Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics

Konstantinos Kloudas[‡], Margarida Mamede[†], Nuno Preguiça[†], Rodrigo Rodrigues[‡*]
[†]NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa [‡]INESC-ID / IST, University of Lisbon

ABSTRACT

In the era of global-scale services, big data analytical queries are often required to process datasets that span multiple data centers (DCs). In this setting, cross-DC bandwidth is often the scarcest, most volatile, and/or most expensive resource. However, current widely deployed big data analytics frameworks make no attempt to minimize the traffic traversing these links.

In this paper, we present PIXIDA, a scheduler that aims to minimize data movement across resource constrained links. To achieve this, we introduce a new abstraction called SILO, which is key to modeling PIXIDA's scheduling goals as a graph partitioning problem. Furthermore, we show that existing graph partitioning problem formulations do not map to how big data jobs work, causing their solutions to miss opportunities for avoiding data movement. To address this, we formulate a new graph partitioning problem and propose a novel algorithm to solve it. We integrated PIXIDA in Spark and our experiments show that, when compared to existing schedulers, PIXIDA achieves a significant traffic reduction of up to $\sim 9\times$ on the aforementioned links.

1. INTRODUCTION

In the era of global-scale services and cloud computing, more and more data sets are *naturally* geo-distributed, as the services producing them run on multiple locations. This happens for various reasons. First, many organizations operate multiple data centers (DCs) across the globe, to improve both reliability and user-perceived latency; second, as cloud computing becomes more popular, organizations will split their data processing between in-house and remote cloud nodes; third, some organizations might prefer to use multiple cloud providers to increase reliability and/or decrease costs [20, 5]. This trend is illustrated by recent work in geo-distributed databases [12, 7], which exemplifies some of the challenges that arise in this setting.

Analyzing large data sets in a timely and resource efficient manner is of crucial importance. To this end, over the last decade, a series of frameworks, such as MapReduce [9] and Spark [22], have contributed to "democratizing" big data analytics, by hiding most of the complexity related to machine coordination, communication, scheduling, and fault tolerance. However, all these frameworks assume single-DC deployments, where the bandwidth between different pairs of nodes is uniformly distributed. This is not the case with Wide-Area Data Analytics (WADA), where cross-DC bandwidth is often the scarcest and most volatile resource.

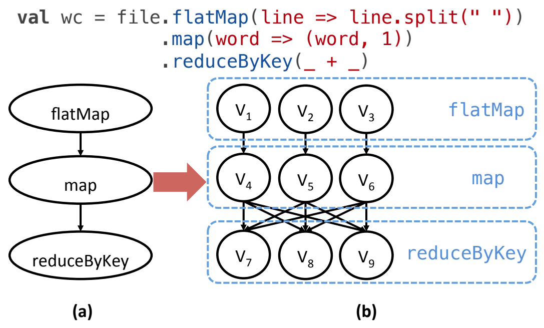
To deal with such jobs, organizations today often copy remote data to a central location, and analyze it locally using relational or Hadoop-based stacks (as reported by other authors [18, 19]). This, not only consumes cross-DC bandwidth proportional to the size of the input data, but it may not even be applicable due to regulatory constraints, such as existing EU regulations that impose data transfer limitations. Very recent proposals have taken preliminary steps to improve WADA execution. Google, reportedly, uses Photon, a system tailored for a particular business-critical job that processes geo-distributed data [4]. However, this is not a general solution, and not all organizations have the resources to build a distributed program for each WADA job. JetStream [16] optimizes WADA execution through adaptive sampling and their data cube abstraction. Although more general, JetStream cannot support all operators, and assumes that applications can support some degree of inaccuracy. Finally, the recent work of Geode [18, 19] provides techniques that are complementary to the ones proposed in this paper, while Iridium [15], although focusing on geo-distributed data processing, has the main objective of minimizing latency instead of bandwidth (Section 8 surveys related work).

In this paper, we make the case for a principled approach to the general problem of conducting data analysis over geo-distributed data sets, in order to make a more judicious use of the available cross-DC bandwidth. To this end, we present PIXIDA, a scheduler for data analytics jobs over geo-distributed data, which minimizes traffic across inter-DC links. Our techniques target cross-DC bandwidth reduction; we make no attempt to directly reduce execution latency.

Our focus on inter-DC bandwidth stems from the fact that the pace at which network capacity increases lags far behind that of the increase of the volume of our digital world. In fact, it has been reported that the 32% network capacity growth in 2013 – 2014 was the smallest of the decade, and the main reason behind this is the cost of adding more capacity [19]. In the same work, the authors report that an analytics service backing one of their applications ingests more than 100TB/day into a centralized analytics stack. Furthermore, and to illustrate the monetary costs incurred by inter-DC transfers, Amazon charges extra for inter-regional transfers, which pass through the open Internet [1]. In fact, for some instances, the cost per GB of transferred data exceeds that of renting an instance for an hour.

*Work done while all the authors were at NOVA LINCS.

Proceedings of the VLDB Endowment, Vol. 9, No. 2
Copyright 2015 VLDB Endowment 2150-8097/15/10.



```
val wc = file.flatMap(line => line.split(" "))
              .map(word => (word, 1))
              .reduceByKey(_ + _)
```

Figure 1: The WordCount example for Spark with its corresponding operator- and task-level job graph ((a) and (b) respectively) for a three-partition input.

In reducing cross-DC transfers, Pixida makes three main contributions:

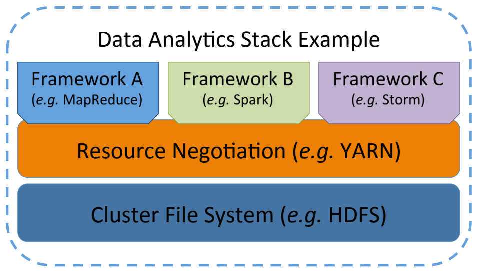
1. First, Pixida introduces Silos, its main topology abstraction. Silos are groups of nodes or, hierarchically, of other Silos that are considered equivalent for executing a given task. Silos simplify the formulation of the problem that the scheduler needs to solve by considering all cross-Silo bandwidth to be equally expensive.

2. The second contribution is the formulation of a new optimization problem that captures the specific aspects of our traffic minimization problem, and the design of a novel, flow-based approximation algorithm for solving it. More precisely, we show that although the problem at hand is similar to the classical Min *k*-Cut problem, the definition of the latter does not map accurately to the problem Pixida needs to solve. That is because Min *k*-Cut fails to capture the cases where the output of an operator is consumed by more than one downstream operator. Our algorithm is based on the Edmonds-Karp algorithm [8], and we formally prove that it has small error bounds.

3. Third, and despite the fact that the design of Pixida is platform agnostic, we integrated Pixida with Spark [22], allowing existing Spark jobs to transparently benefit from the techniques we propose. Our evaluation using both a geo-distributed deployment based on EC2 and a local cluster shows that Pixida significantly reduces cross-DC traffic when compared to alternative solutions. In addition, although none of our techniques targets job latency, our results show that Pixida achieves improved job completion times in most scenarios, with a small penalty in resource-constrained environments (Section 6).

2. PIXIDA IN THE ANALYTICS STACK

When designing Pixida, we set two requirements: i) to make it platform-agnostic, and ii) to avoid breaking the modularity of current data-analytics stacks. To this end, the architecture of Pixida is aligned with the structure of modern data analytics stacks, which separates the framework-specific programming model from scheduling and managing resources, and only modifies the layers that define the locations where computations are scheduled. This is depicted in Figure 2, where a resource negotiation layer stands between the cluster-wide storage system and various different processing frameworks. In our case, we use Spark for our prototype implementation, and we modify its scheduler in a way that is agnostic to the specific API, and also makes no assumptions about the underlying storage system or how data is partitioned or stored in it.



Data Analytics Stack Example
Framework A (e.g. MapReduce)
Framework B (e.g. Spark)
Framework C (e.g. Storm)
Resource Negotiation (e.g. YARN)
Cluster File System (e.g. HDFS)

Figure 2: The typical data analytics stack with a cluster file system at the bottom, a resource manager in the middle, and application frameworks on top.

Given this architecture, Pixida receives as input the result of the compilation of the data-parallel program, which is a graph whose nodes represent data transformations and edges denote data dependencies. For example, Figure 1 presents the `WordCount` example written for Spark [22], and its corresponding graph (code written for other platforms would generate an identical graph). Figure 1 shows that, depending on the level of abstraction, we can consider two different graphs. Figure 1a presents the *operator-level* graph, which includes the dependencies between the operators composing the job. After adding the information concerning i) how the computation of each operator is parallelized into a set of tasks, and ii) the dependencies of each individual task with its parent tasks, we obtain the *task-level* graph, depicted in Figure 1b.

3. OVERVIEW OF PIXIDA

Pixida is a scheduler for data analytics frameworks, designed to minimize the traffic that has to traverse links that are more susceptible to congestion, *i.e.,* the ones between DCs. Jobs submitted to Pixida have their input scattered across different DCs, and request their output to be stored at a specific DC, possibly different from the ones storing the input.

To achieve its goal, Pixida translates this traffic minimization problem into a graph partitioning one, where the job's *task-level* graph (*e.g.,* Figure 1b) is split into partitions. Each of these partitions contains the tasks that are spawned in the same DC (as further detailed in Section 3.2). To construct such graph, Pixida needs to be provided with i) the job's task-level graph, ii) the locations of the input partitions and of the output, and iii) the size of the output of each task in the graph.

The first two pieces of information are easy to determine: the graph is known at compile time, *i.e.,* as soon as the job is submitted for execution, while the locations of the input partitions can be determined from the underlying storage system, *e.g.,* HDFS, and the output location is provided by the user. For the third piece of information, we rely on statistics collected by a *Tracer* phase, which we describe next.

3.1 Tracer

The Tracer allows us to estimate the size of the output of each task. During this phase, Pixida runs the job on a sample of the input partitions (currently set to 20%). We select a subset of the input partitions (*e.g.,* HDFS splits) instead of sampling the input tuples. This allows us to i) select the partitions that are more convenient (*e.g.,* partitions in the same DC, or, in the case of using a public cloud service provider, a subset of the input stored in the in-house infrastructure), and ii) be more resource efficient, as sampling at the tuple level would imply preprocessing the whole input dataset (during the sampling process), thus spawning more tasks.

During its run, the Tracer tracks the size of the output of each task. Tasks are then mapped to their respective operators, such as `map` or `flatMap` in Figure 1, to extrapolate the weights of the outgoing edges of their corresponding nodes in the task-level graph; these weights will then be used by the graph partitioning algorithm. Despite the use of sampling, there is a time and computational overhead to run the Tracer (shown in Section 6.5). However, there are two factors that make this overhead more acceptable. First, the size of the subset of the input is configurable, thus incurring predictable costs. Second, prior work [2, 6, 11] reports that i) 40% of the jobs seen in production clusters are recurring, and ii) the properties of the interplay between code and data, such as operator selectivity (the ratio of output to input size), remain stable across runs for recurring jobs. This implies that the Tracer does not have to be rerun before each execution of the job. Instead, it can be run periodically, or when the runtime statistics gathered during the actual run of the job show that the current execution plan is suboptimal.

In practice, the overhead introduced by the Tracer (Section 6.5) may make it less suitable for ad hoc queries than for recurring ones. Therefore, in the case of ad hoc queries, another strategy for assigning edge weights could be employed, *e.g.,* operator modeling.

3.2 Silos

SILOs are PIXIDA's main topology abstraction. They are groups of nodes that belong to the same location, and therefore our system considers that sending data to a node within the same silo is preferable to sending it to nodes in remote silos. Although in the current version of PIXIDA SILOs correspond to DCs, it is possible to extend our design to allow for nesting, where, for instance, we can have SILOs corresponding to nodes in the same rack, and then racks within the same DC form a higher level SILO. This allows for reducing bandwidth usage both across racks within a DC, and also across DCs. This extension to the design is discussed in Section 7.1. Furthermore, we discuss how to support placement policies and constraints in Section 7.2.

SILOs are of crucial importance for the performance of PIXIDA, as they constitute a sweet spot in the tradeoff between optimal bandwidth minimization and the complexity of its graph partitioning problem. This is due to the following reasons.

First, SILOs simplify our partitioning problem. Cross-SILO transfers in PIXIDA are considered of equal cost, irrespective of the SILOs involved. Sending 1MB of data from a SILO in the West Coast to one in the East Coast is equivalent to sending it to a SILO in Asia. Accounting for more complex cost or latency models would make our partitioning problem more complex, as the cost of each graph edge would depend on the placement of its adjacent nodes.

Second, we reduce the problem statement from assigning tasks to individual nodes, to assigning tasks to SILOs. Reasoning at the granularity of SILOs instead of nodes allows us to transform the task-level graph into a smaller one, as PIXIDA does not have to account for the traffic generated by each node individually, and can instead focus only on inter- versus intra-SILO transfers. This is crucial for scalability, since the complexity of graph partitioning increases fast with i) the size of the graph, *and* ii) the number of partitions to compute [17]. Next, we explain how we transform a task-level graph into its SILO-level form.

3.3 Silo-Level Graph

To reduce the size of the graph, PIXIDA leverages the fact that nodes within the same SILO are equivalent in terms of task placement preference. Therefore, instead of considering each of these nodes individually, as in the task-level graph, for tasks of the same operator, for which we know that they will end up in the same SILO,

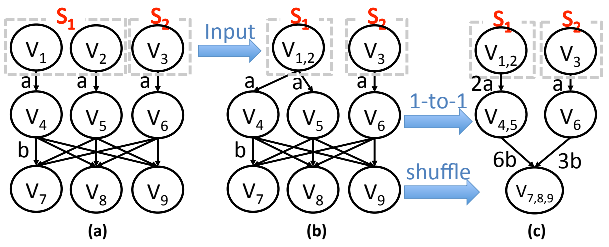


Figure 3: Construction of the SILO-level graph.

we merge their corresponding nodes into a single (super-)node. The weight of the incoming and outgoing edges of the new node will correspond to the aggregate bandwidth of the nodes being merged.

This merging is straightforward for the input: partitions of the data stored on the same SILO are grouped into a single node in the graph. For downstream tasks, this contraction is propagated based on the dependencies between them and their (upstream) parent tasks. In particular, these dependencies take three forms: one-to-one, shuffle, and join-like, and the respective rules for transforming the graph are as follows.

One-to-one. This is the case where, for each task of a given operator, all of its output is consumed by at most one task of the downstream operator, and conversely, each task of the downstream operator receives input from a single task, *e.g.,* `map()`, or `filter()`. In this case, the SILO-level graph is obtained by creating a single node per parent node. This captures the one-to-one relationship across SILOs that is obtained from aggregating the one-to-one relationships across tasks of the same SILO.

Shuffle. This is the case where each downstream task receives part of its input from each parent task, *i.e.,* we have an all-to-all communication pattern, *e.g.,* `reduce()`. In this case, the SILO-level graph contains a single node per downstream operator. This is because, given that each task receives part of its input from each of the parent tasks, traffic is minimized by placing all tasks of the downstream operator in the SILO that holds most of its input[1] (or the output SILO when applicable).

Join. This case includes operators with more than one parent at the *operator level* graph, *e.g.,* `join()`. In this case, there might exist more than one type of dependency, since the join-like operator may have either one-to-one or shuffle dependencies with each of its parents. If it has shuffle dependencies with both parents, then one node is enough for the same reasons stated in the previous point. If there is one one-to-one and one shuffle dependency, then the downstream operator will correspond to as many nodes as the parent with the one-to-one dependency. Each of those nodes aggregates, for the one-to-one parent, the same number of tasks as its parent node, and, for the shuffle parent, all of its tasks. Finally, if it has two one-to-one dependencies, then this implies that both upstream operators are operating on data containing the same key that is being merged, and therefore the data analytics framework applies the same partitioning strategy to both operators, thus assigning them the same number of tasks. However, each upstream operator may partition their tasks into SILOs differently, depending on the initial input locations. In this case, we must allow the output of each task to either stay in the same location or move to the location of the corresponding task of the other upstream operator. As a result, for each node of the first upstream operator in

[1]Note that we assume there is no partition skew, otherwise bandwidth savings might be achieved by splitting the output according to its contents.

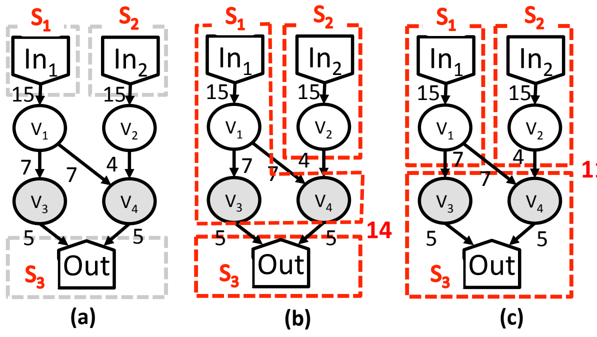


Figure 4: A job with Dataflow Forking, where $e_{1,3}$ and $e_{1,4}$ refer to the same data. Not taking this into account results in the partitioning in b, which is not optimal, as shown by the "cheaper" solution depicted in c.

the SILO-level graph, $p_{1,i}$, we create one child node containing the tasks in the intersection of $p_{1,i}$ with each of the nodes of the other parent (the second upstream operator), $p_{2,i}$. For example, if parent P_1 has two nodes, $p_{1,1}$ with tasks $\{1,2,3\}$ and $p_{1,2}$ with task $\{4\}$, and P_2 has two nodes, $p_{2,1}$ with $\{1,2\}$ and $p_{2,2}$ with $\{3,4\}$, then the child operator C will have three nodes c_1, c_2, and c_3 with tasks $p_{1,1} \cap p_{2,1} = \{1,2\}$, $p_{1,1} \cap p_{2,2} = \{3\}$, and $p_{1,2} \cap p_{2,2} = \{4\}$.

Coming back to the `WordCount` example from Figure 1, we assume that the input partitions of V_1, V_2 are in SILO S_1, while that of V_3 is in S_2, as shown in Figure 3a. Figure 3b shows how the tasks that read the input are merged based on the location of their input, while Figure 3c illustrates how this contraction is propagated to subsequent tasks.

3.4 Task Placement: Strawman Approach

Having the job's SILO-level graph, the locations of the input partitions, and the statistics that determine the weights of the graph, PIXIDA is now ready to assign tasks to SILOs.

A strawman approach to this problem is to augment the SILO-level graph with nodes representing the input and output SILOs, assign weights to its edges based on the statistics from the Tracer phase, and solve the resulting MIN k-CUT problem (Definition 1), with each of the SILOs being one of the terminals to separate.

DEFINITION 1. MIN k-CUT: *Given a weighted graph, $G = (V,E,w), w : E \rightarrow \mathbb{R}$, and a set of k terminals $S = \{s_1,\dots,s_k\} \subseteq V$, find a minimum weight set of edges $E' \subseteq E$ such that removing E' from G separates all terminals.*

Despite the fact that this problem is NP-hard for $k > 2$ [17], there exist a number of approximate solutions. However, it turns out the MIN k-CUT problem does not directly fit the one we are trying to solve, and therefore the solution it finds may miss important opportunities for bandwidth savings. This mismatch happens due to a form of data replication often present in data parallel jobs, which we term *Dataflow Forking*.

Dataflow Forking describes the case where an operator, v, forwards its (entire) output to more than one downstream operator. As an example, in Figure 4 the output of V_1 is consumed by downstream operators V_3 and V_4. In this example, S_1 and S_2 are the input SILOs, while S_3 is where we want the output to be stored.

Simply solving the MIN k-CUT problem in Figure 4a, without taking into account that $e_{1,3}$, $e_{1,4}$ refer to the same data, would result in the partitioning of Figure 4b, with a cost of 14. However, the fact that $e_{1,3}$ and $e_{1,4}$ refer to the same data implies that, if V_3 and V_4 were placed together on a different SILO than V_1, then one cross-SILO transfer (instead of 2), from the SILO of V_1 to that of V_{3-4}, would be enough. In this case, the weight of the edge $e_{1,3}$ should be counted only once. Taking this into account would allow for the partitioning in Figure 4c, which has a cost of 11.

In the next section, we present a variant of the MIN k-CUT problem (and a corresponding solution), which matches our concrete problem statement. Throughout that description, we will refer to a node with more than one child, *e.g.,* V_1 in Figure 4, as a *special* node.

4. GENERALIZED MIN K-CUT PROBLEM

This section formulates a new variant of the MIN k-CUT problem, and presents a novel flow-based *approximation* algorithm to solve it. A more detailed description with proofs of the stated error bounds can be found in a separate technical report [13].

4.1 Background: Solving Min k-Cut

Although the MIN k-CUT problem is NP-hard, an approximate solution can be estimated by computing for every terminal $s_i \in S$, a *minimum isolating cut* that separates s_i from *all* the rest [17]. The latter is the MIN k-CUT problem for $k = 2$, or MIN-CUT problem, and an exact solution can be computed in polynomial time using the Edmonds-Karp algorithm ($O(VE^2)$). The final solution is the union of the $k-1$ cheapest cuts. This algorithm has an approximation ratio of $2 - \frac{2}{k}$ [17].

DEFINITION 2. MINIMUM ISOLATING CUT: *A minimum isolating cut for terminal $s_i \in S$ is a minimum weight set of edges $E_i \subseteq E$ whose removal disconnects s_i from all terminals $s_j \in S \setminus \{s_i\}$.*

To find a minimum isolating cut for a terminal s_i, all remaining terminals $s_j \in S \setminus \{s_i\}$ are connected to a new node, t_i, with edges of infinite weight, and a MIN-CUT for $\{s_i, t_i\}$ is computed.

4.2 Generalized Min-Cut Problem

Using the Minimum Isolating Cuts heuristic allows us to focus on the simpler MIN-CUT problem, but the formulation and solution of the latter need to be adapted to account for Dataflow Forking. In particular, we showed in Section 3.4 that to minimize the data movement in a graph with a single *special* node x with N children, the partitioning algorithm should account *only once* for the cost of transferring x's output (of size w). In contrast, existing algorithms would account for a total cost of up to $N \times w$.

If the set of children of x that *do not* end up in the same partition set as x was known in advance, adding an extra node, e_x, between x and these children would allow us to use MIN-CUT algorithms to optimally solve the problem, by cutting the edge between x and e_x, thus accounting for the corresponding transfer only once. The extra node would be connected to x (upstream of the extra node) and the children of x that are partitioned from x (downstream of the extra node), with edges of the same weight, w, as the ones that it intercepted.

Given that the information about the optimal partitioning of x and its children is not known in advance, an instance of the MIN-CUT problem has to be considered for every possible distribution of the children of x between two sets, as now we are interested in the MIN-CUT problem (where $k = 2$). The solution would then be the one with the smallest cost. This is illustrated in Figure 5, where the isolating cut for In_1 from Figure 4 is computed. For each possible distribution of V_3, V_4, an instance of MIN-CUT is solved (Figure 5b), and the final solution is the cheapest one (Figure 5b(i)).

To formalize the above intuition, the definition of MIN-CUT must be extended to capture the impact of Dataflow Forking. In

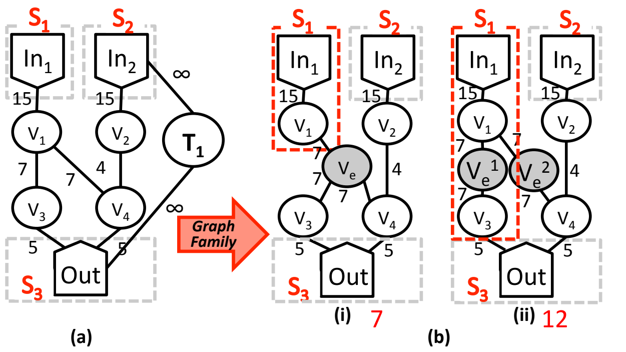


Figure 5: Isolating cut computation for In_1 from Figure 4. Due to Dataflow Forking, the initial DAG (a) gives rise to a family of graphs, with a member for each possible distribution of nodes V_3, V_4. In b(i), V_3, V_4 are kept together in the same set. Edge (V_1, V_e) represents a single cross-SILO transfer of the same data. In b(ii), V_3, V_4 are split into the two sets. An optimal solution is a minimum cut over all members of the family (b(i)). In (b), the second terminal, T_1, is omitted for clarity.

more detail, it has to capture that in the presence of Dataflow Forking, the original DAG of a job, G_o, generates a family of graphs, with an *instance* for each possible partitioning of the children of each *special* node. Let X denote the set of *special* nodes.

DEFINITION 3. INSTANCE*: An instance of $G_o = (V_o, E_o, w_o)$ is a weighted undirected graph $G = (V, E, w)$ obtained from G_o with:*

- *All nodes of the original graph: $V_o \subseteq V$.*
- *All edges whose tail is not a* special *node: $\{(v_1, v_2) \mid (v_1, v_2) \in E_o \wedge v_1 \notin X\} \subseteq E$, with the same weights.*
- *For every* special *node, $x \in X$, whose children in G_o form set Y, one or two* extra *nodes are added to V based on the following:*
  - *One* extra *node, e_x, is added to V if all $y \in Y$ are kept together (in the same set). The new node is connected to x and to all $y \in Y$.*
  - *Two* extra *nodes, e_x^1 and e_x^2, are added to V if the children of x are split according to a partition $\{Y_1, Y_2\}$ of Y. Each* extra *node e_x^k is connected to x and to all $y \in Y_k$ (for $k = 1, 2$).*

  In both cases, the weights of the new edges are equal to that of the edges in G_o connecting x to its children.

Given the above definition, our GENERALIZED MIN-CUT problem can be formulated as follows.

DEFINITION 4. GENERALIZED MIN-CUT*: Given the original graph G_o and two terminals s and t, compute a cut of some instance of G_o that separates s from t and whose cost is minimum over all cuts of all instances of G_o.*

Note that the classical MIN-CUT problem is a special case of the GENERALIZED MIN-CUT, where the graph has no *special* nodes.

From the definitions above, we see that for the optimal solution to the GENERALIZED MIN-CUT problem, the number of MIN-CUT problems to consider increases exponentially with: i) the number of children per *special* node, as the n children of a *special* node have 2^{n-1} possible ways to be split into two sets, *and* ii) the number of *special* nodes in the graph, since for a graph with m *special* nodes, each with n_i children (for $i = 1, \ldots, m$), every distribution of the children of a *special* node has to be combined with all possible distributions of the children of any other *special* node. This results in the following number of instances to be taken into account:

$$\prod_{i=1}^{m} 2^{n_i - 1} = 2^{\sum_{i=1}^{m}(n_i - 1)}. \quad (1)$$

Since this large number can lead to prohibitive latencies, we next introduce an approximation algorithm that solves the GENERALIZED MIN-CUT problem efficiently, without having to compute a MIN-CUT for each instance.

Algorithm 1: Generalized Min-Cut Algorithm.

```
1 $G_{\text{init}} \leftarrow$ getInitialInstance($G_o$);
2 $f_{\text{base}} \leftarrow$ computeBaseFlow($G_{init}$);
3 $G_{\text{final}} \leftarrow$ getFinalInstance($G_{init}$, $f_{base}$);
4 $f_{\max} \leftarrow$ computeMaxFlow($G_{final}$, $f_{base}$);
5 return getMinCut($G_{final}$, $f_{\max}$);
```

4.2.1 Algorithm

To efficiently solve the GENERALIZED MIN-CUT problem, the main challenge is to discover which instance of the family of graphs leads to a minimum-weight cut across *all* instances. To this end, we propose a novel flow-based approximation algorithm (Algorithm 1), which starts by determining an instance of the family of graphs that is *approximately* the one with the minimum-weight cut (lines 1–3), and then computes a minimal cut for that particular instance (lines 4–5) using the Edmonds-Karp algorithm.

The novelty of our algorithm lies in the way it selects the graph instance to use for the final cut computation. The challenge is to compute, in a single pass, the optimal partitioning of the children of each *special* node in the graph, while ensuring that the weight corresponding to the size of the output of that *special* node contributes *at most once* to the final cut value. Our flow-based algorithm, which is summarized in Algorithm 1, follows the structure of the Edmonds-Karp algorithm with some additional restrictions to guarantee the previous property. The main steps of this algorithm are the following:

`getInitialInstance`: This function creates the initial instance, G_{init}, which is the instance of G_o that has just one *extra* node per *special* node, as shown in Figure 6, with V_e being added between V_1 and its children.

`computeBaseFlow`: This is the main step towards the selection of the instance that gives the solution to GENERALIZED MIN-CUT. Our algorithm builds on notions from flow networks and it closely follows the Edmonds-Karp algorithm for computing a maximum flow (a dual of the minimum cut problem [8]) between source s and sink t. In particular, the graph is seen as a network of water pipes, with specific edge capacities, $c(i, j)$. In our case, edge capacities are the *weights* computed by the Tracer phase. Pipes are considered *bidirectional*, *i.e.,* water can flow both ways, with one direction canceling the other. This means that, if along an edge with capacity $c(i, j)$, u units of flow are sent in one direction, then sending $u' \leq u$ units of flow in the opposite direction results in a total flow of $f(i, j) = u - u'$. As in real water pipes, the total flow along an edge cannot exceed its capacity, while the total flow entering a node (except the source s and the sink t) has to be equal to the one exiting it (*flow conservation*). As such, at each step of the algorithm, the additional new flow that can be sent along an edge cannot exceed its *residual* capacity, which is $c_f(i, j) = c(i, j) - f(i, j)$. More formal definitions can be found in most algorithms textbooks [8].

Algorithm 2: Base and Maximum Flow Algorithms.

```
1 $f(i,j) \leftarrow 0$ for every edge $e(i,j)$;
2 while $(p \leftarrow \texttt{findPath}(G_f,s,t)) \neq \emptyset$ do
3     for $e(i,j) \in p$ do
4         $f(i,j) \leftarrow f(i,j) + c_f(p)$;
5         $f(j,i) \leftarrow f(j,i) - c_f(p)$;
6 return $f$;
```

With these basic notions in place, we can now present the general structure of the algorithm that computes the *Base Flow* in Algorithm 2. This follows the structure of the Edmonds-Karp algorithm. At each iteration, the Base Flow algorithm finds a valid path, p, from s to t (`findPath`). This path is then saturated by sending the maximum flow allowed by the constraints on the edges of p, $c_f(p)$, *i.e.,* the minimum residual capacity across all edges in p. Path discovery proceeds in a breadth-first manner, until no more valid paths between s and t exist. Upon termination, a "cut" consists of the partition $\{A,B\}$ of V, where the A consists of the nodes $v \in V$ for which there is still a valid path from s to v, and $B = V \setminus A$. This cut can be found with an additional graph traversal.

Although the Base Flow and Edmonds-Karp algorithms have the same structure, they differ in which paths are considered valid at each iteration (`findPath`). In Edmonds-Karp, a path is considered valid as long as $c_f(p) > 0$. Such paths are called *Augmenting Paths* (APs). For a path to be valid in Base Flow, it has to have $c_f(p) > 0$, but *also* preserve the invariant that the flows on the edges incident to an *extra* node *cannot* have "opposite up-down directions". Figure 6 depicts the possible directions for the flow that do not break this rule. Intuitively, this invariant guarantees that the base flow "could be" a flow in any instance of the family of graphs (after performing a few minor changes). Therefore, its value does not exceed the cost of any cut of any instance of G_o. Given that these new paths are a restricted form of APs, we call them *Restricted Augmenting Paths* (RAPs).

`getFinalInstance`: Upon termination of `computeBaseFlow`, *i.e.,* when no more RAPs exist between s and t, a cut $\{A,B\}$ is computed in the same way as previously described for the Edmonds-Karp algorithm. Set A consists of the nodes reachable from the source through a valid path, while $B = V \setminus A$. After computing this (intermediate) cut, we form the *final instance*, G_{final} of G_o, *i.e.,* the one that will give us our final cut, by transforming the graph in the following way. For each *special* node, x:

1 If all children of x belong to the same set (A or B), then the *special* node, x, has one *extra* node in G_{final}, as in G_{init}.

2 If its children are split between the two sets, then two *extra* nodes are added, and each one is connected to the children belonging to each set.

Optimality: The final result can be non-optimal in some cases, when there are *special* nodes. The resulting cut is optimal if no AP is found when a maximum flow is computed (line 4), even with *special* nodes. Otherwise, in order to define the bounded error, let w_x be the weight of the edge from a *special* node x to each of its n_x children. The cost of the cut computed by Algorithm 1 does not exceed the cost o of the optimal cut by more than $\sum_{x \in X} w_x$. In contrast, if a MIN-CUT algorithm (*e.g.,* Edmonds-Karp) was applied on the original graph, the cost of the computed cut could reach up to $o + \sum_{x \in X}(n_x - 1)w_x$, which can be significantly higher. A formal proof showing how this error bound is derived is presented in a separate technical report [13].

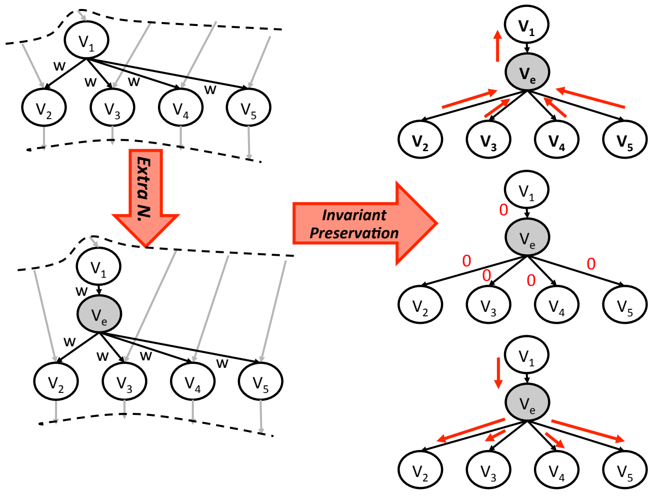


Figure 6: BASE-FLOW ALGORITHM: For each *special* node in the graph, an *extra* node is added and, after each iteration, the total flows of all edges incident to an *extra* node cannot follow opposite up-down directions.

4.3 Generalized Min k-Cut Problem

The problem that PIXIDA has to solve is GENERALIZED MIN k-CUT problem, which differs from the GENERALIZED MIN-CUT problem we analyzed so far. To solve the GENERALIZED MIN k-CUT problem, we apply the Isolating Cuts heuristic (§4.1), and therefore translate the GENERALIZED MIN k-CUT problem to k GENERALIZED MIN-CUT problems. To solve each of these problems, our algorithm initially finds the instance on which to compute the minimum cut, G_{final}. Then, for each problem, its G_{final} is computed independently from the ones selected for the remaining problems, and this may lead to some children of *special* nodes ending up in more than one different partition in the final solution.

To eliminate such overlapping partitions, we apply the following greedy strategy. Suppose that two or more partitions overlap, *e.g.,* $P_i \in P$ where $i \in [m,\ldots,n]$. The overlap is then $G_{over} = \bigcap_{i=m}^{n} P_i \neq \emptyset$. In this case, we compute the cost of removing G_{over} from each of the partitions by summing the weights of the edges with one end in $P_i - G_{over}$ and the other in G_{over}, and we keep G_{over} in the partition where we have to pay the most for removing it, *i.e.,* the one with the minimum cost. A detailed description can be found in a separate technical report [13].

5. PIXIDA ON SPARK

We implemented PIXIDA by extending the code of Spark [22]. This section describes two important design decisions concerning the implementation of PIXIDA as a modified version of the scheduler of Spark.

Stage Pipelining: For performance reasons, Spark organizes operators in *Stages*. Each stage groups together as many consecutive operators with one-to-one dependencies as possible. The boundaries between stages are the shuffle operations, or any cached partitions from previous stages, as would happen in iterative jobs. In the `WordCount` example, the `flatMap` and `map` operators are in the same stage, while `reduceByKey` (shuffle) is in a separate one (Figure 7(b)). Stages are then split into tasks (Figure 7(c)), the basic execution unit, which are executed in parallel. While a stage has to wait for all previous stages (*i.e.,* all their tasks) to be fully executed before it can start its execution, operators within a stage

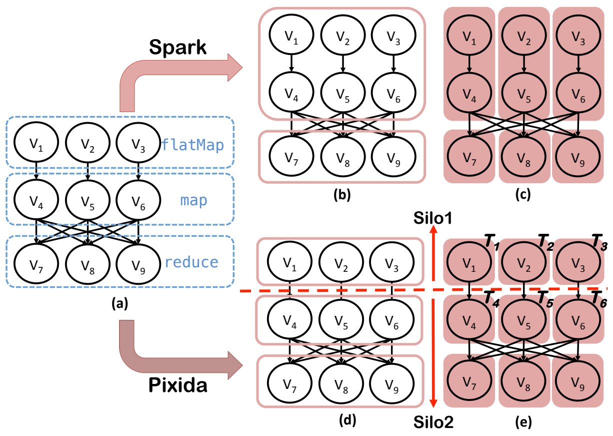


Figure 7: Stage and Task creation in Spark and PIXIDA for WordCount example. In Spark, `map` and `flatMap` are always put in the same Stage, which is not the case for PIXIDA, if the two operators are not placed on the same SILO during graph partitioning.

are pipelined. Pipelining allows for better performance, since otherwise each operator, *e.g.,* `flatMap` in the `WordCount` example, would have to be fully executed (*i.e.,* all its input has been processed), before its following operator, *e.g.,* `map`, can start.

In PIXIDA, we had to introduce the additional constraint that we can only place two operators with a one-to-one dependency in the same stage if they are also placed on the same SILO by the graph partitioning phase. Therefore, this may result in more stages than in Spark, as illustrated in Figure 7(d). There, the graph partitioning phase splits the graph after the `flatMap` operator, causing the `flatMap` and `map` operators to end up in different stages. Given that stages constitute workflow barriers, this may result in additional job completion latency. To reduce these overheads, and only for these cases, PIXIDA introduces *stage pipelining*. This means that for two stages with one-to-one dependencies, as in Figure 7(d), a task of the child stage (*e.g.,* T_4) can start as soon as its parent (T_1) finishes, without waiting for the whole parent stage to finish. This alleviates most of the overheads coming from the creation of additional stages in PIXIDA, as shown in Section 6.4.

SILOs and Delay Scheduling: When assigning tasks to nodes, it may be the case that there are no free resources in the preferred SILO. In these cases, allowing a task to wait indefinitely for resources on the preferred SILO can lead to unacceptable latencies. PIXIDA balances the tension between bandwidth minimization and job completion time using delay scheduling [21]. Under such a scheme, a timer is set when a task is waiting for an available execution slot on its preferred SILO. Upon expiration of a timeout, t_{delay}, currently set to 5 sec, the task is scheduled on the first available execution slot on any SILO.

6. EVALUATION

We evaluated PIXIDA through a series of experiments on Amazon EC2 and a private cluster. The questions we try to answer are:

1. What are the bandwidth savings achieved by PIXIDA? §6.2
2. Does GENERALIZED MIN-CUT lead to a larger cross-SILO traffic reduction than MIN-CUT in the presence of Dataflow Forking? §6.3
3. What is the contribution of each individual optimization? §6.4
4. How do the offline processing steps perform? §6.5

	EU	US-West	US-East
EU	1.02 Gbps	68.4 Mbps	139 Mbps
US-West		1.01 Gbps	145 Mbps
US-East			1.01 Gbps

Table 1: Bandwidth between the different EC2 locations.

6.1 Experimental Setup

For our evaluation, we used two testbeds and two configurations to test a variety of scenarios. We ran all experiments 10 times and report the average and the standard deviation of the results.

Testbeds: We evaluate PIXIDA on EC2 and on a private cluster. The experiments on EC2 show how PIXIDA performs in multi-tenant and geographically distributed environments, while the private cluster provides an isolated and controlled environment. Note that the results on the local cluster are conservative, since all nodes are connected to a single switch. In these settings, there is no bottleneck link, thus the impact of PIXIDA's overheads on the overall performance is more pronounced.

- **EC2:** 16 `m3.xlarge` instances, each with 4 cores and 15 GB of RAM. The machines are located in 3 different regions, namely EU, US-West, and US-East. Table 1 shows the bandwidth between the 3 locations, as measured using `iperf`.
- **Private Cluster:** 6 machines, each with an 8-core AMD Opteron and 16 GB of RAM. All machines are connected to a single 1Gbps switch.

Applications: For our evaluation, we use the following applications, each representing a different class of algorithms. Note that *no modifications* to the application code were required to port them to PIXIDA.

- **WordCount (WC):** represents an "embarrassingly parallel" application. Its code is presented in Figure 1.
- **K-Means (KM):** represents an iterative convergent algorithm. This is an important class of computations, especially in the Machine Learning community.
- **PageRank (PR):** represents an iterative convergent graph processing computation.

Configurations: We further distinguish 2 configurations, which differ on whether the (single) output SILO holds part of the input.

- **Overlapping:** The output SILO holds part of the input, *i.e.,* the input/output SILO sets have an overlap.
- **Disjoint:** The output SILO does not hold any part of the input, *i.e.,* the input/output SILO sets are disjoint.

Assuming 3 SILOs, *A*, *B*, *C*, with *C* being the output SILO, in the *disjoint* case the set of input SILOs will be *A* and *B*, while in *overlapping* it will be *A*, *B*, and *C*.

On EC2, SILOs correspond to regions (*i.e.,* data centers), hence we have 3 SILOs, each consisting of 5 nodes. In the private cluster, we split the nodes into 2 SILOs, one of 3 and one of 2 nodes. In both testbeds, the remaining node is used as the master node of Spark and the NameNode for HDFS.

In all experiments, HDFS was used as the underlying storage system with a block size of 128 MB. In the disjoint case, on EC2 the input is stored in EU and US-West, whereas on the private cluster, the input SILO is the 3-node one.

Inputs: The size of the input of each application on each of the testbeds is presented in Table 2. On EC2, for `WC` we used 10 GB of data from the Wikipedia dump, for `PR` we used the page graph extracted from a *full* snapshot of wikipedia, which accounted for

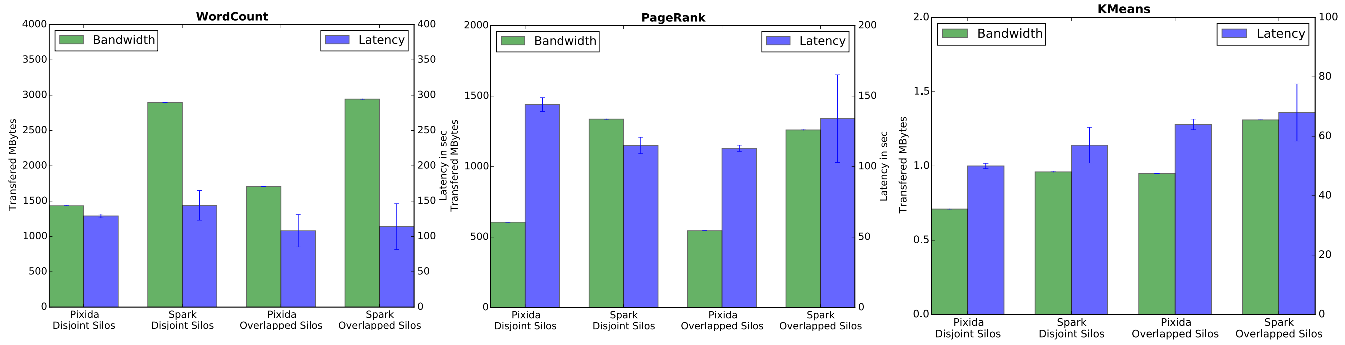


Figure 8: Cross-SILO traffic and latency results of the comparison between PIXIDA and Spark on EC2.

	WC	PR	KM
EC2	10GB	1.1GB	10GB
Private	1GB	250MB	1.5GB

Table 2: Input size of each application for both testbeds.

1.1 GB, and for `KM` we created an artificial dataset of 10 GB of randomly generated vectors of size 50.

On the private cluster, we scaled down the input from the EC2 experiments. Thus, `WC` had 1 GB, `PR` had 250 MB, and `KM` used 1.5 GB of data.

6.2 Comparison to Existing Solutions

6.2.1 Existing Schedulers

To compare PIXIDA against existing schedulers, we run Spark (v. 0.8.0) with and without PIXIDA. For this set of experiments, we run all three applications on both testbeds and for both configurations. The results for bandwidth utilization are presented in Figure 8 and Table 3, for EC2 and the private cluster, respectively.

In addition, and although the goals of PIXIDA do not include optimizing job completion time, the above results also contain the latencies achieved by PIXIDA and Spark. For the latency numbers, we note that the job completion times reported for Spark are *lower bounds* of the actual latency. That is because, for the cases where (part of) the output does not end up in the required output SILO, we do *not* count the time it would take to transfer it there, since this would have to be done outside the Spark framework.

EC2 Results: Starting with EC2, which represents the main target environment for PIXIDA, the results of the comparison between PIXIDA and Spark are presented in Figure 8. The results show that PIXIDA consistently reduces the cross-DC traffic, while maintaining low response times. In particular, in the disjoint configuration, PIXIDA reduces cross-SILO traffic by 51%, 57%, and 26%, for WC, PR, and KM respectively, while for the overlapping one, this becomes 42%, 57%, and 27%.

Focusing on the iterative jobs, *i.e.,* PR and KM, PIXIDA tries to confine iterations to a single SILO. But even when this is not possible, *e.g.,* when there is a `join` between data from more than one SILO, PIXIDA transfers data when its volume (in the data flow) is minimized. This is the reason behind the significant traffic reduction achieved, since the data transferred during and across iterations often has a large volume, which is multiplied by the number of iterations. WC differs in that it is not iterative and consists of only 2

	PIXIDA			Spark		
	WC	**PR**	**KM**	**WC**	**PR**	**KM**
Disjoint Input/Output Silos						
Traffic(MB)	185	34	0.006	248	249	0.052
Latency(s)	51	154	62	44	148	62
Overlapping Input/Output Silos						
Traffic(MB)	213	179	0.039	404	249	0.043
Latency(s)	52	150	21	44	144	21

Table 3: Cross-SILO traffic and latency of the comparison between PIXIDA and Spark on the private cluster.

stages. In this case, PIXIDA transfers data only once from the input to the output SILO, *i.e.,* during the shuffle between these stages. This is not the case for Spark, where in all our experiments, apart from the shuffle that affects all SILOs, most of the final output is stored in a different SILO from the target location, and thus an additional transfer of the final output to the destination SILO would be required.

Regarding latency, we see that PIXIDA's reduction of the traffic that traverses the high-latency cross-DC links leads to better job response times despite i) the latency of Spark being measured conservatively, as mentioned earlier, and ii) the potentially more stages created by PIXIDA (Section 5).

The exception to this improvement in latency was PR in the disjoint case. This is due to the fact that PIXIDA tries to place tasks wherever cross-SILO traffic is minimized, instead of spreading them uniformly among all nodes in the system. In resource constrained environments, this may lead to increased latency, as tasks may have to wait for execution slots to be freed before being able to run. This is the case in this set of experiments. By analyzing the logs, we found that almost all of the difference (24s out of the 29s) is due to `joins` taking place in each iteration. The stages of these joins involve data from both input SILOs and execute the CPU expensive computation of the new weights of the pages, thus creating tasks that are both CPU and IO intensive. By spreading tasks uniformly among the nodes, Spark has 1 or 0 tasks per node, while PIXIDA puts all the tasks on the `EU` SILO, which has an average of 1.8 tasks per node. This results in an average runtime of a task for this stage of 31s on Spark, and 42s for PIXIDA, which explains the difference. This highlights that in the tradeoff between latency and cross-SILO traffic minimization, PIXIDA chooses traffic minimization. This explanation is fully aligned with the results obtained in the private cluster in Table 3, where the input is scaled

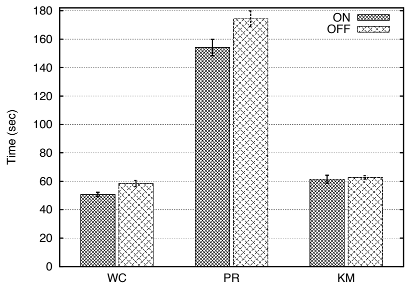


Figure 9: Job Completion times with (ON) and without (OFF) stage pipelining for all 3 applications.

down, and there are enough execution slots for all tasks, thus avoiding contention. In this case, PIXIDA achieves almost the same latency for PR as Spark. If we wanted PIXIDA to be more latency friendly and less aggressive at minimizing cross-SILO bandwidth, delay scheduling with a smaller timeout would amortize the impact of this phenomenon, since the placement constraints are relaxed upon timeout.

Finally, an interesting observation is that PIXIDA manages to decrease the impact of network volatility on the job completion time. By reducing the traffic that has to traverse the "narrow" and bursty cross-DC links (Table 1), which are shared with the rest of the Internet traffic, PIXIDA consistently reduces the standard deviation of the completion times on EC2, as shown in the bars of Figure 8.

Private Cluster Results: Table 3 shows that PIXIDA consistently reduces cross-SILO bandwidth utilization, with results that are mostly comparable to the EC2 case. However, in this case, there is a higher latency cost when using PIXIDA. This was expected since the overheads of PIXIDA are more pronounced, as all nodes are connected to a single switch, and therefore do not have a "narrow" link between them. As such, the latency overheads introduced during our scheduling process are not compensated by the savings brought by avoiding the latency of cross-DC links.

6.2.2 Centralized Solution

As described in Section 1, to process geo-distributed data, organization nowadays initially transfer all the data to a single DC and process them locally. When compared to such a solution, PIXIDA requires from 2 times (for PR) to orders of magnitude (for KM) less bandwidth. This can be seen by comparing the EC2 results for PIXIDA in terms of bandwidth consumption in Figure 8, to the size of the input for each experiment, presented in Table 2.

6.3 Min-Cut vs Generalized Min-Cut

Although some of the graphs of the jobs in Section 6.2 contain *special* nodes, our logs reveal that their final "cuts" do not contain the outgoing edges of these nodes. In other words, both the GENERALIZED MIN-CUT and MIN-CUT algorithms output the same partitions, thus forming identical job scheduling plans, with no difference in the achieved cross-DC bandwidth reduction. To confirm this, we repeated the experiments using the Edmonds-Karp algorithm for graph partitioning and the results for the volume of data transferred were the same as in PIXIDA.

To capture a situation where GENERALIZED MIN-CUT achieves bandwidth savings when compared to MIN-CUT, we developed the Spark application depicted in Figure 10. Even though this example application was developed with the specific goal of highlighting that MIN-CUT may in some cases be sub-optimal, we note that that was inspired by a processing job from a measurement study we are

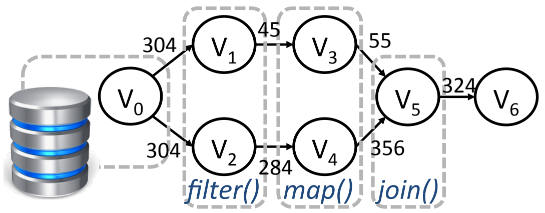


Figure 10: Example illustrating the better fit of the GENERALIZED MIN-CUT against MIN-CUT to our scheduling problem.

conducting for another project. In particular, the job initially reads from a file in HDFS (in `<key, value>` format), and applies two different `filters`, to select two different record subsets. These subsets are later transformed (`map`), *e.g.,* there is a transformation of the key, and then (outer) `joined`, in order to correlate records that refer to the same (new) key. This workflow appears, for instance, when logs from two different dates are collected (using a filter), the key then changes to reflect the event that triggered each record, and finally the records are joined on the new key to detect event correlations.

We ran this application on the local cluster using the *disjoint* configuration. The input consisted of 300+ MB of data, and we scheduled the tasks based on i) our GENERALIZED MIN-CUT algorithm, and ii) the Edmonds-Karp algorithm for the MIN-CUT problem. Edge weights in Figure 10 represent the output of the Tracer for each stage of the application, in megabytes. When partitioning, our algorithm adds an *extra* node, V_e, between V_0 and V_1, V_2, and cuts the edge, $e_{0,e}$ between V_0 and V_e (304 MB), while Edmonds-Karp cuts $e_{5,6}$ (324 MB). In this case, PIXIDA achieves about 8% traffic reduction when compared to a strawman design that applies MIN-CUT[2]. (The job completion latency is approximately the same in this comparison.) Note that these benefits can easily become even more pronounced depending on the characteristics of the analysis. For instance, if the ratio `sizeOf(value)/sizeOf(key)` was larger, then the benefits would also increase. This is because, although common keys between the outputs of V_3 and V_4 are kept only once, the values remain. In addition, it can be argued that as the number and the complexity of the libraries for big data analytics increase, it becomes more likely that some applications will encounter scenarios where PIXIDA is beneficial.

6.4 Design Features of Pixida

We next evaluate the individual design choices of PIXIDA, namely i) the benefits from stage pipelining, and ii) the effects of delay scheduling when computational resources are scarce. The results presented in this section were obtained in the private cluster.

Stage pipelining: To evaluate the contribution of stage pipelining, we ran all applications on the private cluster and measured the average job completion times with pipelining enabled and disabled, for the *disjoint* configuration, where pipelining has a higher impact. The choice of the disjoint configuration highlights the fact that, while the input SILO is executing its tasks, resources at the output SILO are sitting idle, and thus pipelining allows the waiting time on the output SILO to be overlapped with the execution time of the input SILO.

[2]In this example, adding a single *extra* node could also allow the Edmonds-Karp algorithm to find the optimal cut. This is because the *special* node has only two children. For a more complex example of a graph where adding the *extra* node alone does not solve the problem, the reader is referred to [13].

t_{delay}	WC 2GB			WC 4GB		
	Silo (%)	Traffic	t	Silo (%)	Traffic	t
10^2	95.8	501	76	61.4	2049	94
$3 \cdot 10^2$	95.8	501	78	58.4	2049	98
10^3	100	379	84	59.4	2049	99
$3 \cdot 10^3$	100	379	88	58.4	2049	99
$5 \cdot 10^3$	100	379	89	83.3	1482	113
$3 \cdot 10^4$	100	379	95	100	748	152
$6 \cdot 10^4$	100	379	114	100	748	155

Table 4: Percentage of tasks executed on the preferred silo, traffic (MB) and job completion times (s) for different values of the t_{delay} (ms) and the input size for WC.

The results in Figure 9 show that pipelining improves completion times for all applications, with the exception of KM, where there are no benefits. For WC, the reduction is 13.3%, while for PR it is 11.6%. In the case of KM, pipelining offers almost no improvement due to the nature of the job. In this case, only the final transfer is scheduled on the output SILO, and, although this is pipelined with the final stage of the computation, the small size of the output does not enable significant improvements.

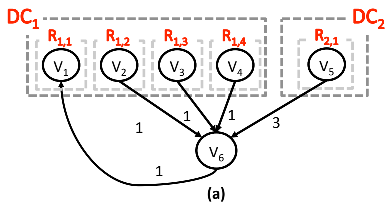
Delay scheduling: To evaluate the impact of delay scheduling in resource-constrained environments, we ran WC in the disjoint configuration with 2 and 4 GB of input, and for different values of the `timeout` (t_{delay}), *i.e.,* the time for a task to wait until it is allowed to be scheduled on a node outside its preferred SILO (Section 5). Table 4 presents the percentage of tasks executed on their preferred SILO (*locally*) and the corresponding cross-SILO traffic and completion time.

The results show that, for the same timeout, as jobs become larger, latency increases and task locality decreases. This is expected since the larger the job the higher the number of tasks that may have to wait for resources to be freed. Furthermore, we see that, for a given input size, as the timeout increases, the percentage of tasks executed on their desired SILO also increases, at the expense of latency. This is due to the fact that tasks are allowed to wait more for resources to be freed on their preferred location, instead of being scheduled on a sub-optimal one (in terms of traffic minimization). Another observation is that although the job completion time increases, the amount of cross-SILO traffic stays the same (up to a certain point). The reason is that: i) we are in the private cluster that has not "narrow" links, and ii) within a SILO, tasks can be placed on a different node than the one that holds (most of) the input of the task, thus leading to additional latency. This is because nodes within a SILO are considered equivalent. Finally, after a t_{delay} of 30 seconds, the difference in cross-SILO traffic for 4 GB is significant. The reason is that 30 seconds is approximately the duration of the slowest stage of the job, as observed in our logs. This leads to all tasks being executed on their preferred SILO.

6.5 Tracer and Partitioning Algorithm

Finally, we evaluate the Tracer and the Partitioning algorithm. In this case, the first thing to note is that in most cases (apart from KM) the Tracer takes more time than the actual application to complete, despite the fact that it processes only 20% of the original input. The reason is that, during the Tracer, each operator runs to completion, before the next operator can execute (*i.e.,* there is a single operator per stage). In particular, the slowdown of the Tracer when compared to the actual job is 2× for WC, 2.7× for PR, whereas for KM we have a speedup corresponding to a Tracer completion time of 0.6×. This difference is due to the fact that, during the execution of the actual job, WC allows for denser stages, *i.e.,* stages composed of more operators, while in KM most stages contain only a single operator. This leads to the structure of the Tracer for KM being closer to that of the actual job (with a single operator per stage). As mentioned, although time consuming, a single run of the Tracer can be amortized over a large number of job executions.

Concerning the Partitioning algorithm, its running time is orders of magnitude lower than the typical time required for a data parallel job, with a value ≤ 0.08 seconds in all of our experiments. In addition, and to compare the running time of our partitioning algorithm to that of Edmonds-Karp for the MIN-CUT problem, we ran the latter on the same graphs as the ones used in Section 6.2. The results show that there was no significant difference in the running times achieved by both of them (the difference was always under 0.01 seconds). This is expected, as, although some of these examples contain *special* nodes, their final "cuts" do not contain any of their outgoing edges. This implies that both algorithms, Edmonds-Karp and ours, perform the same steps, modulo some additional transformations and checks that are not costly.



Rack	Nodes
$R_{1,2}$	V_2
$R_{1,3}$	V_3
$R_{1,4}$	V_4
$R_{2,1}$	V_5, V_6
$R_{1,1}$	V_1

(b)

Figure 11: SILO-nesting and its implications on PIXIDA.

7. DISCUSSION

This section discusses two extensions to the design of PIXIDA, namely i) how it can achieve multi-layer optimization, *e.g.,* both inter-DC and inter-rack (within a DC), and ii) how it can accommodate different placement policies.

7.1 Hierarchical Silos

As previously mentioned, PIXIDA allows SILO nesting, *i.e.,* silos composed of silos. This allows PIXIDA to simultaneously minimize bandwidth usage at multiple levels, *e.g.,* at both the rack and the DC level. To achieve this, we apply delay scheduling at the lower levels of SILOs. For example, in a scenario with DC-SILOs composed of rack-SILOs, a task initially tries to be placed on its preferred node (only for the case of initial tasks, since Spark tries to co-locate them with the input); if it does not succeed within t_{delay} time units, it tries to find a node in the same rack as the preferred node; upon another t_{delay} it tries to find a node in the same DC as the preferred node; and after another t_{delay} time units it is free to choose any node.

The challenge in this case is that, by optimizing data movement in one of the levels, we may not be minimizing the traffic at the other levels. This is illustrated in Figure 11, where we assume rack and DC-level SILOs, and minimizing cross-DC traffic has a higher priority than cross-rack traffic savings. In this example, we have the input of a job consisting of 6 partitions, spread unevenly among 4 racks, $R_{1,2}$, $R_{1,3}$, $R_{1,4}$, and $R_{2,1}$ located in 2 DCs ($DC_{1,2}$), and the output of the job is required to be stored in $R_{1,1}$ in DC_1. Each of the racks in DC_1 holds one partition, while $R_{2,1}$ in DC_2 holds 3. Applying the Isolating Cuts strategy (§ 4.1) using rack-level SILOs and solving each of the 5 resulting MIN-CUT problems would result in the partitioning shown in Figure 11b. This partitioning scheme, although optimal for rack traffic minimization, has a cross-DC cost of 4, while placing V_6 on a rack in DC_1 would have a cost of 3.

To address this, we need to run the optimization algorithm at multiple levels, starting with the higher level in the SILO hierarchy and moving downwards. In the above example, we would initially run the algorithm using DC-level SILOs to guarantee cross-DC traffic minimization. Then, within each partition, we would run the algorithm using rack-level SILOs to determine the preferred racks within each DC.

7.2 Policy Enforcement

PIXIDA is able to accommodate task placement policies and constraints as a natural extension of its graph partitioning problem. Policies for task placement are normally either in the form of i) priority policies, or ii) placement constraints.

Priority Policies: This category includes priority-based schemes, such as strict priority and weighted fair scheduling, among others. Supporting them is the responsibility of the resource negotiator, and is orthogonal to the role of PIXIDA.

Placement Constraints: This category includes jobs that have data placement constraints, for instance due to national regulations, or to internal rules within an organization, which bind some data to specific servers (*e.g.,* financial data must be stored in servers at a specific, protected location within a DC).

To optimize task placement for jobs with such data placement constraints, we make the observation that these constraints concern the storage of the raw data, and not the results of computations over them. This implies that up to some depth in the job graph, all sub-computations involved should be performed at specified locations. After that, we assume that the remaining tasks can be placed in any location, as they operate on intermediate results that are safe to be shipped to different locations (*e.g.,* aggregates). In other cases, *i.e.,* when all the tasks have pre-defined locations, there is no placement flexibility, thus no room for optimization.

With the aforementioned observation in mind, in these cases, the machines with the specific property define a separate SILO, and to specify that a task has to be placed in that specific SILO, it suffices to connect the node that corresponds to the task in the job graph to its preferred SILO with an edge of infinite weight. This transformation guarantees that the "cut" of the graph will never include that edge, resulting in the task being placed on that SILO.

8. RELATED WORK

Several recent systems target efficient wide-area stream processing [4, 23, 3, 16, 14, 18, 19]. With the exception of JetStream [16], Geode [19, 18], and Hourglass [14] (and, to a lesser degree, Iridium [15]), these systems focus on latency and fault-tolerance, and not cross-DC traffic. Geode [19] is the closest related work, since it targets wide area big data analytics. However, the approach of Geode is rather different from our techniques, since it consists of i) pushing computations to edge DCs, ii) caching intermediate results and computing diffs to avoid redundant transfers, and iii) applying an iterative greedy approach to gradually adapt data replication and task placement to minimize cross-DC traffic. Their techniques could be integrated into PIXIDA to allow for further gains and runtime adaptation of the execution plan. In Hourglass [14], the authors assume the use of nodes spread in the open Internet, and propose a decentralized, iterative operator placement algorithm, which assumes *no* global knowledge. This assumption is too pessimistic in our settings, as working with a DC-based deployment allows for some global knowledge that can be leveraged to simplify the algorithms and improve their accuracy. In JetStream [16], the authors propose i) merging the storage with the processing layer, and imposing structure on the input data, and ii) applying *adaptive filtering* to minimize bandwidth requirements. The first design choice contradicts our design goals for a general scheduler that can be integrated in current data-analytics stacks. As to the second choice, filtering/sampling the input to reduce its volume leads to *approximate* results being produced. In environments where reduced accuracy is acceptable, adaptive filtering is complementary in that it can be applied *with* PIXIDA to further improve its performance. In addition, JetStream only supports operators with *one-to-one* dependencies, *e.g.,* `map` and `filter`, thus limiting its expressiveness, as it does not support operators like `reduce` and general `joins`. In Iridium [15], the authors mainly focus on minimizing latency, which, as they show, is often at odds with minimizing traffic. However, their system also includes a knob for trading off latency and cross-DC bandwidth, by using a heuristic to place tasks on nodes where data transfers are minimized. Contrary to PIXIDA, this heuristic accounts for each Stage of a job individually, rather than the job's graph as a whole. In addition, Iridium uses statistics about query frequencies and data accesses in the underlying storage system to move (parts of) datasets to locations where they are most likely to be consumed. This strategy, which crosses the storage and the processing layers of the data analytics stack, could be integrated in PIXIDA to achieve further bandwidth savings. Finally, Photon [4] is an application-specific stream processing engine, in contrast to PIXIDA whose functionality can be integrated with any existing general purpose framework.

In the area of graph algorithms, a graph problem that is related to the GENERALIZED MIN-CUT is the MIN-CUT problem on hypergraphs [10]. Although similar, its formulation has the same limitations as MIN-CUT in our settings. In the GENERALIZED MIN-CUT problem, the optimal partitioning of the children of a *special* node is not known in advance, thus the initial hyper-graph can be seen, once again, as a family of hyper-graphs with one member per possible partitioning of the children of each *special* node.

9. CONCLUSION

In this paper we presented PIXIDA, a scheduler for data parallel jobs that process geo-distributed data, *i.e.,* data stored on multiple DCs, and tries to minimize data movement across resource-constrained links. To achieve this, PIXIDA abstracts its goal as a new graph problem, which extends the classical MIN *k*-CUT problem from graph theory, and includes a solution to this new problem as part of its design. We integrated PIXIDA into Spark. Our experiments show that it leads to significant bandwidth savings across the targeted links, at a small price of an occasional extra latency, only in resource-constrained environments.

10. ACKNOWLEDGEMENTS

Computing resources for this work were provided by an Amazon Web Services (AWS) in Education Research Grant. The research of R. Rodrigues is funded by the European Research Council under ERC Starting Grant No. 307732. This work was partially supported by NOVA LINCS (UID/CEC/04516/2013) and the SwiftComp project (PTDC/EEI-SCR/1837/2012).

11. REFERENCES

[1] Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing/.

[2] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-parallel Computing. In

Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI'12, pages 281–294, San Jose, CA, USA, 2012. USENIX Association.

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, 2013.

[4] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 577–588, New York, NY, USA, 2013. ACM.

[5] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-clouds. In *Proceedings of the 6th ACM European Conference on Computer Systems*, EuroSys '11, pages 31–46, Salzburg, Austria, 2011. ACM.

[6] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou. Recurring Job Optimization in Scope. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 805–806, Scottsdale, Arizona, USA, 2012. ACM.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, 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 *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Hollywood, CA, 2012. USENIX Association.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3rd edition, 2009.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'04, pages 137–149, San Francisco, CA, USA, 2004. USENIX Association.

[10] A. Ene, J. Vondrák, and Y. Wu. Local Distribution and the Symmetry Gap: Approximability of Multiway Partitioning Problems. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 306–325. SIAM, 2013.

[11] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, Bern, Switzerland, 2012. ACM.

[12] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[13] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues. The Generalized Min-Cut Problem. NOVA LINCS Tech. Report. Available at: http://novasys.di.fct.unl.pt/~kkloudas/tr/techreport.pdf.

[14] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE'06, Atlanta, GA, USA, 2006. IEEE Computer Society.

[15] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low Latency Geo-distributed Data Analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 421–434, London, United Kingdom, 2015. ACM.

[16] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Seattle, WA, USA, 2014. USENIX Association.

[17] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[18] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. *CIDR 2015*, January 2015.

[19] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'15, pages 323–336, Oakland, CA, USA, 2015. USENIX Association.

[20] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 292–308, Pennsylvania, PA, USA, 2013. ACM.

[21] 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 *Proceedings of the 5th ACM European Conference on Computer Systems*, EuroSys '10, pages 265–278, Paris, France, 2010. ACM.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'12, pages 15–28, San Jose, CA, USA, 2012. USENIX.

[23] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 423–438, Pennsylvania, PA, USA, 2013. ACM.