also mindful of data transfer cost in the presence of dynamics. Kimchi has been applied to two common GDA MapReduce models: synchronous barrier and asynchronous push-based shuffle. A Kimchi prototype has been implemented on Spark, and experiments show that it reduces cost by 5% $\sim$ 24% without impacting performance and reduces query execution time by 45% $\sim$ 70% without impacting cost compared to other baseline approaches centralized, vanilla Spark, and bandwidth-aware (e.g. Iridium). More importantly, Kimchi allows applications to explore a much richer cost-performance tradeoff space in a multi-cloud environment.

Index Terms—Geo-distributed Data; Multi-DCs; Multi Cloud Providers; Data Analytics System.

1 Introduction

1.1 Motivation

Recently, geo-distributed data analytics (GDA) has become a popular method for mining valuable information from globally distributed data generated by users and systems in a multi-cloud environment[1], in areas as diverse as querying global trend detection on social network data, and log monitoring of geo-distributed CDN servers [23], [37], [39].

One simple approach is to aggregate all data within a single data center (DC) and then analyze the data using a data analytic framework, e.g., Hadoop [25] and Spark [48]. However, this requires a significant amount of time for migrating large amounts of data into a centralized DC via a scarce and expensive resource, WAN bandwidth. Another alternative is to process data in-place, but it is well-known that Hadoop and Spark perform poorly due to the large overhead of inter-DC traffic in the shuffle stages [18].

To address this network overhead, numerous approaches have been proposed [26], [29], [31], [33], [38], [43], [44] that attempt to minimize network usage and consider heterogeneous network bandwidth in their scheduling and data placement decisions. However, much of the existing work focuses primarily on how to efficiently use the WAN for performance but does not address data transfer cost ($)[2], one of the most expensive and heterogeneous resources in a multi-cloud environment. This cost can be significant for continuous queries that require large data transfer between DCs. Recent works [21], [28] confirmed that the WAN bandwidth cost makes up a significant fraction of the overall cost.

TABLE 1
Heterogeneous data transfer cost (per GB) between DCs (as of Dec, 2019) - SA: South America, AP: Asia Pacific.

Origin \ Destination		AWS			Azure		
		US East	SA East	AP NE	US East	SA East	AP NE
AWS	US East	$0	**$0.02**	**$0.02**	$0.09	$0.09	$0.09
	SA East	$0.16	$0	$0.16	**$0.25**	**$0.25**	**$0.25**
	AP NE	$0.09	$0.09	$0	$0.126	$0.126	$0.126
Azure	US East	$0.087	$0.087	$0.087	$0	$0.087	$0.087
	SA East	$0.181	$0.181	$0.181	$0.181	$0	$0.181
	AP NE	$0.138	$0.138	$0.138	$0.138	$0.138	$0

One may think that minimizing WAN usage results in minimized cost. Yet, this is not always true due to heterogeneous pricing policies, e.g., up to an 8X inter-DC transfer cost difference even within the same cloud provider (AWS), and a 12.5X cost difference across cloud providers (AWS and Azure), as shown in Table 1. Since a large amount of data transfer in a GDA occurs between DCs [3], [38], [42], a GDA may encounter the *cost-bottleneck* due to a cost-agnostic approach that may significantly inflate operational cost.

In this paper, we argue that *data transfer cost* must be a first-class consideration for a GDA running in a multi-cloud environment to avoid this cost-bottleneck. To consider cost, we are motivated by the following questions:

- What is the minimal query execution time given a target cost budget ($)?

- *K. Oh and M. Zhang are with the Department of Computer Science, University of Nebraska Omaha, Omaha, NE, 68182.*
  E-mail: {kwangsungoh, minminzhang}@unomaha.edu
  Homepage: http://faculty.ist.unomaha.edu/kwangsungoh
- *A. Chandra and J. Weissman are with the Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, MN, 55455.*
  E-mail: {chandra, jon}@cs.umn.edu
  Homepage: http://dcsg.cs.umn.edu

1. We use the term multi-cloud to refer to both a single cloud provider that spans multiple DCs as well as multiple DCs that span multiple cloud providers.

2. We use the term cost to refer to monetary cost of data transfer unless mentioned.

TABLE 2
Feature comparison with state-of-the-art. △ indicates that metric is considered but with limitations.

	Clarinet	Iridium	Tetrium	Kimchi
Heterogeneous network B/W	✓	✓	✓	✓
Cost-performance tradeoff		△	△	✓
Handling dynamics			△	✓
Heterogeneous network cost				✓
Cost-aware push-based shuffle				✓

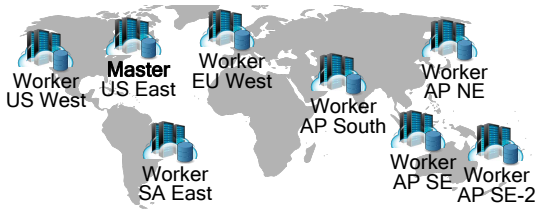


Fig. 1. An example of geo-distributed DCs where a GDA is running to analyze geo-distributed data.

- What is the feasible cost range to execute a query?
- How can a GDA achieve the desired cost-performance tradeoff in a multi-cloud environment?
- How can a GDA handle dynamics for better performance during query execution without additional cost?

To answer these questions, we have designed and implemented Kimchi, a cost-aware GDA system. The goal of Kimchi is to explore a richer cost-performance tradeoff space and to achieve the best performance within a desired cost budget. To this end, Kimchi solves a constrained MIP (mixed integer programming) task placement problem that meets a desired tradeoff preference.

One significant challenge to cost reduction is dynamics that are common in a multi-cloud environment [28], [32], [49], e.g., network contention and bandwidth changes, but most GDA systems [26], [31], [33], [38], [43], [44] ignore dynamics during query execution. While handling dynamics is important for performance, a large data migration may occur for handing dynamics that may lead to a cost-bottleneck. To adapt quickly to dynamics and avoid the cost-bottleneck, Kimchi uses a heuristic that adjusts task placement with cost-awareness at run-time. Finally, Kimchi considers an asynchronous model, i.e., *push-based shuffle* mechanism [19], [22], [33], that overcomes the barrier synchronization of a MapReduce programming model for cost-aware performance to avoid a possible cost-bottleneck.

A prototype implementation of Kimchi is built on the Spark [48] framework. We support new Spark properties that control Kimchi settings, so that Spark applications can utilize Kimchi *without any modification*. We evaluate Kimchi on both simulated cloud and Amazon AWS using the well-known benchmarks TPC-DS [9] and TeraSort [8] to show its benefit. Experimental results show that Kimchi reduces cost by up to 24% without impacting performance and reduces query execution time by up to 70% without impacting cost compared to a centralized approach, the vanilla Spark scheduler, and a bandwidth-aware approach, e.g., Iridium [38]. In addition, the results show that Kimchi can handle dynamics during query execution without additional cost. More importantly, Kimchi allows applications to explore a richer tradeoff space between cost and performance given different data distribution in a multi-cloud environment.

1.2 Research Contributions

The main contributions of this paper are as follows:

- The design and implementation of Kimchi, the first GDA system (to the best of our knowledge) that optimizes task placement with a consideration of heterogeneous data transfer cost ($) in a multi-cloud environment.
- The observation that minimizing data transfer size may not lead to a minimum cost.
- Formulation and solution of the cost-aware task placement problem that allows applications to explore this *richer cost-performance tradeoff space*.
- Handling dynamics during query execution for cost-aware performance that avoids expensive re-evaluation of global task placement.
- Applying our solution to a push-based shuffle mechanism that maintains low cost and improves query performance.

Table 2 shows the comparison between state-of-the-art solutions and Kimchi. While all prior approaches consider heterogeneous network bandwidth, they do not consider heterogeneous data transfer cost. This network cost-agnostic approach can lead to a cost-bottleneck. For the cost-performance tradeoff, Iridium [38] and Tetrium [29] offered a knob to explore the tradeoff space by limiting WAN usage. These systems, however, may not achieve a desired tradeoff due to heterogeneous data transfer cost, i.e., *minimizing data transfer size does not necessarily yield minimized data transfer cost*, as we will show. To handle dynamics, Tetrium [29] re-evaluates the global optimized task placement decision. However, it may encounter a cost-bottleneck due to a cost-agnostic approach for handling dynamics. In addition, frequently re-evaluating global task placement can incur performance overhead. We will show how Kimchi handles dynamics while avoiding the cost-bottleneck and performance overhead of re-evaluation in Section 4.2.

2 System Model and Problem Statement

2.1 System Model

Data Center (DC) Setting: We consider geo-distributed data analytics (GDA) running in a multi-cloud environment. Figure 1 shows the DC locations where a GDA is running, e.g., the master is in US East and the workers are in the other regions. Each DC has heterogeneous data transfer cost policies based on geographical locations and providers as shown in Table 1. Cloud providers only charge for outbound data transfer, while inbound data transfer is free of charge. The network bandwidth between DCs is highly heterogeneous due to different bandwidth capacities and can fluctuate due to dynamics, e.g., network contention on shared network links and bandwidth changes by cloud providers. We could observe varying WAN bandwidth based on DC locations, i.e., 36 Mbps $\sim$ 300 Mbps, using a network performance measurement tool, iPerf3 [5], as shown in [32]. We will discuss a high-bandwidth WAN ($>$ 1 Gbps) in Section 8.

Compute Resources: While computational resources in each DC are finite due to a cost constraint and their costs are also heterogeneous, we attack the WAN bandwidth and cost, as this can significantly inflate cost as well as degrade

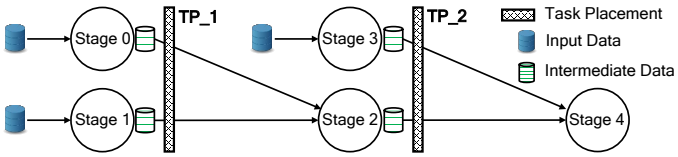


Fig. 2. A DAG (Directed Acyclic Graph) example for a job with 5 stages.

performance for a large class of GDA applications, the latter as noted in many previous works [28], [38], [43], [44], [49]. We will show potential performance degradation with infinite compute resources, and discuss heterogeneous compute resource costs in Section 6.1 and Section 7, respectively.

Applications: Applications generate network bandwidth-intensive MapReduce like queries to a GDA master that will assign tasks to workers that spawn executors to execute tasks. While applications have different cost-performance tradeoff preferences, achieving reduced query latency within their target budget is desirable for these applications. We believe that our approach can be applied to any application that needs to transfer large data frequently between DCs, where cost and performance are important.

Queries and Data: Figure 2 shows a job (query) example that has several stages, i.e., three map stages (0, 1, and 3) and two shuffle stages (2 and 4). Each stage accesses *geo-distributed* input data, e.g., stage 0 may access input data from all DCs in Figure 1. Map tasks access input data locally and output the results (intermediate data) locally. Shuffle tasks access intermediate data from all DCs. Intermediate data can be accessed multiple times for a single query, e.g., self-join operation. We make the following assumptions.

- Map tasks are executed in-situ using data locality-aware scheduling, e.g., Hadoop [25] and Spark [48].
- Intermediate data size is large [3], [38], [42].
- The amount of data drops off quickly in subsequent stages for queries that have many sequential stages [10], [38].

Given these assumptions, map stages are not a performance bottleneck due to data locality [12], [47] and in-memory caching [13], [48]. In shuffle stages, however, large intermediate data transfer occurs via all-to-all communication among DCs using WAN bandwidth, the main performance bottleneck in a GDA [26], [31], [33], [38], [43], [44]. For cost, accessing data (input or intermediate) within a DC does not incur cost. Shuffle stages, however, require large intermediate data transfer via WAN. This incurs cost, which can cause a cost-bottleneck. In short, shuffle stages are the bottleneck for both performance and cost, and we therefore focus on shuffle stages in this work. While we consider multiple stages, determining optimal task placement for multiple shuffle stages results in a non-convex optimization [29], [31], [38]. Instead, we adopt a greedy approach that determines task placement independently for each shuffle stage. This approach may not be optimal for a query but will work well under our assumption: rapid data size reduction in subsequent stages, i.e., a few shuffle stages are significant for overall cost and performance. For data distribution, we will discuss a richer cost-performance tradeoff space with varying data distribution in Section 6.4.

Barrier Synchronization: In a MapReduce programming model, stages cannot start until all their dependencies are resolved due to barrier synchronization, e.g., stage 2 cannot start until stages 0 and 1 are done in Figure 2. To overcome this limitation, intermediate data can be pushed to target DCs in the background (asynchronously) in map stages, i.e., *push-based shuffle* [19], [22], [33]. We consider these two common GDA implementation models (barrier and push-based) and show how they work in our system in Section 4.

Task Placement Problem: The task placement problem consists of determining a set of tuples (tasks and corresponding DC locations) to satisfy the desired cost-performance tradeoff preference. Task placement is determined for shuffle (or result) stages that need to shuffle intermediate data. For example, there are two task placement problems: **TP_1** and **TP_2**, in Figure 2. For task scheduling, Kimchi determines task placements before executing shuffle stages, and shuffle tasks will be scheduled based on these decisions as will be shown in Section 4.1. Note that input data is computed in-situ as assumed, thus task placement decisions for map stages are not considered in this work.

TABLE 3
3 DCs Example.

	DC A	DC B	DC C
Intermediate Data	240MB	120MB	60MB
Uplink BW	10MB/s	10MB/s	10MB/s
Downlink BW	1MB/s	10MB/s	10MB/s

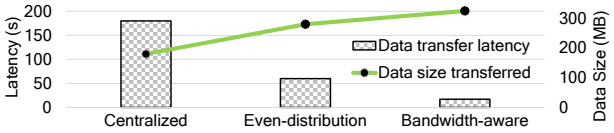


Fig. 3. Data transfer latency and data size transferred.

2.2 Illustrative Example

In this section, we illustrate how data transfer cost can affect overall operational cost by applying three different task placement approaches: centralized, even distribution (scheduling for load balancing), and network bandwidth-aware (e.g., Iridium [38]) to an example GDA scenario. We use the example in Iridium for comparison using the intra-AWS costs as shown in Table 1. As an example, consider an application sending a MapReduce query to a GDA that must process data contained in three DCs. The network environment and intermediate data size for a shuffle stage are as shown in Table 3 in which DC A's downlink is a significant performance bottleneck in terms of bandwidth.

Figure 3 shows the data transfer latency and data size transferred for each approach. While the centralized approach minimizes data size transferred (180MB: 120MB from B and 60MB from C to A), it increases data transfer latency significantly (180 secs) due to the network bottleneck in DC A (1MB/s downlink). The bottleneck link is avoided in the network bandwidth-aware approach in which tasks are assigned based on given bandwidth, approximately (A: 5% (1MB/s), B: 47.5% (10MB/s), C: 47.5% (10MB/s)) to minimize data transfer latency (17.1 secs) with more data transferred (322.5MB: 9MB from B and C to A (9 secs), 142.5MB from A and C to B (14.2 secs), and 171 MB from A and B to C (17.1 secs)). The even-distribution approach

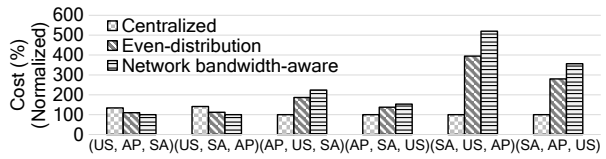


Fig. 4. Data transfer cost comparison with varying AWS DC locations, US East, AP NE, and SA. Costs are normalized to the minimum cost for each configuration.

offers performance somewhere between other approaches as shown in Iridium [38].

Heterogeneous data transfer cost: Figure 4 shows the cost comparison with varying DC locations. Interestingly, the first two left-most cases, (US, AP, SA) and (US, SA, AP), show that *minimizing data transfer size does not necessarily lead to minimized data transfer cost*. That is, the centralized approach results in 131% ~ 140% cost compared to the network bandwidth-aware approach even with less data transferred, i.e., 180MB (centralized) vs. 322.5MB (bandwidth-aware). This is because a large portion of data need to be sent from DCs where data transfer cost is expensive; i.e., SA and AP NE, in the centralized approach. For these cases, cost can be minimized if the centralized DC is determined based on both cost and data size rather than just data size e.g., choosing AP and SA as the centralized DC for each case respectively can minimize cost, i.e., *cost-aware centralized*. Other cases show that the network bandwidth-aware approach can significantly increase the cost up to 5.2X, showing that a consideration of both heterogeneous network bandwidth and cost can open up a richer cost-performance tradeoff space. However, this example also shows the optimization problem to be complex.

Dynamics during query execution: For dynamics, assume that DC B's downlink becomes 1MB/s during query execution and the bandwidth-aware approach is used. This increases data transfer latency significantly (142 seconds: 142.5MB from A and C to B at worst case) if dynamics are not handled. To avoid this, tasks need to be re-assigned (A: 8.3% (1MB/s), B: 8.3% (1MB/s), C: 83% (10MB/s)) to minimize performance degradation (299.9MB from A and B to C (30 secs) at worst case). In this case, network cost can be either increased up by 30% or reduced by 13% based on DC locations as compared to the original task placement, which opens additional tradeoffs.

3 Cost-aware Task Placement

3.1 Cost and Performance Tradeoff

In a multi-cloud environment, applications may have different cost-performance tradeoff preferences. For applications that want to have fast query response irrespective of cost, bandwidth-aware approaches e.g., Iridium [38], would be preferable. On the other hand, for applications that want to minimize operational cost, the cost-aware centralized approach e.g., the four right-most cases in Figure 4, would be preferable. However, most applications likely want to achieve cost and performance somewhere between these two approaches.

Figure 5 shows the extreme possibilities of two approaches in terms of cost, i.e., cost-aware centralized for *Min cost* and bandwidth-aware for *Best performance*. Applications cannot reduce cost below Min cost and cannot improve performance above Best performance, even by paying more than the *Max cost* that is associated with *Best performance*. The figure also shows a tradeoff space between the two extremes. Our goal is to allow applications to explore this tradeoff space by providing their desired tradeoff preference (C_pref), on the continuum between the two extreme cases.

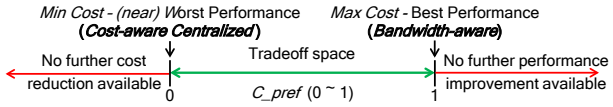


Fig. 5. Possible tradeoff space between two extremes (centralized and bandwidth-aware approaches) in terms of cost.

TABLE 4
Inputs for task placement.

Input	Description
C_pref	Desired cost preference (0 ~ 1)
D	Set of DCs
C_{ij}	Data transfer cost from DC i to DC j
NB_{ij}	Network Bandwidth (MB/s) from DC i to DC j
T	Set of Tasks
I_{ij}	Intermediate Data size for shuffle task i in DC j

To this end, we determine the feasible cost boundaries or ranges, i.e., Min cost and Max cost, to estimate a target cost budget that corresponds to the desired cost-performance tradeoff preference C_pref. With estimated Min and Max costs, the target budget can be estimated as follows.

$$target\_budget = Min\ Cost + C\_pref \cdot (Max\ Cost - Min\ Cost) \quad (1)$$

For example, if Min cost ($100) and Max cost ($500) are estimated with given inputs, and C_pref is set to 0.5, $300 is used as a target budget. The Min cost can be estimated with a task placement that minimizes cost without considering network bandwidth (performance), and the Max cost can be estimated with a task placement that minimizes query execution time without considering cost.

3.2 Task Placement Problem Formulation

Given a target_budget, we formulate the task placement problem as a *budget-constrained optimization problem* that outputs a desired task placement consisting of a list of tuples {task, DC-location}. We use mixed integer programming (MIP) to optimally solve this problem.

3.2.1 Inputs and Outputs

Table 4 shows the inputs to our model. Note that the inputs, NB, T, and I, are continuously updated for each stage.

Tradeoff Preference: Applications need to provide their cost-performance tradeoff preferences, C_pref. Applications can set their preferences to 0 for minimized cost (cost-aware centralized approach), 1 for minimized latency (bandwidth-aware approach), or any number between 0 and 1 to specify a cost-performance tradeoff preference. C_pref will be converted to the target cost budget that is used as a cost constraint. This is the only input that applications need to provide for task placement.

TABLE 5
Output Example.

Task Id	Target DC	Latency (Secs)	Cost ($)
0	US East	9.5	0.0045
1	US West	8.7	0.0041
2	EU West	12.2	0.0034
...			
199	AP SE-2	8.2	0.0031

Data Transfer Cost: Data transfer cost information is available from cloud providers' web pages [15], [16] and is rarely changed (static in this work).
Network Bandwidth Information: The latest information for inter-DC bandwidth (bytes/s) is estimated by executors running on each DC when they transfer data between DCs.
Data Size for Shuffle Tasks: Intermediate data size for all shuffle tasks stored in each DC is required. This information is available from the MapOutputTracker in a GDA.
Output: Given these inputs, we compute task placement consisting of a set of pairs ({task, DC}). Output includes the expected latency and network cost for each task. Table 5 shows an example of task placement. We will show how these values are used by a scheduler to handle dynamics during query execution in Section 4.2.

3.2.2 Optimization Problem Formulation

We solve three optimization problems to determine the task placement.
Determining the Target Budget: As shown in Equation 1, to determine the target budget, we solve two sub-problems to get *Min cost* and *Max cost*. The following variables and constraints (Equation 2) are used in this formulation:

$$\forall i \in T, \forall j \in D : A_{ij}$$

A_{ij} are binary variables (0 or 1): if 1, task i is assigned to DC j.

$$\forall t \in T, \forall i \in D : \sum_{i} A_{ti} \ = 1 \quad (2)$$

This is a constraint with which a task can be assigned only to a single DC.
Min cost: The first sub-problem is to determine the lowest data transfer cost (lower cost bound).
Objective: Minimize total data transfer cost.

$$\forall t \in T, \forall i, j \in D : \sum_{t} A_{ti} \cdot I_{tj} \cdot C_{ji} \quad (3)$$

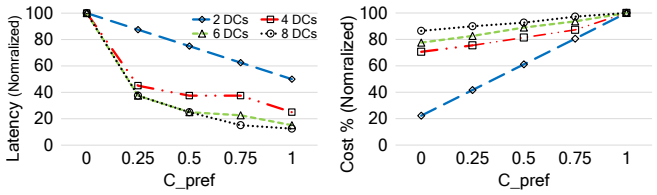
We found that all tasks are assigned to a single DC to minimize cost in the task placement by solving this sub-problem. The Min cost can be computed by using cost information and data size of each DC.
Max cost: The second sub-problem is to determine the upper cost bound for the lowest data transfer latency.
Objective: Minimize maximum data transfer latency.

$$\forall t \in T, \forall i, j \in D : Max(\frac{\sum_t A_{ti} \cdot I_{tj}}{NB_{ji}}) \quad (4)$$

In Equation 4, *we only consider the maximum data transfer latency between DCs, i.e., the main bottleneck, that determines overall latency (performance)*. With the task placement for the minimized network latency (best performance), the upper-bound data transfer cost (maximum cost) can be computed with given cost information. Finally, we can determine the *target_budget* with Min and Max cost with C_pref as shown in Equation 1.
Solving Task Placement with a Target Budget: Once the *target_budget* is determined, we solve the minimum data transfer latency problem (Equation 4) again with a *target_budget* as a constraint.
Objective: Minimize maximum latency, Equation 4 with the following constraint:

$$\forall t \in T, \forall i, j \in D : \sum_{t} A_{ti} \cdot I_{tj} \cdot C_{ji} <= target\_budget \quad (5)$$

By solving Equation 4 with Equation 5 as a constraint, *we minimize the highest data transfer latency, i.e., query execution time, given target_budget*.



(a) Latency comparison. (b) Cost comparison.

Fig. 6. The highest data transfer latency of tasks in a shuffle stage and data transfer cost for each C_pref and DC configuration. The values of Figure 6(a) and Figure 6(b) are normalized to C_pref = 0 case and C_pref = 1 case, respectively.

3.3 Illustrating Cost-performance Tradeoff

Applications can explore a cost-performance tradeoff space using C_pref as explained in Section 3.1. For example, Figure 6 shows the cost-performance tradeoff space for a simple synthetic workload that has a single shuffle stage with varying C_pref and 4 different DC configurations: 1) 2 DCs in US East and SA East, 2) 4 DCs in US East, US West, AP SE-2, and SA East, 3) 6 DCs in US East, US West, EU West, AP SE, AP SE-2, SA East, and 4) all 8 DCs shown in Figure 1. Note, we assume that the intermediate data are evenly distributed and use measured network bandwidth between DCs.

Figure 6(a) shows that if C_pref is increasing, the highest latency is decreasing and thus performance is improving. While the latency decreases smoothly from C_pref = 0.25 to C_pref = 1, C_pref = 0 case shows a steep latency increase as all data are sent to a single DC (or a few DCs), i.e., network contention. Figure 6(b) shows that cost is increased as C_pref is increased, i.e., performance improvement with additional cost. The figure also shows a trend where the available cost reduction range decreases as the number of DCs increases. For example, 2 DCs case has 77.5% cost reduction opportunity but 8 DCs case only has 13.5%. This is because the cost variance is getting smaller as the number of DCs increases in the DC configurations, i.e., 2 DCs case for the biggest cost variance and 8 DCs case for the smallest cost variance, and we consider even data distribution. We will show how data distribution affects cost reduction opportunities in Section 6.4. Applications can

TABLE 6
DCs to execute tasks and number of tasks for the DCs with 8 DCs.

C_pref	DC selections (the number of tasks for the DC)
1	US East (1), US West (1), EU West (1), SA East (1), AP SE (1), AP SE-2 (1), AP NE (1), AP South (1)
0.75	US West (1), AP SE (1), AP SE-2 (2), AP NE (2), AP South (1), SA East (1)
0.5	SA East (2), AP SE (2), AP SE-2 (2), AP NE (2)
0.25	SA East (3), AP SE (1), AP SE-2 (4)
0	SA East (8)

Algorithm 1 Cost-aware Task Scheduling

```
1: procedure CostAwareScheduling(taskPlacement tp, dc d, adjustOption o, dcs [] idle)
2:     t_id = DequeueTask (tp, d)
3:     if t_id >= 0 then
4:         if CheckDynamics (tp, d, t_id) == false then
5:             ExecuteTask(t_id, d)
6:         else
7:             AdjustTask(d, t_id, o, idle)
8:     else
9:         idle += d
```

check the approximate cost-performance curves shown in Figures 6(a) and 6(b) without actual query execution as we will explain in Section 5.

Table 6 shows which DCs are chosen for tasks for each C_pref value with an 8 DCs configuration. For example, tasks are assigned to all DCs if C_pref is set to 1 and all tasks are assigned to a single DC (SA East) if C_pref is set to 0. From the table, we can see that *Tasks are assigned to DCs that have expensive outbound data transfer cost in order to reduce cost for a small C_pref value by avoiding data transfer out from those DCs*. For example, tasks are mainly assigned to SA East, AP SE, and AP SE-2 with small C_pref values. Overall, task placements are determined to meet desired cost-performance tradeoffs by adjusting the highest data transfer latency, i.e., minimizing the highest data transfer latency (best performance) with a cost constraint.

4 Cost-aware Task Scheduling

In this section, we will present how Kimchi uses cost-aware task placement to help applications trade off between cost and performance. We will also show how Kimchi handles dynamics using a heuristic algorithm for cost-aware performance. Lastly, we will present how Kimchi determines task placement by a cost-aware push-based mechanism that utilizes WAN bandwidth efficiently.

4.1 Task Scheduling

When a shuffle stage is ready to be executed, Kimchi estimates a task placement for the stage at *query execution time* (run-time) by solving a constrained MIP task placement problem as explained in Section 3. Note, Kimchi can get the latest inputs, including exact intermediate data size and locations, due to a barrier synchronization. Once a task placement is determined, the Kimchi scheduler starts scheduling tasks as determined by the task placement optimization that meets cost-performance preferences. Note, we only consider shuffle stages for cost-aware scheduling in this work, and tasks of map stages are scheduled using data locality-aware scheduling, e.g., Hadoop [25] and Spark [48].

Algorithm 1 outlines how task placement is performed by the Kimchi scheduler. The task scheduling function will be called repeatedly whenever the scheduler receives a DC offer to execute one of the tasks in a stage (line 1). The scheduler finds a task for the DC from the task placement list and assigns the task to the DC after checking dynamics for the DC (line 2 $\sim$ 5). If dynamics are detected, the scheduler finds another DC among the idle DCs for the task to avoid performance degradation (line 7). An idle DC is defined as one for which there are no tasks remaining in the current stage (line 9). We will show how Kimchi detects and handles dynamics in the next section.

4.2 Cost-aware Task Adjustment

A static optimal task placement may be sub-optimal due to mis-estimated bandwidth, or may become sub-optimal due to dynamics that are common in a multi-cloud environment [28], [32], [49], e.g., network contention and bandwidth changes. Since sub-optimal task placement can affect overall cost and performance, it needs to be handled. While most GDA systems [26], [31], [38], [43], [44] have ignored dynamics during query execution, Tetrium [29] re-evaluated task placement to handle dynamics at run-time. However, Tetrium may encounter a cost-bottleneck due to its cost-agnostic approach for handling dynamics that may require large data transfer. For performance, re-evaluating an optimal task placement can incur additional overhead, especially for large-scale jobs. In addition, frequent re-evaluation may occur if dynamics are frequent and/or the sensitivity trigger for re-evaluation is improperly tuned, leading to significant performance overhead.

To avoid the cost-bottleneck and performance overhead of frequent re-evaluation, Kimchi uses a heuristic algorithm that re-assigns tasks to other idle DCs in the presence of dynamics if doing so does not incur additional cost. The intuition behind this heuristic is that dynamics will lead to congestion at one or more DCs, leaving others to complete their tasks much more quickly, and then become idle. Kimchi utilizes these idle DCs with an expectation that the latency for the task can be amortized by running other tasks in parallel. That is, Kimchi tries to utilize bandwidth completely to improve performance. Note, idle DCs are tracked by Scheduler 1 and treated as idle only within the current stage. To avoid additional cost by task adjustment, Kimchi adjusts task placement such that they do not incur additional cost, i.e., cost-aware task adjustment. For better performance, Kimchi offers a scheduling option *always_adjust_tasks* that allows the scheduler to adjust tasks without cost awareness, i.e., cost-agnostic task adjustment.

Algorithm 2 outlines how Kimchi detects and handles sub-optimal task placement due to dynamics. The Kimchi scheduler estimates a latency for a task using the latest bandwidth information before it assigns the task to the designated DC (line 2). The scheduler compares the estimated latency with the expected latency computed by the optimizer as explained in Section 3.2.1. If the difference is larger than a threshold (*LatThreshold*), Kimchi concludes that dynamics have occurred (lines 3 $\sim$ 5) and tries to assign the task to another DC if doing so improves performance (line 7). To pick a new DC location that best meets application preferences, e.g., cost or performance, Kimchi traverses idle DCs and executes the task on the chosen one. If no DC is

Algorithm 2 Checking and Handling Dynamics

```
1: function CHECKDYNAMICS(taskPlacement tp, dc d, taskId t_id)
2:     curLat = GetLatForTask(t_id, d)
3:     expectedLat = tp[t_id].expectedLat
4:     if curLat > expectedLat + LatThreshold then
5:         return true
6:     return false
7: procedure ADJUSTTASK(dc d, taskId t_id, dcs [] idle, adjustOption o)
8:     chosen_d = d
9:     candidates = []
10:    for each idle_d in idle do
11:        if CompareLat(idle_d, d, t_id) <= 0 then
12:            if o.always_adjust_tasks == false or
               CompareCost(idle_d, d, t_id) <= 0 then
13:                candidates += idle_d
14:    if candidates.size > 0 then
15:        chosen_d = Choose(candidates, t_id, o.pref)
16:    ExecuteTask(t_id, chosen_d)
```

Algorithm 3 Cost-aware Push-based Shuffle

```
1: function EXECUTESTAGE(stage s)
2:     c_tp = GetChildStageTaskPlacement(s)
3:     if s == MapStage then
4:         LocalityAwarePushBasedScheduling(c_tp)
5:     else
6:         tp = GetTaskPlacement(s)
7:         CostAwarePushBasedScheduling(tp, c_tp)
8: function GETCHILDSTAGETASKPLACEMENT(stage s)
9:     c_tp = []
10:    if hasChildStage(s) then
11:        c_s = getChildStage(s)
12:        if IsTaskPlacementAvailable(c_s) == false then
13:            inputs = GetInputs(s, c_s)
14:            c_tp = Evaluate(inputs)
15:            SetTaskPlacement(c_s, c_tp)
16:        else
17:            c_tp = GetTaskPlacement(c_s)
18:    return c_tp
```

chosen, the task can still be assigned to the designated DC (lines 10 ~ 16).

A low $LatThreshold$ value will trigger re-scheduling and additional scheduling overhead to search for idle DCs. But this heuristic is fast and the overhead is much lower than re-evaluating global task placement, hence we set the threshold to a small value (3 seconds by default). We will show how cost-aware task adjustment helps Kimchi remedy the sub-optimal task placement for better cost-aware performance in the presence of dynamics in Section 6.2.

4.3 Cost-aware Push-based Shuffle

Previous works [19], [22], [33] showed that a *push-based shuffle* mechanism can improve performance by pushing intermediate data in the background (asynchronously) in map stages. These systems, however, may encounter a cost-bottleneck because they did not consider data transfer cost heterogeneity to determine the target DCs to push intermediate data.

To improve performance while avoiding the cost-bottleneck, Kimchi adapts the push-based shuffle to use a cost-aware task placement. To push intermediate data, Kimchi needs to determine the child stages' task placement in the parent stages (including the map stages) with imperfect input data information. For example (Figure 2), **TP_1** needs to be evaluated before stages 0 and 1 start. To estimate intermediate data sizes, Kimchi relies on input data information from all parent stages, e.g., input information of stages 0 and 1, with an assumption that intermediate data size is *proportional* to input data size at a configurable rate (R), as done in previous works [22], [26], [33], [38]. While this assumption may not always be accurate, this simple approach yields cost and performance benefits as we will show in Section 6.3. With predicted intermediate data information and other inputs, Kimchi can estimate task placement for a child stage, e.g., **TP_1** for stages 0 and 1.

Algorithm 3 outlines how push-based shuffle works in Kimchi. Before Kimchi executes a stage, Kimchi determines task placement for a child stage (line 2). If the stage has a child stage, Kimchi checks if a task placement for a child stage is available (lines 10 ~ 12). If there is no task placement for the child stage, Kimchi collects the MIP inputs, i.e., *expected* intermediate data sizes, tradeoff preference, and others as before, and runs the optimization (lines 13 and 14). Task placement for the child stage is stored for later use (lines 6, 15, and 17). The tasks in given stage are scheduled based on stage type, i.e., data locality-aware scheduling for map stages and cost-aware scheduling for shuffle stages. In either case, the task placement for a child stage is used for pushing intermediate data in parent stages (lines 3 ~ 7). The pushed intermediate data can serve as a *cache* that prevents repetitive remote access to the same intermediate data in a query, e.g., self-join, that results in both cost reduction and better performance as will be shown in Section 6.3.

Note that activating both the task adjustment (Section 4.2) and the push-based shuffle mechanism can lead to cost inflation due to duplicated intermediate data transfer, i.e., intermediate data pushed may not be accessed if a task is reassigned. In this work, we adjust tasks only when the push-based shuffle is not activated to avoid cost inflation. We plan to explore both options as future work for better cost-aware performance.

5 KIMCHI ARCHITECTURE AND IMPLEMENTATION

We have implemented a Kimchi prototype on Apache Spark (2.2.1) [48]. Thus, applications can submit jobs through the same interfaces provided by Spark to benefit from Kimchi *without modification*. Figure 7 shows the Kimchi architecture. The following components are added and modified:

- The network bandwidth monitor aggregates bandwidth information from executors.
- The input manager stores the application's tradeoff preference and data transfer cost information.
- MapTracker provides the intermediate data size and locations for shuffle stages.
- The task placement optimizer determines task placement based on inputs as explained in Section 3.2.1.
- The task scheduler sends inputs to the task placement optimizer for shuffle stages and assigns shuffle tasks as determined. Note that map tasks are assigned using the locality-aware approach, i.e., where input data is stored. The task scheduler also checks and detects dynamics before it assigns tasks, as explained in Section 4.2.

In our prototype implementation, the task placement optimizer API is implemented with Thrift [14], a remote procedure call framework. The optimizer is written in Python (~ 700 lines of code) and receives inputs in a JSON format [6]. Kimchi uses PuLP [34] to model a task placement problem as a mixed integer programming (MIP) and uses CPLEX [20] to solve the optimization problem. For network bandwidth

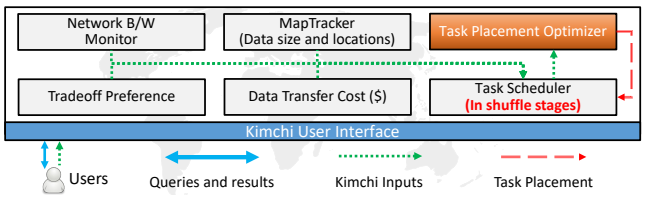


Fig. 7. Kimchi Architecture.

TABLE 7
Kimchi (Spark) property examples

Property Name	Description
spark.kimchi.taskScheduling	Using Kimchi (True or False)
spark.kimchi.costPreference	Desired cost preference (0 ∼ 1)
spark.kimchi.adjustTask	Adjusting task placement (True or False)
spark.kimchi.pushShuffling	Using push-based shuffle (True or False)
spark.kimchi.estimateCurves	Estimating curves (True or False)

and cost information, we modified the executor to estimate the network bandwidth while it fetches data from other DCs and to track the number of remote bytes read by each executor for each origin and destination. This aggregated executor information is sent to the scheduler by piggybacking on heartbeat messages and used as input to solve the placement problem. We also modified the executor to use Wiera [36], a geo-distributed policy driven storage system, to push and fetch intermediate data, as the Spark API does not fully support functions for the push-based shuffle.

Table 7 shows the properties that applications can set to utilize Kimchi easily. To estimate cost-performance curves (Figures 6(a) and 6(b)), applications can set the 'estimateCurve' option to True for the target query. Kimchi estimates the curves using current available inputs including raw input information and the R (explained in Section 4.3) with varying C_pref value *without actual query execution*. Since the R can affect the curves significantly, an incorrect R will result in inaccurate curves that are not useful for decision making. While the fixed R provides moderate benefits as will be shown in Section 6.3, determining the R precisely by predicting query workloads is our future work.

6 Evaluation

Experimental Setting: We deployed and evaluated Kimchi across 8 AWS regions (outbound network cost per GB): US East-Virginia ($0.02), US West-California ($0.02), EU West-Ireland ($0.02), AP SE-Singapore ($0.09), AP SE-2-Sydney ($0.14), AP North-Tokyo ($0.09), AP South-Mumbai ($0.086), and SA East-Sao Paulo ($0.16). We used AWS t2.medium (2 vCPU cores and 4 GB of RAM) for workers and AWS t2.large (2 vCPU cores and 8 GB of RAM) for both the Spark master and the Spark driver for job submissions. For workload, we used TPC-DS [9], a standard decision support benchmark, and TeraSort [8], a standard sorting benchmark, to benchmark the performance of GDA systems. For input data, 40 GB data were evenly distributed and used for TPC-DS queries that produce large intermediate data transfers in shuffle stages. We used 10 GB input data for TeraSort. While 10 GB is relatively small for a GDA, TeraSort produces large intermediate data that is sufficient to show the availability of a richer tradeoff space between cost and performance for different data distributions. We used the Hadoop Distributed File System (HDFS) [4] as an underlying storage system to fetch input data. We used Wiera [36] to push and fetch intermediate data only when the push-based shuffle was activated (Section 6.3). While we mainly used the aforementioned experimental setting for all experiments, we also deployed and evaluated Kimchi on a simulated multi-cloud environment using CloudLab [40] to evaluate Kimchi for a large data set (Section 6.1).

We used different baseline approaches: 1) data locality-aware (vanilla Spark); 2) centralized (minimized network usage); and 3) bandwidth-aware, e.g., Iridium. The centralized approach minimizes network usage for each stage without considering bandwidth and cost heterogeneity; the centralized DC has the largest portion of intermediate data. Note, the centralized approach does not necessarily result in the overall minimized data transfer size, as the centralized DC is determined independently for each shuffle stage. The Iridium approach is equivalent to the $C\_pref = 1$ without task adjustment, which does not consider data transfer cost but only WAN bandwidth for performance reasons. All experimental results are an average of 10 runs, plotted with 95% confidence intervals. The option to handle dynamics is deactivated if not mentioned for comparison purposes.

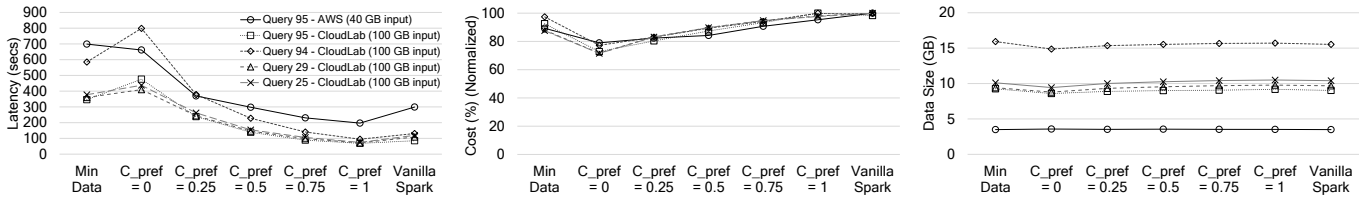
Overhead of Solving The Task Placement Problem: Though MIP is not efficient, the time for computing task placement has a negligible impact on the overall performance, as Kimchi can get a feasible integer solution to be within 5% of optimal for each stage in approximately 0.5 seconds with AWS t2.medium for 8 DC locations setting. We will discuss the scalability of Kimchi in Section 6.5.

6.1 Cost and Performance Comparison

In this experiment, we show the benefit of the cost-aware task placement for task scheduling in simulated (CloudLab) and real (AWS) WAN environments. For a simulated environment, we use 1 node for the master and 8 nodes for workers. Each node has 20 CPU cores and 60 GB of RAM. We throttle the network bandwidths between nodes using TC [7] based on measured bandwidths between 8 AWS DCs shown in Figure 1 to mimic a WAN environment. We use 100 GB data for the simulated environment and data are evenly distributed across 8 worker nodes.

We compare the query execution time and data transfer cost using different baseline approaches and our approach with varying C_pref (0 ∼ 1) for TPC-DS queries (query 25, 29, 64, 94, and 95), consisting of several shuffle stages with large intermediate data, i.e., a bandwidth-intensive workload. Note, we use only query 95 for experiments on AWS due to budget constraints. The $C\_pref = 0$ case can be considered another centralized approach, as all tasks are assigned to a single DC, but it considers cost heterogeneity, i.e., cost-aware centralized.

Figure 8(a) and Figure 8(b) show the query execution time and the cost comparison respectively for the TPC-DS queries. For all queries, the results show that the $C\_pref = 1$ case provides the best performance due to network bandwidth-aware task placement, and that the $C\_pref = 0$ case provides the cheapest cost due to cost-aware task placement. While we observed slightly different performance and cost based on data distributions, we could see a similar



(a) Query latency comparison. (b) Cost comparison. (c) Data transfer size comparison.

Fig. 8. Query latency, cost, and data transfer size comparisons for TPC-DS queries in a real (AWS) cloud and a simulated (CloudLab) cloud. Costs are normalized to the case that incurs the maximum cost.

pattern of results. We will show how data distribution affects overall performance and cost in Section 6.4.

The results show that the C_pref = 1 case (the Iridium approach) does not significantly improve performance compared to the vanilla Spark case. This is because of "multi-waved" execution caused by finite compute resources [13], [29] in our experiments, with which a small fraction of tasks can be executed simultaneously. This allows the vanilla Spark scheduler to naturally avoid assigning tasks to the network bottleneck DCs by giving up data locality after user-specific timeout (3 seconds by default). Infinite compute resources may make the vanilla Spark scheduler assign a large fraction of tasks to the network bottleneck DCs, which causes significant performance bottlenecks as shown in the previous work [38]. Interestingly, the C_pref = 1 case can provide cheaper (up to 11%) cost compared to the vanilla Spark case, except query 95, on the simulated cloud even with better performance, because *Max cost* is fixed as a cost constraint in C_pref = 1, i.e., cost-aware scheduling in Kimchi. The C_pref = 0.5 and C_pref = 0.75 cases show 5 ∼ 22% cost reduction without impacting performance compared to the vanilla Spark case. For the centralized approaches, the minimized data transfer and the C_pref = 0 case, the query latency increases due to network contention as a result of data being sent to a single centralized DC. Lastly, the results show that the query latency decreases as cost increases.

Figure 8(c) shows the data transfer size for each approach. Interestingly, we cannot see any clear relationship with cost. However, the results clearly show that *reducing data transfer size does not necessarily lead to cost reduction*. That is, the centralized approach incurs more cost than C_pref = 0.25 and C_pref = 0.5 cases even with the smaller data transfer size. In addition, the data size difference between the C_pref = 0 and centralized approach is -2.5 ∼ 7.7 %, but the cost difference is 12.8 ∼ 35.1%. This confirms that the cost heterogeneity can significantly affect overall cost. The vanilla Spark case shows a similar data transfer size to the minimized network usage approach due to its data locality-awareness but with 2.6 ∼ 12.3% more cost. These results agree with the results in Section 2.2.

Overall, these experimental results shows that *Kimchi can reduce query execution time without impacting cost compared to other approaches and can allow applications to explore richer cost-performance tradeoff options*.

6.2 Cost-aware Task Adjustment

As explained in Section 4.2, the task placement for a stage may be sub-optimal due to mis-estimated bandwidth or become sub-optimal during query execution due to dynamics, e.g., network contention and bandwidth fluctuation. In this section, we show the benefits of cost-aware task adjustment.

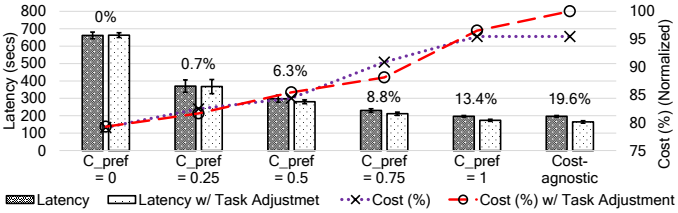


Fig. 9. Cost and performance comparison by adjusting task placement. The percentage numbers show the performance improvement.

We use the same AWS setting and query 95 used in Section 6.1, but we activate the option to adjust task with cost-awareness to change task placement at query execution time. With this option, tasks will be re-assigned only if doing so does not incur additional cost, i.e., cost-aware task adjustment. Additionally, we set the *always_adjust_tasks* option to be activated to allow the scheduler to re-assign tasks without a consideration of cost for the C_pref = 1 case, i.e., cost-agnostic task adjustment. To be responsive to dynamics, we set *LatThreshold* explained in Section 4.2 to 1 second.

6.2.1 Task Adjustment at Run-time

Figure 9 shows the cost and performance comparison with the results of Section 6.1 for each C_pref value. In terms of performance, adjusting task placement helps reduce query execution time up to 19.6% compared to not adjusting task placement. The results show that adjusting task placement improves performance for large C_pref values, e.g., 1 and 0.75. But it does not improve performance for small C_pref values, e.g., 0.25 and 0. This is because, tasks are assigned to a few specific DCs to minimize cost with a small C_pref value as shown in Section 3.3. Thus, the scheduler has less opportunity to find DCs that offer the same or cheaper cost for re-assigning tasks. For example, if C_pref value is set to 0, tasks cannot be re-assigned even if there are available (idle) DCs because none of them can provide a cheaper cost.

For cost, little difference was observed because tasks are re-assigned only when doing so does not incur additional cost, i.e., cost-aware task adjustment. However, for the C_pref = 1 case with cost-agnostic task adjustment, the cost increases by 5% for 6.2% performance improvement. This is because task re-assignment does not consider cost but only network bandwidth, i.e., tasks are always re-assigned to idle DCs as long as performance improvement is expected.

Comparison with baselines: The C_pref = 1 with cost-agnostic task adjustment case incurs a similar cost to the

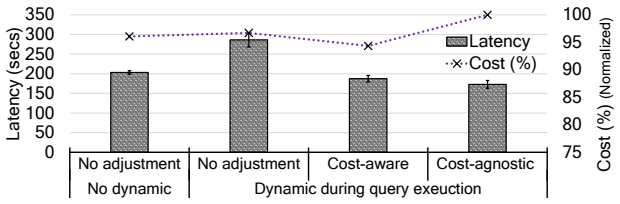


Fig. 10. Cost and performance comparison when one of DCs becomes a bottleneck. For all cases, C_pref is set to 1.

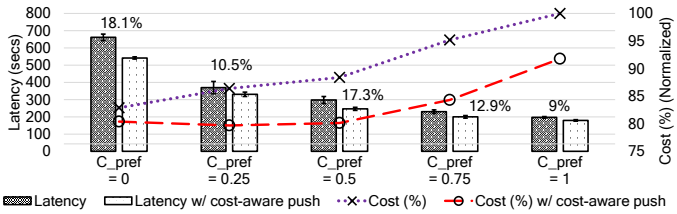


Fig. 11. Cost and performance comparison using push-based shuffle. The percentage numbers show the performance improvement.

vanilla Spark case, but it provides 45% performance improvement. The C_pref = 0.75 case provides 70% performance improvement without impacting costs compared to the centralized case. The C_pref = 1 with cost-aware task adjustment case improves performance by 13.4% without additional cost compared to the static approach (Iridium).

6.2.2 Handling Network Bandwidth Change

In this section, we show the benefit provided by Kimchi in the face of significant dynamics during query execution. While the same experimental setting is used, we randomly throttle one of the links between DCs (3 Mbit/s) using Linux Traffic Control [7] during query execution. Note that we could observe lower network bandwidth than 3 Mbit/s during our experiment even between AWS DCs, e.g., from SA East to AP SE-2.

Figure 10 shows the cost and performance comparison with a static approach (the two left-most cases in the figure), e.g., Iridium, and our non-static approach, both cost-aware and cost-agnostic task adjustments. In terms of performance, the figure shows that the injected network throttling causes 41% additional latency in the static approach as it does not consider dynamics during query execution. The cost-aware task adjustment case shows that it can improve performance by 35% compared to the static approach in the presence of dynamics. In addition, the cost-agnostic task adjustment case can further improve performance (40%) with additional cost. Interestingly, the cost-aware case task adjustment provides better performance than the static approach without dynamic. This is because Kimchi uses the network efficiently by adjusting tasks as shown in the previous experiment (Section 6.2.1).

For cost, the static approach shows a similar cost regardless of dynamics as expected. That is, the tasks are assigned as specified in task placement even in the presence of dynamics. The cost-aware case reduces cost by 2% compared to the static case. This is because tasks are re-assigned to other DCs only when doing so improves performance without additional (or with less) cost in the cost-aware case (read this again). Cost-agnostic case improves performance by 8% at 5% additional cost compared to the cost-aware case.

The results show that any single link that becomes a bottleneck during query execution can significantly affect overall performance, thus needs to be considered. Kimchi can adjust task placement to handle dynamics, e.g., network contention and network fluctuation during query execution time, to achieve better cost-aware performance.

6.3 Cost-aware Push-based Shuffle

In this section, we show benefits of cost-aware push-based shuffle in terms of both cost and performance as explained in Section 4.3. In this experiment, we use the same setting and environment used in Section 6.1, but we set the push-based shuffle to be activated. We set the R value explained in Section 4.3 to 0.8, e.g., 8GB intermediate data will be given with 10GB input data. Kimchi estimates the task placement of child stages with this *expected* intermediate data size and locations before executing the parent stages.

TABLE 8
Data and cost savings by accessing local pushed intermediate data.

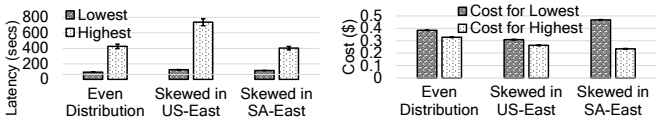
C_pref	1	0.75	0.5	0.25	0
Data Savings	192MB	249MB	257MB	290MB	379MB
Cost Savings	8.2%	10.9%	8.3%	6.7%	2.5%

Figure 11 shows the cost and performance comparison with the results of Section 6.1 for each C_pref value. In terms of performance, push-based shuffle reduces query execution time by 9% $\sim$ 18.1%. The results show that we could get moderate performance improvement even with a simple assumption (fixed R). We believe even further performance improvement is possible with more precise estimation using historical information and recent machine learning techniques.

For cost, we would expect to see similar cost regardless of using push-based or barrier-based shuffle. Yet, the figure shows that the cost can be reduced by 2.5% $\sim$ 10.9%. This is because the same intermediate data is accessed several times within a query, e.g., self-join. By using a pushed-based mechanism, repetitive remote access can be avoided, leading to both cost reduction and performance improvement as presented in Section 4.3.

Table 8 shows data savings by accessing pushed intermediate data locally and their corresponding cost savings. The table shows a trend in which saved data transfer size increases as C_pref value approaches 0. This is because a small number of DCs are chosen for shuffle stages with a small C_pref value, and this leads to a greater amount of intermediate data fetched remotely. The table also shows that more data savings does not necessarily lead to more cost savings. This is because DCs chosen with a small C_pref value have greater opportunity to fetch data from DCs that have cheaper data transfer costs.

Comparison with baselines: The result shows that the C_pref = 0.5 with the push-based shuffle case reduces cost by 23.6% without impacting query performance compared to the vanilla Spark case. The C_pref = 0 case shows a 14% cost reduction without impacting performance compared to the centralized approach. The C_pref = 1 with cost-aware push case improves performance by 9% compared to the C_pref = 1 without push case with reduced cost (8%).



(a) Latency range. (b) Cost range.

Fig. 12. The lowest and highest latencies and cost for each latency with varying data distribution.

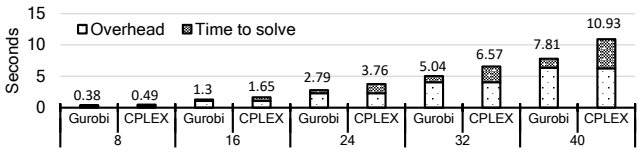


Fig. 13. Time for modeling and solving task placement problem with a varying number of DCs.

6.4 Impact of Varying Data Distribution

In this section, we show the tradeoffs between cost and performance based on different data distributions of intermediate data. We use the same setting and environment as in previous sections. For simplicity, we use TeraSort to clearly show the tradeoffs for a single shuffle stage. For input data, we use 10GB and distribute data in three different ways: 1) even distribution, 2) 1/3 data stored in US East and 2/3 data evenly distributed in the rest of the DCs, and 3) 1/3 data stored in SA East and 2/3 data evenly distributed in the rest of the DCs. Note, US East has the cheapest data transfer cost ($0.02/GB), while SA East has the most expensive data transfer cost ($0.16/GB).

Figure 12(a) shows the lowest latency (C_pref = 1 with a cost-agnostic task adjustment case) and the highest latency (C_pref = 0 case), and Figure 12(b) shows corresponding cost for each latency with different data distribution. Note, we could see similar pattern of results shown in TPC-DS with different C_pref values and options in this experiment, but omitted for space reasons. For the first case (even distribution), we can see 78.5% performance improvement (lowest latency/highest latency) and 15% cost reduction (lowest cost/highest cost) opportunities and this agrees with our results shown in Section 3.3. For the second case (1/3 input data in US East), skewed data increases the highest latency significantly, which opens up a 83.2% performance improvement opportunity without reducing the cost reduction opportunity (15%). This is because a large portion of intermediate data in US-East ($0.02/GB) needs to be sent to another centralized DC that has a more expensive network cost, to minimize cost due to data transfer cost heterogeneity. In this case, applications may be willing to spend additional cost for greater performance improvement compared to an even data distribution. For the last case (1/3 input data in SA East), skewed data increases cost reduction opportunities (50%) with similar performance improvement opportunities with even distribution. This is because a large portion of intermediate data in SA East ($0.16/GB) needs to be migrated to other DCs to improve performance, which causes a cost increase. In this case, applications may be willing to bear additional latency for more cost reduction compared to other cases.

The results show that different data distributions open up a diverse tradeoff landscape, and thus a GDA should consider the nature of the data distribution in order to meet desired tradeoff preferences.

6.5 Scalability of Kimchi

We next conduct experiments to evaluate the scalability of Kimchi using AWS t2.large to show how quickly Kimchi estimates task placement with a varying number of DCs. We used 8 DC setting inputs naturally generated during our experiments and synthetic inputs generated by using 8 DC inputs for other numbers of DC cases. In synthetic inputs, the intermediate data size for each DC is the same, and thus overall data size is increased as the number of DCs increases. For performance comparison purposes, we used two popular MIP solvers, CPLEX [20] and Gurobi [24]. We observed that both solvers result in similar patterns of results, i.e., similar numbers of tasks for each DC, while task IDs are varying.

Figure 13 shows time for solving a task placement problem with a varying number of DCs. Note that the overhead includes time for solving two sub-problems and modeling the final problem explained in Section 3.2.2. The results show that time for solving task placement problems increases as the number of DCs increases, i.e., problem inputs become larger, and that the overhead has a negligible impact on overall execution time, e.g., $< 1\%$ of the running time for the 8 DCs setting. Note that overall execution time will be increased as the number of DCs is increased due to additional overhead such as network contention.

Overall, these results show that Kimchi is able to solve task placement problems with tolerable performance overhead, and thus not a performance bottleneck. Note, if a latency for solving MIP becomes a performance bottleneck, using LP (linear programming) [29] and adjusting diverse parameter options offered by the solvers can be alternative approaches to find feasible solutions within a given amount of time, which we will not discuss in detail in this paper.

7 Discussion for Compute Resource Cost

While we mainly consider heterogeneous data transfer cost, compute cost may take up a large portion of overall costs based on query execution time. Compute costs are also heterogeneous based on DC locations, e.g., $0.00001288/second in AWS US East and $0.000020667/second in AWS SA East for AWS t2.medium. In this section, we discuss how such heterogeneous compute costs affect overall costs by applying different compute cost models to our experimental results using the query 95 (AWS) case shown in Section 6.1.

Many popular data analytics frameworks, e.g., Spark [48], Hadoop [25], and Flink [17], rely on resource managers (RMs) such as Mesos [27] and Yarn [41] to share compute resources across frameworks. In our system models, compute resources can be idle for the lower C_pref values, e.g., most VM instances except those running on the centralized DC will be idle in the C_pref = 0 case. These idle compute resources can be shared by RMs for other frameworks and thus are free of charge for Kimchi. If we apply this compute cost model to the results, we find that compute costs take 6.1 $\sim$ 9.2% of the overall cost based on C_pref values, which

yields similar patterns of results, i.e., data transfer cost is dominant. Note that we also consider external storage such as AWS S3 and its cost for sharing intermediate data among DCs without VM instances to apply this cost model, while storage cost is negligibly small, e.g., $< 1\%$ of overall cost.

If we consider an environment where the compute resources are not shared and idle resources are charged, we find that only considering heterogeneous data transfer costs does not result in cost reduction for lower C_pref values, e.g., the C_pref=0 case results in 8% more cost compared to the C_pref=1 case because idle VM instances' cost takes up large potion of overall costs in the environment. This result shows that exploiting both heterogeneous data transfer cost and compute cost together would open an additional cost-performance tradeoff when compute resources are not shared, as our future work. To avoid the cost of idle resources, a newly emerging compute resource, serverless, would be appealing as it charges only when the code is executed, i.e., pure pay-as-you go, while providing other benefits such as scalability and agility. We plan to exploit serverless in Kimchi to meet desired cost-performance goals.

8 Related Work

Geo-distributed Data Analytics: Many GDA systems have been proposed [26], [29], [31], [38], [43], [44] to overcome the limitations of the WAN. Another set of previous works [19], [22], [33] introduced the push-based shuffle to reduce query latency. While recent works [21], [28] showed that the cost of WAN bandwidth makes up a significant fraction of the overall cost, no existing GDA has considered heterogeneous data transfer cost. To the best of our knowledge, Kimchi is the first GDA that considers heterogeneous data transfer cost to avoid a cost-bottleneck in a multi-cloud environment.

High-bandwidth WAN: In scientific collaboration environments, WAN bandwidth may not be a performance bottleneck by using dedicated (reserved) network (up to 100 Gbps) between DCs as shown in [35], [46]. Cloud providers offer similar dedicated network connection services, e.g., AWS Direct Connect [1] and Azure ExpressRoute [2]. These services, however, are for users who have physical network links and want to increase WAN bandwidth between their premises, e.g., a private DC, and one of cloud providers' DCs. That is, these services are not available to improve WAN bandwidth between cloud providers' DCs located in different regions, which is considered in this work.

Tradeoff between Cost and Performance: Recent GDA systems [29], [38] have offered a knob with which applications can tradeoff the WAN usage and query latency, similar to *C_pref* in Kimchi. These systems, however, may not achieve the desired tradeoff due to the cost-agnostic approach, i.e., *reduced data transfer size does not necessarily lead to reduced data transfer cost*, as we have shown throughout this paper. Tetrium [29] considered compute resource heterogeneity but not monetary cost. We plan to extend Kimchi to consider compute capacities and costs for achieving cost-performance goals.

Handling Dynamics: Most GDA systems [26], [31], [38], [43], [44] do not handle dynamics during query execution. While Tetrium [29] considered dynamics, it can encounter cost- and performance-bottlenecks due to the cost-agnostic approach and the overhead of re-evaluating global task placement. While Kimchi adjusts pending tasks to handle dynamics, tasks running in bottleneck DCs can be detected and re-assigned to other DCs as done in Lube [45].

Scalable GDA System: Kimchi focuses on exploring the tradeoff space given VM instances deployed by other resource configuration systems [11], [30]. While resource configurations determined by these systems may work well for recurring workloads, any workload changes may result in sub-optimal configurations and thus performance degradation. To avoid both performance- and cost-bottlenecks, we plan to make Kimchi scalable to handle workload changes.

9 Conclusion

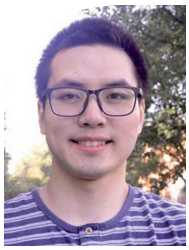
In this paper, we show that data transfer cost can significantly affect overall operational costs for geo-distributed data analytics (GDA), and thus should be considered. We present Kimchi, a GDA system that determines task placement with consideration of data transfer costs, network bandwidth, data size and locations, and applications' desired cost-performance preference. Kimchi improves query performance without additional costs by a cost-aware task placement, a cost-aware task adjustment, and a cost-aware push-based mechanism. Experimental results on AWS show that Kimchi enables applications to reduce costs without impacting performance, improve performance without impacting costs, and enable users to explore a richer cost-performance tradeoff space given different data distributions in a multi-cloud environment.

References

[1] AWS Direct Connect. https://aws.amazon.com/directconnect.
[2] Azure ExpressRoute. https://bit.ly/3kNOgSh.
[3] facebook Engineering. https://bit.ly/3f66NV8.
[4] Hadoop Distributed File System. http://hadoop.apache.org.
[5] iPerf3. https://iperf.fr.
[6] JSON. https://www.json.org.
[7] Linux Traffic Control. http://lartc.org/manpages/tc.txt.
[8] TeraSort. http://sortbenchmark.org/YahooHadoop.pdf.
[9] TPC-DS. http://www.tpc.org/tpcds.
[10] S. Agarwal et al. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX.
[11] O. Alipourfard et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, Berkeley, CA, USA, 2017. USENIX.
[12] G. Ananthanarayanan et al. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, Berkeley, CA, USA, 2011. USENIX.
[13] G. Ananthanarayanan et al. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX.
[14] Apache Thrift. https://thrift.apache.org.
[15] AWS Pricing. http://aws.amazon.com/ec2/pricing.
[16] Azure Pricing. https://bit.ly/3kHGV2O.
[17] P. Carbone et al. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.
[18] M. Cardosa et al. Exploring mapreduce efficiency with highly-distributed data. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 27–34, New York, NY, USA, 2011. ACM.

[19] T. Condie et al. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX.
[20] CPLEX Optimizer. https://goo.gl/6BKOZY.
[21] A. Greenberg et al. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, Dec. 2008.
[22] Y. Guo et al. ishuffle: Improving hadoop performance with shuffle-on-write. In *Proceedings of the 10th International Conference on Autonomic Computing*, pages 107–117, San Jose, CA, 2013. USENIX.
[23] A. Gupta et al. Mesa: A geo-replicated online data warehouse for google's advertising system. *Commun. ACM*, 59(7):117–125, June 2016.
[24] GUROBI Optimizer. http://www.gurobi.com.
[25] Hadoop. http://hadoop.apache.org.
[26] B. Heintz et al. End-to-end optimization for geo-distributed mapreduce. *IEEE Transactions on Cloud Computing*, 4(3):293–306, 7 2016.
[27] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, Mar. 2011. USENIX.
[28] K. Hsieh et al. Gaia: Geo-distributed machine learning approaching lan speeds. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 629–647, Berkeley, CA, USA, 2017. USENIX.
[29] C.-C. Hung et al. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 12:1–12:16, New York, NY, USA, 2018. ACM.
[30] A. Klimovic et al. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference*, pages 759–773, Boston, MA, 2018. USENIX.
[31] K. Kloudas et al. Pixida: Optimizing data parallel jobs in wide-area data analytics. *Proc. VLDB Endow.*, 9(2):72–83, Oct. 2015.
[32] F. Lai, M. Chowdhury, and H. Madhyastha. To relay or not to relay for inter-cloud transfers? In *10th USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud 18, Boston, MA, 2018. USENIX.
[33] S. Liu, H. Wang, and B. Li. Optimizing shuffle in wide-area data analytics. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 560–571, 2017.
[34] S. Mitchell, othersStuart Mitchell Consulting, and I. Dunning. Pulp: A linear programming toolkit for python, 2011.
[35] I. Monga et al. SDN for End-to-End Networked Science at the Exascale (SENSE). In *2018 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*, pages 33–44, 2018.
[36] K. Oh, A. Chandra, and J. Weissman. Wiera: Towards flexible multi-tiered geo-distributed cloud storage instances. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 165–176, New York, NY, USA, 2016. ACM.
[37] P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 49–, Washington, DC, USA, 2006. IEEE Computer Society.
[38] Q. Pu et al. Low latency geo-distributed data analytics. *SIGCOMM Comput. Commun. Rev.*, 45(4):421–434, Aug. 2015.
[39] A. Rabkin et al. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Berkeley, CA, USA, 2014. USENIX.
[40] R. Ricci, E. Eide, and the CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login:*, 39(6):36–38, Dec. 2014.
[41] V. K. Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. ACM.
[42] S. Venkataraman et al. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 301–316, Berkeley, CA, USA, 2014. USENIX.
[43] R. Viswanathan et al. CLARINET: wan-aware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 435–450, 2016.
[44] A. Vulimiri et al. Global analytics in the face of bandwidth and regulatory constraints. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 323–336, Berkeley, CA, USA, 2015. USENIX.
[45] H. Wang and B. Li. Lube: Mitigating bottlenecks in wide area data analytics. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, Santa Clara, CA, 2017. USENIX.
[46] Q. Xiang et al. Toward fine-grained, privacy-preserving, efficient multi-domain network resource discovery. *IEEE Journal on Selected Areas in Communications*, 37(8):1924–1940, 2019.
[47] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.
[48] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX.
[49] B. Zhang et al. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 236–252, New York, NY, USA, 2018. ACM.

Kwangsung Oh is an assistant professor in the Department of Computer Science at the University of Nebraska at Omaha. His research interests include distributed storage systems and cloud computing. He received his B.S. degree in Computer Science and Engineering from Sejong University, Korea, and his M.S. and Ph.D. degrees in Computer Science from the University of Minnesota-Twin cities.



Minmin Zhang is a graduate student in the Department of Computer Science at the University of Nebraska at Omaha. His research interests include distributed systems and cloud computing. He received his B.S. degree in mechatronic engineering from Nanjing Institute of Technology, China.

Abhishek Chandra is a 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 received his B.Tech. degree in Computer Science and Engineering from Indian Institute of Technology Kanpur, and his M.S. and Ph.D. degrees in Computer Science from the University of Massachusetts Amherst.

Jon Weissman 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. Weissman has a Ph.D. in computer science from the University of Virginia.