Nap: Network-Aware Data Partitions for Efficient Distributed Processing

Or Raz and Chen Avin
School of Electrical and Computer Engineering
Ben-Gurion University of the Negev, Beer-Sheva, Israel
Email: razo@post.bgu.ac.il, avin@cse.bgu.ac.il

Stefan Schmid
Faculty of Computer Science
University of Vienna, Vienna, Austria
Email: stefan_schmid@univie.ac.at

***Abstract*—In order to support emerging data-intensive applications, many clever frameworks have been developed over the last years to efficiently and distributedly process big data sets, such as MapReduce. However, these frameworks are often optimized for relatively homogeneous environments, and accounting, e.g., for the varying connectivity of wide-area network infrastructure, may require complex placement algorithms. In this paper, we present *Nap*, which allows optimizing distributed data processing frameworks such as MapReduce for heterogeneous environments. *Nap* allows adapting resources dynamically, without requiring complex placement or migration algorithms, or modifications to the logic of the mappers and reducers. Rather, *Nap* simply changes the *data partition*, by spawning virtual nodes (e.g., reducers) depending on the demand. To this end, *Nap* leverages a connection to integer partition problems and employs Young lattices to guarantee minimal completion times (i.e., the makespan). In fact, *Nap* comes with provable performance guarantees and also supports applications that leverage redundancy to speed up executions further. In particular, to demonstrate our framework, as a case study, we show how to execute *multiway* joins across wide-area networks with limited bandwidth efficiently. Our experiments, based on a proof-of-concept prototype implementation, confirm the potential of *Nap* to reduce completion times.**

***Index Terms*—Distributed systems, networks, multiway joins, heterogeneity, Young lattices**

I. Introduction

With the advent of next-generation data-centric applications related, e.g., to health, business, science, social networking, or artificial intelligence, the amount of raw data that needs to be processed will continue to grow exponentially. While traditionally, data processing frameworks such as Hadoop and Spark were designed to run within a single datacenter, the need for more distributed and hence scalable approaches has become evident over the last years [1], [2], [3], [4]. Distributed approaches are not only motivated by the mere scale of data, but also by the fact that much of the to-be-processed raw data (e.g., related to user activity logs or Internet-of-Things) is also generated in a geographically distributed fashion.

However, the design of data processing frameworks for distributed environments is significantly more challenging than for a single datacenter. In particular, while datacenters usually provide a high bisection bandwidth and are fairly homogeneous, wide-area bandwidth is a scarce resource and connectivity between geographically distributed sites may vary significantly. This is problematic since cloud-based applications, including batch processing, streaming, and scale-out databases generate a significant amount of network traffic and a considerable fraction of their runtime is due to network activity [5]. Furthermore, the network is not the only resource which can lead to a higher degree of heterogeneity in geographically distributed applications. Ignoring the specific constraints of the environment in which the distributed application runs can lead to poor performance.

This paper studies the design of distributed data processing solutions which are *aware of* and *adapt to* the underlying resource infrastructure, such as network constraints, but also more generally. In particular, we are interested in algorithms which account for (and exploit!) heterogeneity.

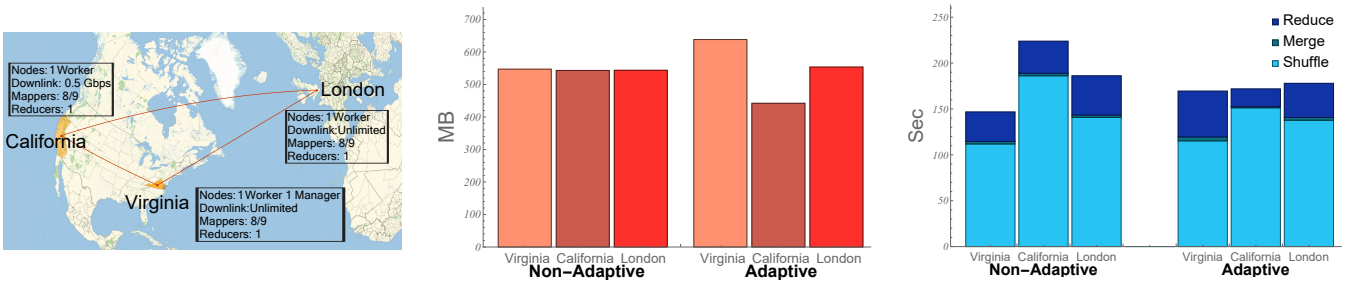
Contributions We present, analyze, and implement *Nap*[1], a network-aware and adaptive mechanism for fast distributed data processing, based on MapReduce. *Nap* does not require any complex placement or migration algorithms in order to account for and adapt to heterogeneity in the underlying infrastructure. Rather, *Nap* simply adjusts the data partition, assigning more data to faster nodes, using an efficient greedy algorithm motivated by *Young lattices*.

Nap comes with several attractive properties. First, it comes with *provable* performance guarantees, minimizing the *makespan*. Another interesting property of *Nap* is that it does not require any modifications to the logic of the mappers and reducers: it simply "fools" the application by spawning multiple logical nodes (in our case: virtual reducers), in order to make the application use the right resources seamlessly. Furthermore, *Nap* supports applications that exploit redundancy to improve performance further. In particular, we demonstrate how to use *Nap* to support *multiway* joins, which rely on redundancy and can often significantly improve query completion times compared to conventional cascades of binary joins (resp. chain joins, star joins, and even bushy joins) [6]: a multiway join requires just *one* phase, reducing communication overheads. In this regard, we generalize the model and approach by Afrati and Ullman [6], to account for differences in the available resources.

Our formal analysis provides insights into the usefulness of poorly connected nodes in heterogeneous infrastructures.

978-1-7281-2522-0/19/$31.00

[1]*Nap* stands for Network-Aware (data) Partitioning system.



(a) (b) (c)

Fig. 1. (a) The topology of the Hadoop's cluster on AWS. The cluster consists of four machines, one master in N. Virginia and three slaves which are spread evenly in each region (US East (N. Virginia), US West (N. California), and EU (London)). (b) The average amount of shuffled data to each computer/region, for both the adaptive (purple) and non-adaptive (orange) schemes. (c) The average completion times of the three reducers, including intermediate phases: shuffle (blue), merge (turquoise), and reduce (yellow) for both the adaptive and non-adaptive schemes.

For example, we show that it can sometimes be beneficial to exclude a weak node entirely from the computation, despite the "missed" resources.

We also report on a proof-of-concept implementation of *Nap*, based on minor modifications of Hadoop. In order to ensure reproducibility and to facilitate follow-up work, we share the code with the experimental results in [7].

Organization The remainder of this paper is organized as follows. In Section II, we revisit the state-of-the-art approach for performing a distributed data processing (e.g., Apache Hadoop), and demonstrate its limitations in the presence of heterogeneous environments empirically. Based on this motivation, we then introduce a formal model in Section III, and a lower bound in Section IV. Next, we present our solution in detail in Section V, and report on the first experimental results for our case study, multiway join, in Section VI. After reviewing related work in Section VII, we conclude in Section VIII. Due to space constraints, some of the proofs are left out but can be found in the Thesis of the first author [8].

II. Empirical Motivation

To motivate the potential benefits for network-aware and adaptive optimizations in distributed data processing, we report on a simple Hadoop experiment, considering a multiway join operation using Amazon Web Services (AWS). We deliberately omit some of the details now (more details will follow when we describe our prototype experiment): the purpose of this example is to provide intuition.

The input to our multiway join operation consists of three tables, based on ACM's digital library [9]. $X(v,p)$ - a Papers table, $Y(p,a)$ - a Papers-Authors table, and $Z(a,n)$ - an Authors table, with the following attributes: v - the Venue, p - the Paper ID, a - the Author ID, and n - the Author name. The multiway join is $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$ where $\bowtie$ denotes the join operator.

We employ Afrati and Ullman's approach [6] (which is non-adaptive) to execute the multiway join, and compare it to the adaptive approach of *Nap* which we will describe later in this paper. More specifically, we compare the completion time of the last reducer (C) when shuffling 1.6 GB of data. The experiment was conducted on AWS with one master in US East (N. Virginia) and three slaves, spread over three regions (US East (N. Virginia), US West (N. California), and EU (London)), where each one resides in a different region (multi-homing cluster). The downlink capacities of the reducers differ, and in particular, we study the impact of a lower processing rate at the reducer in California. See Fig. 1 (a) for details.

Fig. 1 (b) and (c) show the average results for ten runs of Hadoop jobs where each job had 25 mappers and 3 reducers that are allocated equally in the cluster. Fig. 1 (b) presents the amount of shuffled data sent to each reducer and (c) displays the completion time of each reducer indicating the time of each intermediate phase: shuffle, merge, and reduce. Both in the non-adaptive and in the adaptive scheme, the shuffle times dominate the completion times. In the *non-adaptive* case, the data were equally partitioned: each reducer received about 544 MB; in this setup, California is the bottleneck and delays completion (the last reducer finishes much later), see (b), Here, the merge and reduce phases required an equal amount of time for all the reducers, see, (c). However, there is a large difference in terms of the shuffle time between the reducers, due to the slow downlink in California: we incur more than a minute delay.

The *adaptive* scheme (based on *Nap*) is explicitly designed to overcome the gap between the different completion times of the reducers in the above experiment. Therefore, in the partition of the adaptive case, approximately an additional 100 MB was sent to Virginia instead of California due to its slow downlink rate. This change leads to a significant improvement in the job's completion time, reducing the makespan by about 20%. On the one hand, Virginia's reducing times become longer due to the additional 100 MB of transferred data. This results in almost identical completion times between the reducers, which (as we also show theoretically later) minimizes the makespan.

To implement the adaptive scheme that distributes the output of the map phase in a *non-uniform* way, we modified Hadoop and overrode the default *Partitioner class* (which divides the output uniformly). The modification enabled us to know the location of the mappers and reducers (their host computers), before we select how to re-partition the mappers' output.

III. Model and Problem Definition

We consider a MapReduce model [10] of computation. The input is first processed by a set of mapper processes $\boldsymbol{M}$, which distribute (during a shuffle phase) their outputs across a set of reducer processes $\boldsymbol{R}$. In a nutshell:

- Every mapper processes a share of the input, applies a map function, and stores the result locally. The function creates a tuple $\{key, value\}$, where the key is an identifier for a *partition function*, and the value is a smaller share of the input.
- Each map output, $\{key, value\}$, is passed to the local partition function which determines which tuple is assigned to which reducer, based on the tuple's key. By default, this function assigns a fair share to each of the reducers (uniform partition), if the job's input is uniform as well.
- In the *shuffle phase*, the data is transferred to the reducers through the network (at speed determined by the downlinks).
- Every reducer downloads its desired input (according to the partition), then merges and sorts the input locally. Afterward, the reduce phase function is applied.

Our objective is to optimize the job's completion time, in the presence of heterogeneous resources and bottlenecks. While our approach is more general, for now, we assume that the reducers are the bottleneck, and in particular, that their *(resource) rates*, like the downlink bandwidths or the processing rate, vary. We assume that there are $|\boldsymbol{R}| = r$ many reducers with reduce rates that are described by the vector $\bar{f} = (f_1, f_2 \dots f_r)$, where f_i [Bits/Sec] is the rate of reducer i. These rates are assumed to be positive integers[2], and w.l.o.g. the vector is sorted in decreasing order: reducer one has the maximum rate, and the minimum rate is one, i.e., $f_r = 1$. We denote by $W = \sum_{i=1}^{r} f_i$ the sum of the rates.

Let B denote the *total communication cost* [6], the amount of data resulting from the map phase. We assume that the number of mappers is large, so that the map phase is not a system bottleneck. Let B_i [Bits] be the amount of information that needs to be communicated to reducer i and $B = \sum_{i=1}^{r} B_i$. The completion time of a reducer i is denoted by $C_i = B_i/f_i$ [Sec]. The completion time of a job (i.e., it's makespan), C, is determined by the last reducer to complete the job, i.e., the *straggler*. Formally $C = \max_i(C_i)$.

We note that our model is general in that the total communication cost could be a function of the number of reducers: this will be relevant, for example, in our case study, considering multiway joins: To emphasize this, we will sometimes write $B(r)$. Usually, $B(r) = B_c \cdot D(r)$ where B_c is the original input to the problem and $D(r)$ is a non-decreasing function that indicates the amount of data duplication (redundancy) needed to support the execution of a job on r reducers. We naturally assume that the duplication function is sublinear, i.e., $D(r) < r$, but $D(r)$ could also be constant (e.g., as in the case of a word count task).

[2]This can be extended to rational.

Regarding our case study, we will consider a multiway join job, J, which involves multiple tables. Also, here, our goal is to minimize the (job) completion time C (other metrics of efficiency are possible [11], [6], [12]).

IV. Lower Bounds for Network-Aware Optimization

We first present a lower bound on the completion time of a MapReduce job in our model, when we use r reducers with reduce rates vector $\bar{f}$. This will help us identify a potential gain when optimizing completion time.

Theorem 1. *Consider a MapReduce job J on r reducers with reduce rates vector $\bar{f}$. Let $W = \sum_{i=1}^{r} f_i$ and $B(r)$ be the total communication cost. Then the completion time, C, is lower bounded as follows: $C \geq B(r)/W$.*

Proof. Assume by contradiction that $C < B(r)/W$, which means that for every reducer i it holds that $C_i < B(r)/W$. We can now bound its downloaded data, B_i:

$$\forall i \;\; C_i = \frac{B_i}{f_i} < \frac{B(r)}{W} \Longrightarrow B_i < \frac{B(r)}{W} \cdot f_i \tag{1}$$

But this leads to a contradiction.

$$B(r) = \sum_{i=1}^{r} B_i < \frac{B(r)}{W} \cdot \sum_{i=1}^{r} f_i = \frac{B(r)}{W} \cdot W = B(r) \tag{2}$$

□

It is important to note that the bound in Theorem 1 is not always feasible, because it is hard to ensure that all the reducers finish together. More importantly, the common use of MapReduce is to *uniformly* partition the $B(r)$ between the r reducers: in a manner which is *oblivious* (i.e., *non-adaptive*) to the reduce rates vector, $\bar{f}$ (but rather implicitly assuming that all the reduce rates are equal to one, i.e., $\forall i, \;\; f_i = 1$). We denote this scheme as *NA*, where each reducer downloads $B(r)/r$ data, i.e., $\forall i, \;\; B_i = B(r)/r$. In turn, the completion time of the NA scheme will be $B(r)/r$, since the rate of the slowest reducer is $f_r = 1$.

The NA scheme has several nice properties. First, when the reduce rates vector is uniform and $\forall i, \;\; f_i = 1$, NA achieves an optimal completion time. For this case $W = r$. Second, when the reduce rates vector is uniform, the more reducers we use, the better the completion time. This is one of the basic motivations for the MapReduce approach: the more resources we throw at a problem, the better is the performance.

But, when the reduce rates vector is *heterogeneous* and $W > r$, the NA scheme results in non-optimal completion time.

Corollary 1. *When executing a MapReduce job J on r reducers with scheme NA, and $\max_i\{f_i\} > 1$, this results in a higher completion time than the lower bound of Theorem 1 predicts.*

$$C_{NA} = \frac{B(r)}{r} > \frac{B(r)}{W} \tag{3}$$

In this paper, we present a scheme which optimizes the data partition toward a heterogeneous rate vector.

V. *Nap*

In this section, we present *Nap*, a scheme which is aware of (i.e., adaptive to) the reduce rates vector $\bar{f}$. The resulting optimizations can improve the performance over the uniform partition described above. The basic idea of the adaptive scheme, *AD*, is simple and easy to deploy: We realize an improved partition by fooling the system, and introducing a notion of *virtual reducers*. More concretely, instead of using only r reducers, we use v virtual reducers, and we assign them smartly among the r *physical* reducers to minimize the completion time of the MapReduce job.

A. Adaptive Algorithm and First Analysis

AD works as follows. First, we partition the reducers input (mappers output), $B(v)$, *uniformly* between v virtual reducers, using the *same* partition function of the NA scheme. This results in every virtual reducer receiving $B(v)/v$ input data. Second, we introduce an *assignment* function $\lambda = (v_1, v_2, \ldots, v_r)$, where v_i denotes the number of virtual reducers we assign to physical reducer i, and $v = \sum_i^r v_i$. Since a physical reducer i hosts v_i virtual reducers its input data is $B_i = (B(v)/v) \cdot v_i$. Clearly when the input data are divided into v smaller pieces (virtual reducers), we can tune the partition function by assigning the v virtual reducers among the r "physical" reduce processes.

Note, however, that in principle the number of virtual reducers could be even lower than r: namely if we do not need to use some of the reducers for minimizing the completion time. Let AD$[v]$ denote the use of the AD scheme with v virtual reducers, and assume it corresponds to an optimal assignment (which we will discuss later).

So far, we have seen that executing a MapReduce job using the NA scheme can be inefficient (Corollary 1). We now present a tighter upper bound of the completion time for the AD scheme, denoted as $C_{\text{AD}[v]}$ when we must use *all* the r reducers. We show that when $v = W$ and $\lambda = \bar{f}$, the AD scheme is optimized.

Theorem 2. *Consider a MapReduce job J performed using the adaptive scheme AD, and let $W = \sum_{i=1}^r f_i$. Then:*

1) *If $v = W$ virtual reducers and the assignment $\lambda = \bar{f}$, then the completion time will be $C_{AD[W]} = B(W)/W$: an identical completion time, $C_i = C_{AD[W]}$, for every reducer.*
2) *For any $v \neq W$ virtual reducers and any assignment λ using* **all** *the reducers, the completion time, is $C_{AD[v]} \geq C_{AD[W]}$.*

Proof. We first prove (1). Consider a $\lambda = \bar{f}$ partition on W virtual reducers where we have that $\forall i,\ v_i = f_i$. Using this partition, the completion time of all reducers is identical.

$$C_i = \frac{B(W) \cdot v_i}{v \cdot f_i} = \frac{B(W)}{v} \cdot \frac{v_i}{f_i} = \frac{B(W)}{v} = \frac{B(W)}{W} \quad (4)$$

We turn to prove (2). First note that since we assume that λ uses all the r reducers, namely each reducer has at least one virtual reducer, this means that $v \geq r$. So we have two cases to consider (i) $r \leq v < W$, and (ii) $v > W$. For case (i) recall that in our model reducer r has a reduce rate that equals one, $f_r = 1$, and it has at least one virtual reducer, $v_r \geq 1$; thus its completion time is at most the completion time of the operation, i.e., $C_{\text{AD}[v]} \geq C_r = (B(v) \cdot v_r)/(v \cdot f_r) = (B(v)/v) \cdot (v_r/f_r) \geq B(v)/v$. Recall that $B(v)/v$ is decreasing with v (since $D(r)$ is sublinear); so consequently, the completion time for $r \leq v < W$ is also decreasing $C_{\text{AD}[v]} \geq B(v)/v > B(W)/W = C_{\text{AD}[W]}$.

To prove case (ii), we use Theorem 1 which states that the optimal C for a total communication cost B is B/W. For $v > W$ we have $B(v) \geq B(W)$ so $C_{\text{AD}[v]} \geq B(v)/W \geq B(W)/W = C_{\text{AD}[W]}$ □

What we have shown until now is that the optimal partition of virtual reducers when we use *all* of the r reducers is to have $v = W$ virtual reducers and an assignment $\lambda = \bar{f}$. This improves the performance of the NA scheme. It follows from these results that if the reduce rates vector is homogeneous and $W = r$, then we would always benefit from using all the r reducers. But, what we show next is that this is not always the case. Consider a MapReduce job that requires replication (redundancy) of the data as a function of the reducers/virtual reducers, in this case $B(v) = B_c \cdot D(v)$ and $D(v)$ is non-decreasing, but also not a constant function. In the next section, we will see an example of this kind of job: a multiway join of multiple tables.

If additionally, the reduce rates vector is heterogeneous, then we might *not* need to use all of the r reducers. The next claim states this formally. For simplicity, we only prove it for an increasing $D(v)$.

Claim 1. *Consider a MapReduce job J performed using AD scheme and let $B(v) = B_c \cdot D(v)$ where $D(v)$ is monotonically increasing. Then, using all the r reducers might not result in the optimal completion time.*

Proof. We will prove this with a simple example. Consider a case where $r = 2$ and $\bar{f} = (1, x)$ so $W = x + 1$. By assumption $D(2) > D(1)$. When we use the two reducers, the optimal completion time we can achieve is $B(2)/W = (B_c \cdot D(2))/(x + 1)$. On the other hand, if we use a single virtual reducer only on reducer two (with rate x), then the completion time will be $B(1)/x = (B_c \cdot D(1))/x$. Solving for x we have that for x that satisfy the following equation, it is better to use only a single physical reducer. Such an x always exists (since ϵ is a constant larger than zero).

$$\frac{x}{x+1} > \frac{D(1)}{D(2)} = 1 - \epsilon \quad (5)$$

Hence, sometimes using less reducers is helpful. □

B. Using Less Physical Reducers

Given the above claim, we are left with one important question: What is the optimal number of virtual reducers, v, and what is the assignment to the r reducers, λ, that minimize C? We next present a solution to this problem using a *Young Lattice* [13].

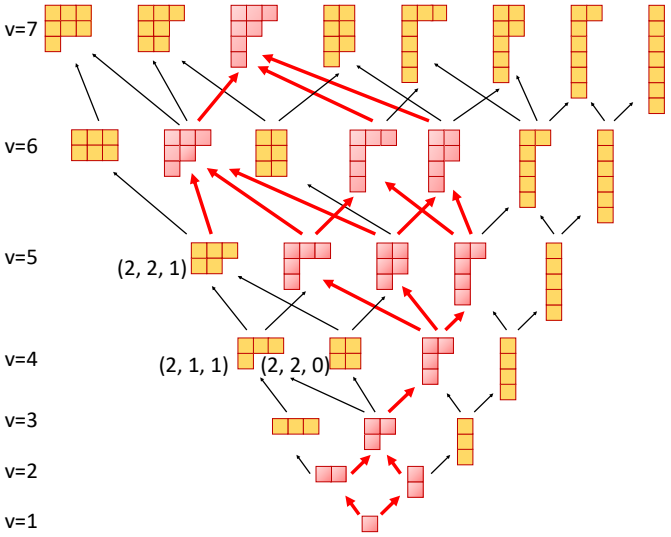


Fig. 2. A Young lattice for three parts, $r = 3$ reducers with reduce rate vector, $\bar{f} = \{4, 2, 1\}$, and seven levels, virtual reducers ($v = 7$). The graph has optimal walks which are highlighted, sequence of red Young diagrams, optimal assignments, follows by red edges.

We note that since the number of virtual reducers on each reducer is an integer, the assignment of v virtual reducers can be seen as an *integer partition*. Integer partition λ of $v \in \mathbb{Z}^+$ is a sequence of positive integers (called *parts*), whose sum is v [13], [14], [15]. These partitions are distinct, and the sequence of non-decreasing integers known as *Young diagram*. A Young diagram [13] is a finite collection of v boxes, or cells, arranged in left-justified rows, with the row lengths in nonincreasing order. It, therefore, presents an integer partition of v. The containment of one Young diagram in another defines a partial ordering on the set of all partitions. This creates a lattice structure, known as *Young's lattice* (see Fig. 2). This lattice, which is also known as a Young graph, is a graph composed of vertices that represent partitions (Young diagrams); a directed edge between two vertices exists when one partition (vertex) can be obtained from another partition (vertex) by adding *one* box to it.

We explain the lattice and its connection to our problem using Fig. 2 for three physical reducers with reduce rates vector, $\bar{f} = \{4, 2, 1\}$, and at most $W = 7$ virtual reducers. Each diagram (partition) has an order of the boxes corresponding to the order of the reducers (sorted by their reduce rates); the number of boxes on each part, namely, the number of virtual reducers on each reducer, is labeled as $(v_1, v_2, \ldots, v_r)$. For instance, the leftmost diagram on level four (integer partitions of $v = 4$) has two virtual reducers on R_1, one virtual reducer on R_2, and one virtual reducer on R_3, $(2, 1, 1)$; and to the right of it, on the same level, there is another partition of $v = 4$ virtual reducers, two on R_1, two on R_2 and none on R_2, $(2, 2, 0)$.

Each diagram (i.e., virtual reducers assignment), can be translated to a completion time. So in each level, which denotes the given number of virtual reducers, we color in red

Algorithm 1 `NapAssign` $(\bar{f})$

Input: Reducers' reduce rates vector, $\bar{f}$.
Output: Optimal assignment λ.

```
W ← sum [f̄], λ ← f̄, TempP ← ∅
for i ← 1 to (W − |f̄|) do
    TempP ← TempP ⊕ 1
    if C(TempP) < C(λ) then
        λ ← TempP
    end if
end for
return λ
```

the best assignment, i.e., the assignments with the minimal completion time. Note that several assignments can achieve the minimum. Accordingly, edges that are directed into such optimal assignments are also colored in red. These edges create directed paths, which we call each an *optimal walk* that connects optimal assignments, starting from one virtual reducer up to seven virtual reducers. We will use these walks as the basis for a greedy search, for an optimal assignment (see later, Algorithm 1).

An *optimal walk* can be created greedily starting from $v = 1$ virtual reducer placed optimally (on the fastest reducer) and by using the operator $\bigoplus 1$ which adds one virtual reducer to the current assignment optimally (breaking ties arbitrarily). The operator selects a reducer (out of all the possible reducers) that will result in minimal C for the new assignment, after the addition of one more virtual reducer. It was shown in [8] (and earlier in [16]) that given the optimality of the assignment for level v, the $\bigoplus 1$ operator ensures the optimality of assignment $v + 1$.

To find the optimal number of virtual reducers and the best assignment of them to the physical reducers we use an optimal walk on the Young lattice as the basis for our algorithm, Algorithm `NapAssign`, presented in Algorithm 1. The algorithm starts with an assignment $\lambda = \bar{f}$ and then go along an optimal walk and replace the current assignment with a better one if it finds such. We denote $C(\lambda)$ as the completion time for the assignment λ, note that the number of virtual reducers, v, is known from λ. Formally we can prove the correctness of the algorithm.

Theorem 3. *Algorithm* `NapAssign` *finds an assignment with a minimum completion time for running the AD scheme.*

Notice that the `NapAssign` algorithm goes along a walk only up to $W - r$ virtual reducers instead of W virtual reducers as mentioned earlier. This optimization in the running time can be proven formally.

Claim 2. *The completion time of an optimal assignment with $(W - r + \beta)$ virtual reducers is longer than an optimal assignment with W virtual reducers, for any positive integer $\beta \neq r$.*

We conclude the discussion on the algorithm by showing its

running time is $O(W \cdot \log r)$. Using prior work by Dessouky et al. which solves a similar problem: scheduling jobs on uniform parallel machines (homogeneous reduce rates vector) [16], it can be shown that the $\bigoplus 1$ operator can be implemented using a priority queue with $O(\log r)$ running time.

Claim 3. *The running time of Algorithm* `NapAssign` *is* $O(W \cdot \log r)$.

In the next section we present a case study on a multiway join preformed by MapReduce job [6].

VI. Case Study: Multiway Joins

We now demonstrate our framework for an interesting case study: multiway joins, i.e., MapReduce queries that join multiple tables in one phase in our model. The challenge here is how to efficiently join all the tables, which initially do not necessarily have to have a joint attribute. We first revisit the elegant scheme for performing multiway joins by Afrati and Ullman [6]. Our approach will build upon [6], which so far was non-adaptive to the reduce rates. We will compare the non-adaptive and the adaptive schemes both analytically and empirically, mainly for the multiway join mentioned in the motivation, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$ (see Section II).

A. NA and AD Schemes for Multiway Join

First, let us explain Afrati and Ullman's approach, denoted by NA: it is oblivious to the reduce rates vector (i.e., it assumes that all the reduce rates are the same). Afrati and Ullman's method partitions, and replicates, the rows of the tables between the reducers, in a way that every reducer could complete a unique local join (reduce-side join [17]). The union of these local (multiway) joins provides the answer to the query. Due to the replication (i.e., redundancy) in the approach of [6], the total communication cost becomes a function of the number of reducers, $B(r) = B_c \cdot D(r)$ where $D(r) = O(r^{1-\alpha})$ and α depends on the multiway join parameters. The scheme performs a multiway join using the set of s joint attributes in the tables, denoted as $A_1, A_2, \ldots A_s$. For each joint attribute A_i, we define a shared variable s_i which will determine the amount of replication (i.e., "share") each joint attributes will have. In [6]'s method, every row in a table is replicated to s_i reducers (keys) for each joint attribute A_i that is *not* in the table. The goal of [6]'s method is to optimize the shared variables, but this is independent of our approach, and we use [6]'s method as it is.

We will demonstrate both schemes by an example motivated from [6] and shown in Fig. 3. We use nine reducers ($r = 9$) and reduce rates vector $\bar{f} = (8, 8, 6, 4, 4, 2, 2, 1, 1)$. So, $W = 36$.

For the multiway join $X(v,p) \bowtie Y(p,a) \bowtie Z(a,\ n)$ there are two joint attributes: $A_1 = p$ and $A_2 = a$; therefore, there are also two shared variables, which can be assumed, for this example, to be $s_1 = s_2 = 3$. Recall that in the MapReduce workflow, the map output is a tuple with a key which helps identify the reducer (by the partition function). Here the key is a vector of size s (2, the number of joint attributes), created by s hash functions, one for each join attribute, $(h_1(p), h_2(a))$.

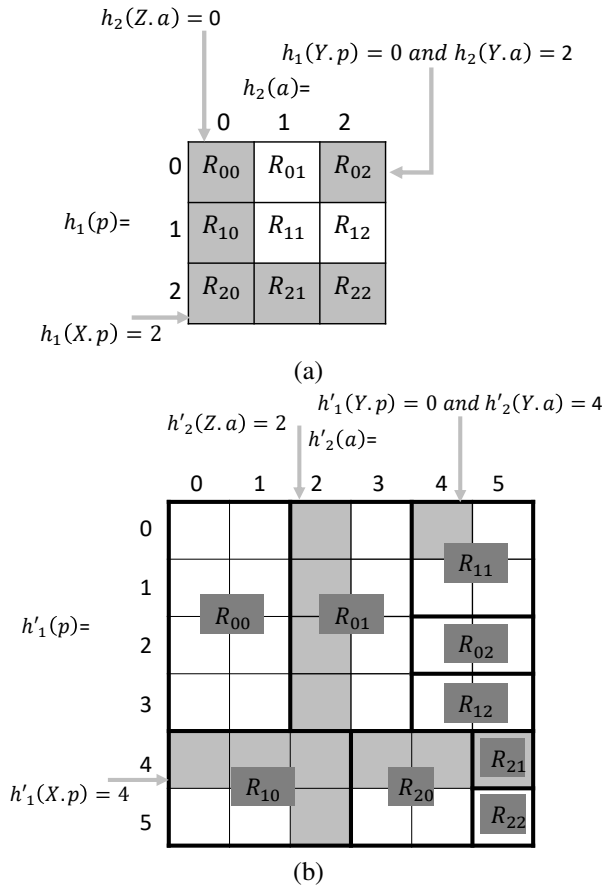


Fig. 3. Partitioning the rows of tables X, Y and Z for $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$ among $r = 9$ reducers. (a) shows the distribution of r keys to r reducers in NA while (b) shows the distribution of $v = W = 36$ keys to r reducers based on the reduce rates vector $\bar{f} = (8, 8, 6, 4, 4, 2, 2, 1, 1)$ in AD.

Each shared variable can be seen as the number of buckets that the attribute is hashed to. Consider Fig. 3 (a); in our example there are $s_1 \cdot s_2 = 9$ keys (or cells in the matrix) where each key represents a reducer, so there are $r = 9$ reducers. The set of keys is $(0,0), (0,1), \ldots (2,2)$, which can be expressed as a 2-dimensional matrix for this example, one dimension for each shared variable. Now consider a row in Table X and assume $h_1(X.p) = 2$; then this row will be mapped to three keys and be duplicated to three reducers, $(2,0), (2,1), (2,2)$, since Table X does not have attribute $A_2 = a$ and it must be assumed to have all the possible s_2 values (in this case $0, 1, 2$). For the same reason a row in table Z, which is missing attribute p and when assuming $h_2(Z.a) = 0$, is mapped to three reducers $(0,0), (1,0), (2,0)$, because of the degree of replication from the related "share" ($s_1 = 3$). If we consider Table Y, which has the two join attributes a and p, then every row in Y will always map to a single key $(h_1(Y.p), h_2(Y.a))$. This method guarantees that any rows that need to join based on a joint attribute will end up in a unique reducer that has all the corresponding rows from other tables.

The AD scheme for multiway join uses the same idea of replicating the tables' rows per keys as in NA scheme but with the mechanism of virtual reducers. It creates keys/cells

as the number of virtual reducers, and then it partitions the v keys in a non-uniform way between the reducers, instead of uniformly partitioning the r keys between the reducers. Consider Fig. 3 (b) , which presents the AD scheme. We see that it divides the v keys between nine reducers, but now each physical reducer can be identified by more than a single key. Consider for example, that the AD scheme uses $v = W = 36$ virtual reducers. Now we can provide R_1 with 8 virtual reducers while R_9 will receive only 1 virtual reducer since its reduce rate is much slower ($f_9 = 1$). Note that the basic join method of Afrati and Ullman also works here. Every two rows that need to join will end up at a unique virtual reducer, and in turn at a unique physical reducer.

Next, we analyze the completion time for both of the schemes. Afrati and Ullman have shown that for this example $D(r) = O(r^{\frac{1}{2}})$. Since Afrati and Ullman assumed $\forall i, f_i = 1$, $\forall ij, B_i = B_j = B(r)/r$ and the minimal C by NA is shown to be [6]

$$C_{\text{NA}} = \frac{B(r)}{r} = O(\frac{r^{\frac{1}{2}}}{r}) = O(r^{-\frac{1}{2}}) \tag{6}$$

In the AD scheme the total communication cost can be calculated in the same way, $B_{\text{AD}[v]} = O(v^{\frac{1}{2}})$, and thus, the completion time for AD$[W]$ is $C_{\text{AD}[W]} = O(W^{(-\frac{1}{2})})$ which is better since $W > v$. Furthermore, the above analysis can be extended to any multiway join as long as the duplication function has $0 < \alpha < 1$.

B. Prototype Implementation & Evaluation

After understanding the multiway join example and analyzing the performance (i.e., completion time) of both schemes, non-adaptive and adaptive, we now return to our proof-of-concept experiment from Section II. This experiment demonstrates the merits of the network-aware approach, comparing the *Nap* prototype to the state-of-the-art implementation. Effectively, the implementation of *Nap* is straight-forward, and boils down to extending Hadoop with a new Partitioner class, which allows us to take control of the reducers' input selection and send relatively more data to well-connected nodes, using virtual reducers.

The experimental setup is as follows. We use four elastic compute cloud machines (one master and three slaves) running Ubuntu 14.04 with a modified version of Hadoop 2.9.1 on AWS. The master is a t2.xlarge instance located in Virginia while the slaves are M4.xlarge instances spread over three regions (US East (N. Virginia), US West (N. California), and EU (London)). The master has 16 GB of RAM and 4 VCPU cores, 50 GB of SSD, and moderate link bandwidth, while each worker has 16 GB of RAM and 4 VCPU cores, 100 GB of SSD, and a 10 Gbps link. We use Wonder Shaper [18], a command-line utility for limiting an adapter's bandwidth, to fix a downlink rate to 0.5 Gbps. Hadoop is using HDFS and YARN daemons (processes) to store the data and monitor its jobs [19], [20], [17]. The master instance is in charge of the whole computation by running the NameNode (NN), and the Resource Manager (RM) daemons, and the slaves

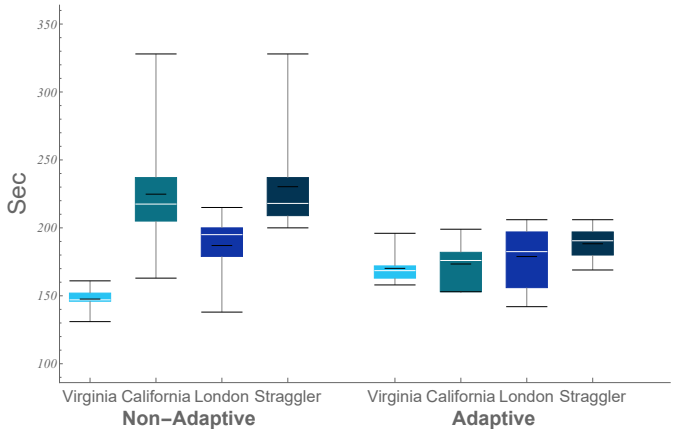


Fig. 4. A box plot of the completion time of all the ten jobs per reducer/location and the straggler (four in total) per partition, non-adaptive and adaptive.

are responsible for storing the data and running the workload within containers. Every worker runs two daemons, Datanode (DN), and Node Manager (NM). Upon Job execution, the RM decides where to allocate each YARN Child (Application Master (AM), map or reduce process) inside the slaves (see Fig. 1).

Each job has 25 mappers and 3 reducers, which are allocated equally across the cluster. Moreover, we have changed the starting point of the reduce containers and the shuffle phase to the same time of allocating map containers (field *mapreduce.job.reduce.slowstart.completedmaps*). This enables our code (Partitioner class) to distribute the data according to the downlink. Furthermore, the containers' memory specifications are 512 MB for the AM or map container, and 1024 MB for the reduce container. Only the slaves are contributing to the scheduling process because they have an NM daemon. Thus 48 GB RAM and 12 VCPU cores are left for scheduling the containers and it is vital for our implementation that all the containers be allocated in parallel. HDFS has a default block size of 128 MB and a replication factor of three; thus, all the data, also the input, resides in every worker. In addition, we have managed to split the input (almost) evenly around the 25 mappers.

We present results for ten runs of Hadoop jobs with two different partitions, non-adaptive (uniformly) and adaptive (non-uniform, $\lambda = (7,\ 6,\ 6)$), with 1.6 GB shuffled data.

Consider Fig. 4, which displays a box plot of the completion time per reducer, in every location, and their average per partition. Each box emphasizes the distribution of the ten results with a mean confidence interval. On each box, the black marker is the mean, the white is the median, and the low and upper fences are the min an max values, respectively. The first four box plots are for the non-adaptive partition, and the rest is for the adaptive partition. Concerning the non-adaptive partition, Virginia's reducer is by average the fastest while London and California struggle, depending on the job. There was no straggler, as we can easily see from the fences of the

straggler box. California struggles the most with a completion time that can be up to 250% in comparison to Virginia's fast reducer and with a high variance in comparison to the rest. This confirms our observation that the adaptation of the data partitioning according to the network constraints is beneficial. In the adaptive partitioning, the variance difference between each region is minor, unlike before, and still Virginia's reducer is by average the fastest. However, now on average, we have a new straggler, London's reducer, which slightly delays the job before completion (six seconds). There is also an improvement in the completion time per partition by 20%, and now the box plot is more skewed, with a lower variance.

We emphasize that our prototype implementation and experimental results should be understood as proofs-of-concept. Our main contribution lies on the conceptual and theoretical side. In particular, the prototype still has many limitations, as the following examples demonstrate. First, our prototype should be more carefully integrated with the speculative execution mechanism, to overcome the risk that in a cluster with slow containers many speculative containers are launched, overloading the server. This requires fine tuning of the thresholds for starting a speculative execution. Second, our current implementation requires clusters with enough RAM for allocating all the containers in parallel, and it can make sense to study more memory-efficient solutions. Clearly, these aspects as well as others (e.g., the specific type of reduce function) will influence the performance of our prototype and need to be investigated.

VII. Related Work

The impact of the network on cloud application performance is well-explored in the literature already. Mogul and Popa [5] argued that even inside a datacenter, cloud application performance intrinsically depends on high-performance networks: many cloud-based applications (batch processing, streaming, scale-out databases, etc.) generate much network traffic and a considerable fraction of their runtime is due to network activity. In order to ensure a predictable performance, many systems provide relative [21], [22], [23] or even absolute [24], [25], [26], [27] bandwidth guarantees, through reservation and network virtualization. The situation becomes more critical in wide-area networks, where bandwidth is usually much more scarce and connectivity more heterogeneous. Indeed, much existing work on wide-area analytics [28], [3] implicitly or explicitly deals with constraints introduced by the network, e.g., by using clever placement strategies [29], [30].

Performance optimization in MapReduce has been an active field of research for many years already, and job completion time is often the main concern [31], [32], [33], [34]. Much work also focuses on the shuffle phase, and in particular, I/O overheads [35], or issues related to data skew [36] and load-imbalances [37], but networking issues are also considered more generally [38], [39], [40], [41]. Many solutions do not aim to identify the specific reason for a bottleneck but generically deal with stragglers , e.g., using speculative executions and adaptive task placement [41], [39]. However, there are also solutions which explicitly and jointly optimize compute and network resources [40], [30].

In contrast to these approaches, the solution presented in our paper does not require to replace or speculate mappers and reducers, but rather, we aim to make most effective use of them in their current locations, by adapting the data partition.

As a case study, we considered join operations in this paper, and especially multijoins, which highlight the generality of our approach. The planning and optimization of queries, and in particular joins and its variants [42], [12], is an evergreen topic in the literature, often centering around completion time [43], and we are also not the first to consider joins in the context of MapReduce. However, most of these solutions do not account for the underlying infrastructure on which the query is executed [44]. The papers closest to ours are by Giroire et al. [38], by Afrati and Ullman [6], and by Slagter et al. [45]. Giroire et al. aim to minimize the completion time by proposing a better scheduling algorithm for the tasks accounting for the network. However, their approach is less general than ours and, e.g., does not support multijoins (nor improving completion time based on redundancy). Afrati and Ullman present a model for computing *multijoins* in MapReduce, accounting for communication costs by changing the data partition. However, their approach is non-adaptive as it assumes that all link capacities are equal. In this paper, we remove this restriction and generalize their model and result. Slagter et al. also present a network-aware multiway join algorithm for MapReduce, called SmartJoin. SmartJoin dynamically redistributes tuples directly between reducers in order to further improve performance, while we optimize the data partitioning, also exploiting redundancy.

VIII. Conclusion

This paper presented *Nap*, a simple approach to improve distributed data processing performance in heterogeneous environments (e.g., due to varying networking connectivity in wide-area networks), *without* the need to reposition mappers and reducers. Rather than performing complex placement optimizations, *Nap* just adapts the data partition (leveraging virtual reducers), requiring only minimal changes to the application. *Nap* provably achieves optimal completion times in our model, and supports a broad range of use cases, speed up execution times even further. We have also shown that *Nap* is able to determine the optimal number of reducers, and sometimes even decides not to use a physical resource at all, as it may harm performance. In order to achieve its guarantees, *Nap* leverages integer partition techniques to ensure uniform completion times (and hence minimize the makespan) with a greedy algorithm. Our of proof-of-concept implementation in [7] shows promising first results.

We believe that our work opens several interesting avenues for future research. In particular, while our prototype serves as a proof-of-concept, many additional practical optimizations need to be performed to optimize its performance in practice, which involves algorithm engineering and more extensive measurement studies. More generally, it will be interesting

to explore additional case studies for network-aware optimizations, e.g., for other join operators beyond multijoins. It will also be interesting to consider scenarios with more complex bottlenecks, involving different resources whose capacity changes over time in an online manner. as well as scenarios where multiple jobs can arrive over time.

ACKNOWLEDGMENT

We would like to thank Niklas Semmler for his help and fruitful discussions.

REFERENCES

[1] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "Wanalytics: Analytics for a geo-distributed data-intensive world." in *CIDR*, 2015.

[2] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing data parallel jobs in bandwidth-skewed environments," in *Proc. VLDB*, 2015.

[3] 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 *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 275–288.

[4] 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 *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 323–336.

[5] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 44–48, 2012.

[6] F. N. Afrati and J. D. Ullman, "Optimizing multiway joins in a map-reduce environment," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1282–1298, 2011.

[7] O. Raz, "Nap — Github, repoistory," 2019. [Online]. Available: https://github.com/razo7/Nap

[8] ——, "Nadj: Network-aware and adaptive multiway joins," Master's thesis, Ben Gurion University of the Negev, 2019.

[9] ACM, "The association for computing machinery digital library — ACM, international learned society for computing," 2018. [Online]. Available: https://dl.acm.org/

[10] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] S. D. Viglas, J. F. Naughton, and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 285–296.

[12] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis, "Cliquesquare: Flat plans for massively parallel rdf queries," Ph.D. dissertation, INRIA Saclay; INRIA, 2014.

[13] G. Kreweras, "Sur une classe de problemes de denombrement lies au treillis des partitions des entiers." Ph.D. dissertation, Paris., 1965.

[14] R. P. Stanley, "Differential posets," *Journal of the American Mathematical Society*, vol. 1, no. 4, pp. 919–961, 1988.

[15] ——, "Algebraic combinatorics," *Springer*, vol. 20, p. 22, 2013.

[16] M. I. Dessouky, B. J. Lageweg, J. K. Lenstra, and S. L. van de Velde, "Scheduling identical jobs on uniform parallel machines," *Statistica Neerlandica*, vol. 44, no. 3, pp. 115–123, 1990.

[17] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2015.

[18] magnific0, "Wonder shaper — Github, the world's leading software development platform," 2019. [Online]. Available: https://github.com/magnific0/wondershaper

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[20] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[21] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: sharing the network in cloud computing," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 187–198, 2012.

[22] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing," in *ACM SIGCOMM Computer Communication Review*, vol. 43-4, 2013, pp. 351–362.

[23] A. Shieh, S. Kandula, A. G. Greenberg, and C. Kim, "Seawall: Performance isolation for cloud datacenter networks." in *HotCloud*, 2010.

[24] C. Fuerst, S. Schmid, L. Suresh, and P. Costa, "Kraken: Online and elastic resource reservations for multi-tenant datacenters," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[25] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 199–210, 2012.

[26] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM computer communication review*, vol. 41-4. ACM, 2011, pp. 242–253.

[27] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proc. ACM CoNEXT*. ACM, 2010, p. 15.

[28] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: optimizing data parallel jobs in wide-area data analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.

[29] T. Renner, L. Thamsen, and O. Kao, "Network-aware resource management for scalable data analytics frameworks," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 2793–2800.

[30] Q. Pu, G. Ananthanarayanan, P. Bodík, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *Computer Communication Review*, vol. 45, pp. 421–434, 2015.

[31] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[32] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 472–483, 2010.

[33] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[34] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Proceedings of the 15th International Conference on Extending Database Technology*. ACM, 2012, pp. 38–49.

[35] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 43.

[36] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 25–36.

[37] J. Myung, J. Shim, J. Yeon, and S.-g. Lee, "Handling data skew in join algorithms using mapreduce," *Expert Systems with Applications*, vol. 51, pp. 286–299, 2016.

[38] F. Giroire, N. Huin, A. Tomassilli, and S. Pérennes, "When network matters: Data center scheduling with network tasks," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2019.

[39] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *Osdi*, vol. 10, 2010, p. 24.

[40] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang, "Wide-area analytics with multiple resources," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 12.

[41] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *Osdi*, vol. 8, 2008, p. 7.

[42] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2rdf: Rdf querying with sparql on spark," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 804–815, 2016.

[43] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1998, pp. 34–43.

[44] Y. E. Ioannidis, "Query optimization," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 121–123, 1996.

[45] K. Slagter, C.-H. Hsu, Y.-C. Chung, and G. Yi, "Smartjoin: a network-aware multiway join for mapreduce," *Cluster computing*, vol. 17, no. 3, pp. 629–641, 2014.