Formalization and Taxonomy of Compute-Aggregate Problems for Cloud Computing Applications

Pavel Chuprikov[a], Alex Davydow[b], Kirill Kogan[c], Sergey I. Nikolenko[d,*], Alexander Sirotkin[e]

[a]*IMDEA Networks Institute, Av. Mar Mediterraneo, 22, 28918 Leganes, Madrid, Spain*
[b]*Harbour Space University, Carrer de Rosa Sensat, 9, 08005 Barcelona, Spain*
[c]*Harbour Space University, Carrer de Rosa Sensat, 9, 08005 Barcelona, Spain*
[d]*IMDEA Networks Institute, Av. Mar Mediterraneo, 22, 28918 Leganes, Madrid, Spain*
[e]*National Research University Higher School of Economics, St. Petersburg, Russia*

Abstract

Efficient representation of data aggregations is a fundamental problem in modern big data applications, where network topologies and deployed routing and transport mechanisms play a fundamental role in optimizing desired objectives such as cost, latency, and others. In traditional networking, applications use TCP and UDP transports as a primary interface for implemented applications that hide the underlying network topology from end systems. On the flip side, to exploit network infrastructure in a better way, applications restore characteristics of the underlying network. In this work, we demonstrate that both specified extreme cases can be inefficient to optimize given objectives. We study the design principles of routing and transport infrastructure and identify extra information that can be used to improve implementations of compute-aggregate tasks. We build a taxonomy of compute-aggregate services unifying aggregation design principles, propose algorithms for each class, analyze them theoretically, and support our results with an extensive experimental study.

Keywords: compute-aggregate problems, cloud computing

1. Introduction

Data centers store data at different interconnected locations. Modern big data applications are highly distributed, and requests need to satisfy different, often conflicting objectives: latency, cost efficiency, and others [1, 2, 3]. *Compute-aggregate* problems, where several data chunks must be aggregated in a network sink, encompass an important class of big data applications implemented in modern data centers. Traditionally, applications have little control over how network transport handles the data. Latency optimization should account for properties of underlying transports in order to avoid, e.g., the *incast problem* [4, 5], and optimizing latency for several compute-aggregate tasks can overload "fastest" (and more expensive) links. We believe that more fine-grained control is required to implement desired objectives transparently for applications.

In this work, we assume that each compute-aggregate task should conform to a budget constraint since different cloud tenants are able to invest different economic resources to compute their aggregations. To avoid oversubscription of "fastest" links, they can also have different costs of sending data over a link. The problem now divides into two completely decoupled phases:

*Corresponding author
Email addresses: pschuprikov@gmail.com (Pavel Chuprikov), adavydow@gmail.com (Alex Davydow), kirill.kogan@gmail.com (Kirill Kogan), snikolenko@gmail.com (Sergey I. Nikolenko), alexander.sirotkin@gmail.com (Alexander Sirotkin)

(1) find a "cheapest" plan given a distribution of data over the network, an aggregation function (that computes the size of aggregating two different pieces of data), and the cost of sending a unit of data over a link, and

(2) actually redistribute aggregations computed on the first phase while optimizing desired objectives.

In this setting, we can solve the first phase in a serial way, independently of the properties of underlying transport protocols, while the second phase can address such problems as incast. This is a natural generalization of traditional transports designed to implement efficient aggregations.

The first phase is of separate interest since it can represent various economic settings (e.g., energy efficiency) during aggregation; this phase can also lead to better utilization of network infrastructure since the cost to send a unit of data through the links can differ for different compute-aggregate instances. Hence, our primary goal is to identify universal properties of compute-aggregate tasks that allow for unified design principles of "perfect" aggregations on the first phase. Incorporating properties of aggregation functions into final decisions requires new insights on the model level and may lead to more efficient aggregation. There is definitely room for it: the average final output size of a job has been estimated at 40.3% of the initial data sizes in Google [6], 8.2% in Yahoo, and 5.4% in Facebook [7].

Most existing solutions dealing with incast problems attack them by varying the settings of flow and congestion control [8]; they are reactive in nature. Instead of directly optimizing desired objectives such as flow completion time or throughput, for compute-aggregate jobs we propose a preliminary step of computing an aggregation plan that would allow for a better exploitation of network infrastructure and cost reduction of compute-aggregate jobs. Note that the proposed step does not replace various optimizations of congestion control in underlying transports but rather complements them. In difference from specific proposed network topologies optimizing behaviours of compute-aggregate jobs such as [9], we introduce a general framework requiring only aggregation functions from applications to guarantee the efficiency of data exchange.

In this work, we define a model for constructing an aggregation plan under budget constraints that requires applications to specify only one aggregation property: the (approximate) size of two data chunks after aggregation. Properties of aggregation functions can have a significant effect on the aggregation plan. We classify compute-aggregate tasks into several categories with respect to this property, propose algorithms for these optimization problems, and analyze their properties, proving a number of results on their performance and complexity, both positive (polynomial algorithms with good approximation ratios) and negative (inapproximability results).

The paper is organized as follows. In Section 2 we summarize prior art. Section 3 introduces the model, motivating our main abstraction, the aggregation size function, and introducing the basic CAMproblem. In Section 4 we classify aggregation size functions with regard to their effect on input data chunks and study their computational properties, i.e., hardness and approximability of optimization problems. We introduce two special cases of the CAMproblem, TCAMfor trees and CCAMfor complete graphs, present a number of algorithms for different cases of aggregation size functions, and prove bounds on their approximation ratios. Section 5 presents and discusses our experimental results based on realistic network topologies, and Section 6 concludes the paper.

2. Related work

Various frameworks split computations into multiple phases: Map-Reduce-Merge [10, 11] extends MapReduce to implement aggregations, Camdoop [9] assumes that an aggregation's output size is a specific fraction of input sizes, Astrolabe [12] collects large-scale system state and provides on-the-fly attribute aggregation, and so on.

For example, STAR [13] extended the line of research started by SDIMS [14], which uses distributed hash tables (DHTs) to create information management systems. STAR adaptively sets precision constraints for processing aggregation. The system considers mechanisms to optimize aggregation but does not explicitly optimize its overlay network fan-in or structure. In fact, all of these systems use measurements and metrics to approach a practical optimal setup and are limited in their adjustments by the frameworks used to create

them. This allows them to adapt and react to the network conditions, but does not approach the level of rigor of creating an application-specific overlay from ground up based on mathematically optimized principles. The overlays are once again restricted by the underlying framework, but the system itself efficiently aggregates results to respond to queries.

Similar to other data-flow systems [6, 15, 16], Naiad [17] offers the low latency of stream processors together with the ability to perform iterative and incremental computations. The work [16] introduces a distributed memory abstraction for fault-tolerant in-memory computation on large clusters, with orders of magnitude better latency than disk accesses. According to [16], this implementation can have orders of magnitude better latency than disk accesses, which addresses the problem raised by Venkataraman et al. [18] that data access is a significant bottleneck for iterative calculations in various distributed frameworks.

Other stream processing frameworks support low-latency dataflow computations over a static dataflow graph [19, 20, 21], while [22] explores optimal tree overlays to optimize latency of compute-aggregate tasks under specified budget constraints. More recent efforts in this direction have concentrated on reducing the latency for time-critical applications in the clouds and generally attempts to streamline project development in the cloud and make it suitable for time-critical applications. These efforts include SWITCH [23, 24, 25], a framework that provides an abstraction layer for the development of low-latency native cloud applications, CloudWave [26] and NDP [27] that provide tools for runtime monitoring in the cloud, and Mobi-IoST [28] that also supports real-time applications but concentrates on IoT and edge devices. As for earlier work, we refer to [29] for a general overview of cloud computing environments prior to 2014.

3. Motivation and model description

Our main objective is to use a network in the best possible way for a given compute-aggregate task. This is a problem with many variables. In this work, we leave most of them to the network transport layer (e.g., it chooses how transmissions should be spread in time), concentrating on the *aggregation plan* that defines the order of aggregation. The aggregation plan is fully decoupled from transport implementation and will be formalized below.

3.1. Compute-aggregate tasks

We model a network as an undirected connected graph $G = (V, E)$, where V is a set of computing nodes connected by links (edges) E. Note that since we operate on an application level, we are free to use any overlay topology in place of G that captures only information relevant to a specific compute-aggregate task. The task is represented as a set of initial data chunks $C = \{\boxed{x_0}, \boxed{x_1}, \ldots, \boxed{x_k}\}$, with each chunk $\boxed{x_i}$ characterized by its location $\mathrm{v}(\boxed{x_i})$ and size $\mathrm{size}(\boxed{x_i})$. Since many compute-aggregate tasks require the result to be fully available on a specific node (e.g., to allow low-latency responses), we assume a special root vertex $t \in V$ where all data chunks should be finally aggregated.

The hardest part to define is "the best possible way": objectives are application-specific and may include latency, throughput, or a more subtle objective such as congestion avoidance. We model them with a single per-link parameter, the *cost*, a flexible way to both freely combine objectives and keep the optimization problem clear. Formally, the cost function $c : E \to \mathbb{R}_+$ on the topology graph G maps each link e to its transmission cost per data unit $c(e)$; to transmit $\boxed{x}$ through e one must pay $c(e) \cdot \mathrm{size}(\boxed{x})$. If load balancing is needed, it can be achieved by using low values of c for different subsets of links for different tasks.

A simple example of a compute-aggregate task is shown on Fig. 1a. Costs are shown on the edges, square brackets denote chunks, and the root vertex is marked by t.

3.2. Move to root

To define an aggregation plan in a clean but practical way it is important to understand the impact it may have on the rest of the system (e.g., on a transport layer or computing infrastructure). We begin with the simplest form of an aggregation plan that we call "move to root": bring everything to the root node t (we say that an aggregation plan *moves* or *aggregates* for simplicity; in practice data transmission and aggregation are handled by the transport and application layers respectively). "Move to root" can be suboptimal with

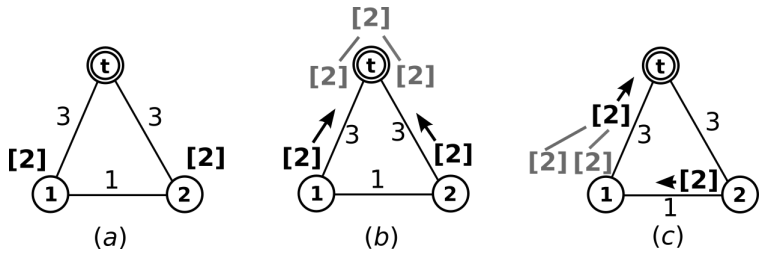


Figure 1: A sample compute-aggregate task with three vertices t (target), u, and v: (a) the problem; (b) "move to root" plan with cost 12; (c) optimal aggregation plan with cost 8. Transmission cost of every edge is specified near the middle of the edge (e.g., $c(u, v) = 1$).

regard to transmission costs. Suppose that in the example on Fig. 1 the aggregation function chooses the best chunk, so the aggregated size does not exceed the maximal size of initial chunks. Now "move to root" has total cost 12 (Fig. 1b: two chunks of size 2 each moving along edges of cost 3), while on Fig. 1c one chunk moves to vertex 1 paying 2, then chunks merge, and chunk of size 2 moves to t with total cost 8.

But concerns arise even apart from transmission costs. A naive implementation of the "move to root" plan that moves all data chunks to the root and then aggregates leads to the transport layer directing a lot of traffic towards t, possibly overflowing ingress buffers there and increasing latency due to the notorious TCP-incast problem. Moreover, in a low-latency application that aggregates in RAM [30, 31] storage capacity can be exhausted when all data chunks are stored at t.

This problem can be alleviated with intermediate aggregations. Data chunks can be sent to t sequentially in some order, and in the process some arriving chunks are immediately aggregated. Recent studies [6, 7] show that the final result of a compute-aggregate task is often only a small fraction (usually less than half) of the total size of initial data chunks; e.g., in counting problems the aggregation result is just a few numbers. Thus, keeping in memory a single intermediate chunk instead of several initial data chunks can significantly reduce storage requirements.

In general, not every order can be used for intermediate aggregations because the final aggregation result might depend on this order (e.g., string field concatenation), and it is undesirable for an aggregation plan to affect the result [32]. Fortunately, most aggregation functions do not depend on the aggregation order, that is, they are *associative*: $\mathbf{aggr}(\boxed{x}, \mathbf{aggr}(\boxed{y}, \boxed{z})) = \mathbf{aggr}(\mathbf{aggr}(\boxed{x}, \boxed{y}), \boxed{z})$, and *commutative*: $\mathbf{aggr}(\boxed{x}, \boxed{y}) = \mathbf{aggr}(\boxed{y}, \boxed{x})$. Below we assume that aggregations are both associative and commutative; such systems as *MapReduce* already assume this for most *reduce* functions and allow aggregations of intermediate data chunks with combiner functions [6]. The TCP-incast problem, on the other hand, can be mitigated by carefully spreading chunk transmissions in time (to reduce overlap), which requires complex synchronization on the part of the transport layer. Low-latency in-RAM applications also have to synchronize data transmissions to avoid too many data chunks "in the air" at the same time that cannot be aggregated; this is hard to implement in a distributed system, and if **aggr** is not commutative and associative this leads to more constraints since transmissions must occur in a specific order.

All of the above suggests that it is hard for the "move to root" heuristic to reconcile network transport limitations with storage constraints and the distributed environment. Hence, in this work we explore aggregations at intermediate nodes.

3.3. Exploiting intermediate nodes

The basic principle of *data locality optimization*, which lies at the heart of the *Hadoop* framework [33], is to *move computation to data* and as a result save on data transmission. We extend this strategy and try to *move aggregation to data* by allowing an aggregation plan to exploit intermediate nodes. Formally, an aggregation plan is a sequence P of operations $(o_0, o_1, \ldots, o_m)$, where each o_i is one of the following:

- either $\mathbf{move}(\boxed{x}, v)$, which moves a chunk x to a vertex v, or

- **aggr**($\boxed{x}$, $\boxed{y}$), which merges chunks $\boxed{x}$ and $\boxed{y}$ located at the same vertex; the result is a new chunk $\boxed{xy}$ at that vertex.

After all operations have been applied, the result must be a single data chunk $\boxed{z}$ at the root: $\mathrm{v}(\boxed{z}) = t$. For example, Figs. 1b and 1c show aggregation plans for the problem shown in Fig. 1a. Aggregation plans are fully decoupled from the transport layer, producing instructions and constraints that the transport layer must satisfy.

With this definition of an aggregation plan, it is easy to assign it with a transmission cost and pose the minimization problem. First, every operation o_i has an associated cost that we denote $\mathrm{cost}(o_i)$, which is defined as follows:

- $\mathrm{cost}(\mathbf{aggr}(\boxed{x}, \boxed{y})) = 0$ (there is no data transmission), and
- $\mathrm{cost}(\mathbf{move}(\boxed{x}, v)) = \mathrm{size}(\boxed{x}) \cdot d(\mathrm{v}(\boxed{x}), v)$, where $d(u, v)$ is the total cost of the cheapest path from u to v (if there are several such paths, any one can be used).

The total cost of an aggregation plan P, $\mathrm{cost}(P)$, is the sum of costs of all operations in P.

This approach of "moving aggregation to data" has some important advantages over "move to root". First, the TCP-incast problem becomes less pronounced because inbound traffic is spread among different nodes, and fewer nodes need to be synchronized. Moreover, the total number of transmitted bits is reduced due to earlier aggregations (we usually expect an aggregation result to be smaller than the total input size). Second, storage capacity is now less of a constraint since less data has to be collected per node. Last but not least, data transmission cost is also reduced (cf. examples on Fig. 1). Note, however, that in practice not all nodes may be used for data aggregation. For example, we may be restricted to nodes where initial data chunks reside because it is expensive to allocate additional compute nodes; or it can be a security concern to perform computation on intermediate nodes (e.g., initial nodes belong to a private cloud, and the rest are transit nodes). This question must be carefully answered, and in what follows we assume that the overlay graph G reflects this answer and maps **aggr** operations to appropriate nodes.

3.4. Aggregation size function

In order to formally define the optimization problem for aggregation, we have to know the following: given $\boxed{x}$ and $\boxed{y}$, what is the size of their aggregation result $\boxed{xy}$? This directly affects the cost of an aggregation plan, and different aggregation result sizes can lead to very different solutions. For example, if on Fig. 1 we assumed that the task is, e.g., sorting, where the size of an aggregated chunk is the sum of input sizes, the cost of the first plan would still equal 12, but the plan on Fig. 1c would now cost 14 and become suboptimal.

Unfortunately, the size of an aggregation result is application-specific, and in most cases the exact value depends on the actual content of $\boxed{x}$ and $\boxed{y}$; moreover, to determine this value we may need to actually perform aggregation (e.g., the number of key-value pairs in the counting problem cannot be predicted exactly unless we actually count). This is clearly infeasible since an aggregation plan must be constructed (and its cost evaluated) before the application performs any aggregations and the transport layer transmits any data. Therefore, we require each application to supply the *aggregation size function* $\mu : \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{R}_+$ that would estimate this size using only sizes of the inputs, so that for the purposes of optimization $\mathrm{size}(\boxed{xy}) = \mu(\mathrm{size}(\boxed{x}), \mathrm{size}(\boxed{y}))$. We do not expect these functions to be exactly correct, but they should provide the correct order of magnitude in order for the optimal solution to be actually good in practice. Since **aggr** is assumed to be associative and commutative, every aggregation size function μ should also have these properties. Some examples of μ for practical problems include:

- $\mu(a, b) = \mathrm{const}$ for finding the top k elements in data with respect to some criterion;
- $\mu(a, b) = \min(a, b)$ or $\mu(a, b) = \max(a, b)$ for choosing the best data chunk;
- $\mu(a, b) = a + b$ for concatenation or sorting;

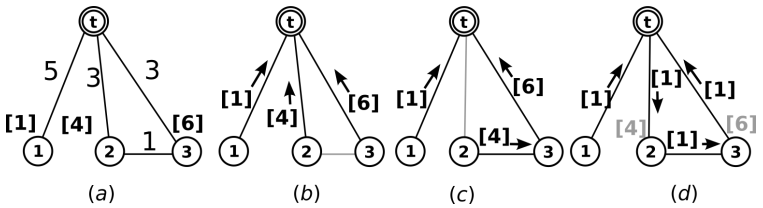


Figure 2: Different μ lead to different plans: (a) a sample task; (b) optimal plan for $\mu(a,b) = a + b$; (c) optimal plan for $\mu(a,b) = \max(a,b)$; (d) optimal plan for $\mu(a,b) = \min(a,b)$.

- $\max(a,b) \leq \mu(a,b) \leq a + b$ for set union (e.g., word count).

Fig. 2 shows how the choice of μ can affect the optimal aggregation plan. Fig. 2a shows chunks of size 1 at vertex 1, of size 4 at vertex 2, and of size 6 at vertex 3, and the goal is to aggregate them at vertex 0. For $\mu(a,b) = a + b$, the optimal plan is to move each chunk to the root separately (Fig. 2b). For $\mu(a,b) = \max(a,b)$, it is cheaper to first move the chunk of size 4 along edge $2 \to 3$ and merge it, then move the resulting chunk of size 6 to the root (Fig. 2c). Finally, for $\mu(a,b) = \min(a,b)$ the optimal plan is to traverse the whole graph with the smallest chunk, merging larger ones along the way (Fig. 2d). Thus, even in a simple example the aggregation plan can change drastically depending on μ. We get the following optimization problem.

Problem 3.1 (CAM — compute-aggregate minimization)**.** *Given an undirected connected graph $G = (V, E)$, cost function c, a target vertex t, a set of initial data chunks C, and an aggregation size function μ, the* CAM$[\mu]$ *problem is to find an aggregation plan P such that* $\text{cost}(P)$ *is minimized.*

Interestingly, unless the aggregation size function is well-behaved and at the same time there are constraints on the graph structure, there is not much we can do in the worst case.

Theorem 1. *Unless $P = NP$, there is no polynomial time constant approximation algorithm for* CAM *without associativity constraint on μ even if G is restricted to two vertices.*

Proof. We can encode an NP-hard problem in choosing the correct order of merging for a non-associative μ. For example, consider an instance of the knapsack problem with weights $w_1, \ldots, w_n$, unit values, and knapsack size W; then we have n chunks of size $w_1, \ldots, w_n$, and μ is defined as follows: if either $x = 0$, $y = 0$, or $x + y = W$ then $\mu(x,y) = 0$; else $\mu(x,y) = x + y$. This way, if we can fill the knapsack exactly the total resulting weight will be zero, and if not, it will be greater than zero, leading to unbounded approximation ratio unless we can solve the knapsack problem. □

In this work, we investigate two main degrees of freedom that the CAM problem has: network topology graph G and aggregation size function μ. Let us begin with μ. In the next section, we provide a taxonomy with respect to how fast aggregation size functions μ grow.

4. A Taxonomy of Aggregation Functions

There exist different types of big data applications, a large variation in datacenter network topologies, and countless data distributions, all of which collectively define constraints for a compute-aggregate task. Handling each and every variation of these constraints separately does not scale, so a generalized decision procedure should be used to construct an aggregation plan. In this section, we present such a procedure and show worst-case guarantees for every choice.

Intuitively, stricter constraints may lead to better decisions, both in terms of the *cost of an aggregation plan*, which is our primary objective, and in terms of *performance* (running time). For instance, if the network graph is a tree, it might be possible to construct an aggregation plan in linear time. Similarly, a better algorithm (in terms of the resulting cost) can be chosen under certain constraints on the aggregation

size function. Thus, a possible solution may have to account for *network topology*, *aggregation size function*, and *initial chunk distribution*. Chunk distribution varies from one problem instance to another, and is unlikely to provide useful information since it is expected to be roughly uniform (big data storage systems try to achieve even load distribution). Although there are common network topologies, such as *hypercube*, *fat-tree*, or *jellyfish*, there are plenty of variations and exceptions. So, while algorithms specifically tailored for, e.g., the hyper-cube topology [9] remain a valid topic for future study, in this work we mostly consider the aggregation size function, with only two special cases.

First, a tree is a topology that is both widespread and has a high potential for better algorithmic solutions; we call the CAM problem where G is a tree TCAM (tree CAM). Second, sometimes it is reasonable to limit aggregation to only those nodes that contain data chunks initially, either for security reasons or due to the need for additional resource provisioning on intermediate nodes that may significantly increase latency, while nodes with initial chunks usually already have computing resources for a preprocessing stage. If either security or provisioning impose the aforementioned restriction then a network graph G can be reduced to a complete graph over the nodes that contain chunks, and we call this special case CCAM (complete CAM).

We have summarized our theoretical results in Table 1. Cells in the table show approximation ratios for various cases of the aggregation size function and μ and various problem settings (CAM, TCAM, and CCAM) together with references to specific theorems and corollaries where these results are proven below. Rows of Table 1 correspond to various special cases of μ, ranging from the slowest to the fastest growing functions, and columns correspond to the three problem variations. In the table, α denotes the approximation factor for the polynomial algorithm for the minimum Steiner tree problem (MSTT) used as a subroutine in our algorithms. An approximation ratio of 1 means that there is a polynomial-time optimal algorithm for this case, and ∞ means that there can be no constant approximation factor achieved by a polynomial algorithm unless P=NP. As a visualization, in order to simplify the two-dimensional structure of our result, we have also presented them in three figures corresponding to the three cases of topologies we consider: Fig. 3 shows the results for CAM, Fig. 4 shows approximation factors for TCAM, and Fig. 5 for CCAM. On each figure, the horizontal axis corresponds to how fast μ grows, and each rectangular block refers to a result in the form of an approximation ratio for the corresponding problem and a reference to a specific theorem proving it for the corresponding algorithm.

4.1. General case

In this subsection, we assume no constraint on the behavior of the aggregation size function μ. To devise a meaningful strategy in this general case, we begin by considering a simpler setting where all chunks have size x, and a merge result of two chunks has the same size as one of the chunks, i.e., $\mu(x, x) = x$. In this case, when paths of two chunks intersect it is always better to merge at the intersection. Thus an optimal aggregation plan always proceeds along a tree subgraph of G, and the weight of that tree multiplied by x equals the aggregation plan cost since μ does not change weights. Thus, the problem reduces to finding a minimum weight tree that connects a given set of vertices, which is a well-studied minimum Steiner tree problem [34], MSTT, that has many constant approximation algorithms. Using one of those, we build our first aggregation plan construction algorithm `steiner_rec` (Alg. 2); we summarize the above discussion in the following theorem.

Theorem 2. *If there is a polynomial α-approximate algorithm for* MSTT*, then* `steiner_rec` *runs in time $O(V^2 \log V)$ and provides an α-approximation for the special case when* $\text{size}(x) = S$ *for any $x \in C$, and $\mu(S, S) = S$.*

The `steiner_rec` algorithm has a number of interesting properties; e.g., it does not require any knowledge of μ or even chunk sizes. The infrastructure can run `steiner_rec` even before preprocessing (in map-reduce terminology, before a *map* phase).

It turns out that in the general case, the price of using `steiner_rec` does not exceed the *ratio between the largest and smallest intermediate chunk*. We denote by $W_C[\mu]$ the maximal aggregate size of a subset of chunks from the set C, $W_C[\mu] = \max_{C' \subseteq C}\{\mu(C')\}$; it is well defined since μ is associative and commutative. We also denote by $w_C[\mu]$ the corresponding minimal aggregate size, $w_C[\mu] = \min_{C' \subseteq C}\{\mu(C')\}$.

	CAM		TCAM		CCAM	
Arbitrary μ (general case)	$\alpha\frac{W_C[\mu]}{w_C[\mu]}$	*Thm. 3*	$\frac{W_C[\mu]}{w_C[\mu]}$	*Cor. 6*	$\frac{W_C[\mu]}{w_C[\mu]}$	*Cor. 6*
$\mu(a,b) \le \min(a,b)$	∞	*Thm. 7*	∞	*Thm. 7*	∞	*Thm. 7*
$\mu(a,b) = \min(a,b)$	2α	*Thm. 13*	1	*Thm. 15*	2	*Cor. 14*
$\min(a,b) \le \mu(a,b) \le \max(a,b)$	$\alpha\frac{W_C}{w_C}$	*Thm. 9*	$\frac{W_C}{w_C}$	*Thm. 10*	$\frac{W_C}{w_C}$	*Thm. 10*
$\mu(a,b) = \max(a,b)$	4α	*Thm. 16*	1	*Thm. 12*	4α	*Thm. 16*
$\max(a,b) \le \mu(a,b) \le a+b$	$2N\sqrt{\alpha V\frac{c_{\max}}{c_{\min}}}$	*Thm. 11*	1	*Thm. 12*	$2N\sqrt{\alpha V\frac{c_{\max}}{c_{\min}}}$	*Thm. 11*
$\mu(a,b) \ge a+b$	1	*Thm. 8*	1	*Thm. 8*	1	*Thm. 8*

Table 1: A taxonomy of our theoretical results in relation to the aggregation size function μ and specific problem (CAM, TCAM, CCAM). Numbers show approximation factors of polynomial algorithms presented in the corresponding theorems, and α denotes the approximation factor for a polynomial algorithm solving the MSTT problem.

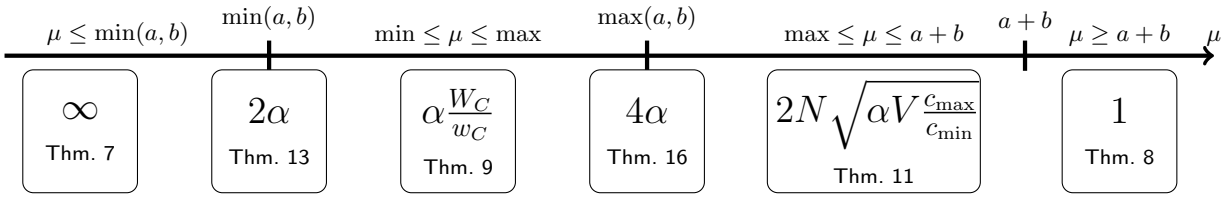


Figure 3: A visual summary of our theoretical results for the CAM problem.

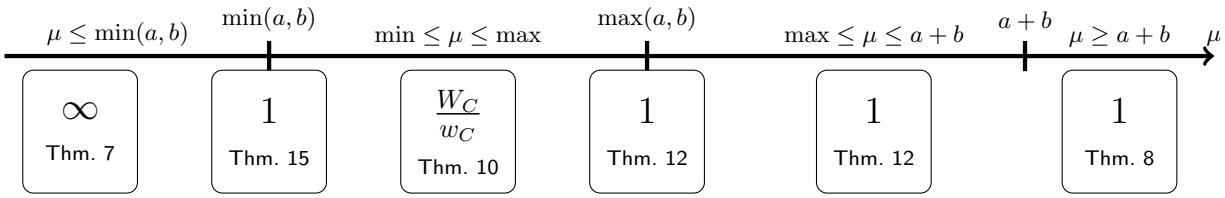


Figure 4: A visual summary of our theoretical results for the TCAM problem.

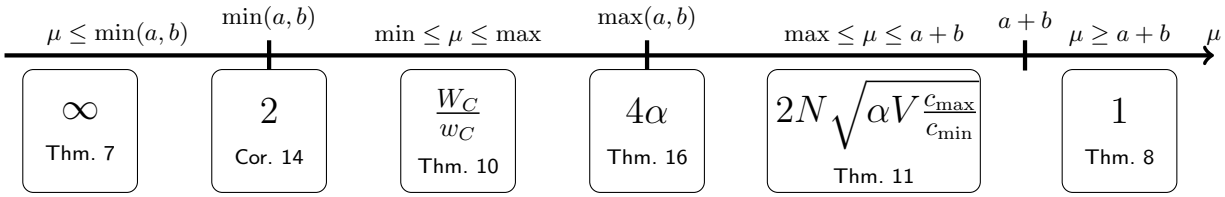


Figure 5: A visual summary of our theoretical results for the CCAM problem.

Theorem 3. *If there exists a polynomial α-approximate algorithm for* MSTT*, then there exists a polynomial algorithm that solves* CAM$[\mu]$ *with approximation factor* $\alpha\frac{W_C[\mu]}{w_C[\mu]}$.

Proof. First, note that any algorithm, even optimal, has to traverse at least the Steiner tree of G in total size, and has to carry at least weight w_c over each edge. The approximate algorithm begins by constructing the approximate Steiner tree with approximation ratio α, and then carries all chunks along this tree to the root, merging the chunks at first opportunity; in this process, the maximal possible chunk size is W_c, and it is carried over at most α times longer distance than in the actual Steiner tree, getting the approximation bound. □

As for the actual algorithms, there is a well-known 2-approximation to MSTT based on a minimum spanning tree (MST) of the distance closure G^* of G; a much more involved construction leads to the best

Algorithm 1 `steiner`$(G, V' \subseteq V(G))$

if G is a tree **then**
    return unique subtree $T_{V'}$ covering V'
else if $V(G) = \{\mathrm{v}(x) : x \in C\}$ **then**
    return `min_spanning_tree`(G)
else
    return `steiner_tree_approx`$(G, \{\mathrm{v}(x) : x \in C\})$

Algorithm 2 `steiner_rec`(G, t, C)

$P \leftarrow (); T \leftarrow$ `steiner`$(G, \{\mathrm{v}(x) : x \in C\})$
for $v \in T$ in decreasing order of depth(v) **do**
    ▷ Denote $C(v) = \{x \in C : \mathrm{v}(x) = v\}$
    while $|C(v)| > 1$ **do**
        P.`append`(**aggr**(x, y)), where $x, y \in C(v)$
        C.`update`(**aggr**(x, y))
    if $\exists\, x \in C(v)$ **and** $\exists$ parent(v) **then**
        P.`append`(**move**$(x, \text{parent}(v))$)
        C.`update`(**move**$(x, \text{parent}(v))$)
return P

known approximation ratio of $\ln 4 + \varepsilon \leq 1.39$ [35]. Although `steiner_rec` does not depend on either μ or chunk sizes, the approximation factor in Theorem 3 includes both.

This result improves for special cases of TCAM and CCAM. The following two theorems show that Algorithm 1 can improve the result when moving from CAM to these two more constrained problems.

Theorem 4. *If every vertex in G contains a data chunk, then* MSTT *can be solved exactly in polynomial time.*

Proof. In this case, MSTT is equivalent to MST. □

Theorem 5. *If G is a tree, then* MSTT *can be solved exactly in polynomial time.*

Proof. There is only one subtree in G that connects a given set of vertices, and it can be found in polynomial time. □

Theorem 4 and Theorem 5 essentially say that in these special cases we have 1-approximation algorithms for MSTT. Theorem 3 and this observation together imply the following.

Corollary 6. *There exist polynomial algorithms that solve* CCAM$[\mu]$ *and* TCAM$[\mu]$ *on a set of chunks C with approximation factor* $\frac{W_C[\mu]}{w_C[\mu]}$.

However, for many μ, including important ones (e.g., set union), Theorem 3 and Corollary 6 provide rather weak approximations; in particular, we would like to have approximation ratios independent of chunk sizes and specific values of μ since in practice $\frac{W_C}{w_C}$ may be very high. Unfortunately, it is impossible even for a restricted class of functions μ that reduce the weights, that is, for functions smaller than min.

Theorem 7. *There exists an aggregation size function μ such that $\forall a, b\ \mu(a, b) \leq \min(a, b)$, and no polynomial time constant approximation algorithm for* CCAM$[\mu]$ *or* TCAM$[\mu]$ *exists unless $P = NP$.*

Proof. Consider a complete graph G where the root r contains an infinitely large chunk, all non-root vertices are terminals, edges between two terminal vertices cost 1, and edges between a terminal vertex and the root cost ∞. Then we can reduce the Set Cover problem to an instance of CAM$[\mu]$ on G.

Given an instance of Set Cover, where a set S must be covered with a minimal number of m subsets $S_i \subseteq S$, we define $n(S_i)$ as the number with binary representation equivalent to $S \setminus S_i$ (for some fixed order of elements in the set). We encode S by a chunk of size 0 and any other subset $A \subset S$ by a chunk of size $n(A) + 4n \times 2^{|S|}$. The aggregation size function for two chunks corresponding to subsets A and B produces a chunk of size $n(A \cup B)$.

Now, if there exists a set cover $S_{i_1}, S_{i_2}, \cdots, S_{i_k}$ then there is a solution to CAM$[\mu]$ of size $k \times 4n \times 2^{|S|} + c$, where $c \leq n \times 2^{|S|}$ (we can aggregate S_{i_j} in any order and then aggregate the rest with a zero chunk we obtained). On the other hand, if there exists a solution to CAM$[\mu]$ of size $g \times 4n \times 2^{|S|} + c$, where $c \leq n \times 2^{|S|}$, then there exists a solution to Set Cover of size g (to achieve this solution of CAM$[\mu]$ we have to obtain 0 in

at most g agregations). Thus, a constant approximation for CAM[μ] implies a constant approximation of Set Cover which is impossible unless $P = NP$.

For TCAM[μ], consider the following transformation of G to a tree T_G: remove all the edges; introduce a new vertex c; connect c with r by an edge of weight ∞ and with the rest of G's vertices by edges of weight 1. Changing G to T_G does not increase cost more than twice (we traverse two edges now). Thus, the transformation preserves approximations, which again implies that TCAM[μ] does not have constant approximations unless $P = NP$. □

Since CCAM is a strict subset of CAM, there is no constant-approximation solution for CAM either.

In the next subsection, we proceed to finer distinctions between aggregation size functions that can make CAM easier: where μ is known to be contained between a pair of known functions. Specifically, following the negative result of Theorem 7, we concentrate on the functions μ that grow at least as fast as the minimum of their arguments.

4.2. Range-bounded aggregation size functions

Depending on the application, the value of μ may be known to lie in a certain range. For example, if **aggr** represents *set union* then $\mu(x, y) \in [\max\{x, y\}, x + y]$, and if **aggr** represents *outer join* then $\mu(x, y)$ is likely to be always larger than $x + y$. We show a taxonomy of algorithms for different μ; the goal of this subsection is to find a decomposition of a general case into subranges that allow for better worst-case guarantees.

Theorem 7 showed that aggregation size functions that reduce size too much are provably hard. On the other side of the spectrum, where $\mu(x, y) \geq x + y$, there is an optimal solution: bring all chunks to the sink. The intuition is opposite to that of using MSTT: it never makes sense to merge, and every chunk's path can be optimized individually. The algorithm `SPT` does so by calculating shortest paths to root, a process that does not depend on either chunk sizes or precise values of μ and, similar to `steiner_rec`, can run in parallel with a local preprocessing phase. Optimality is proven in the next theorem.

Theorem 8. *If $\mu(a, b) \geq a + b$ for all a, b then there exists a polynomial optimal algorithm for* CAM[μ], CCAM[μ], *and* TCAM[μ]; *for* TCAM[μ] *the running time is $O(|C| + |G|)$.*

Proof. In this case, it does not make sense to merge chunks at all; the optimal algorithm is to bring all chunks separately to the sink. Formally, consider an optimal aggregation plan for CAM that merges two chunks not at the sink. Next, consider a transformed plan that carries both chunks separately and treats them separately until the final vertex. Since $\mu(a, b) \geq a + b$, the total cost will not increase in this transformation, and we can sequentially get a plan without any merging without increasing the costs. The optimal strategy without merging is to move all chunks to the root along shortest paths, which can be computed in polynomial time. Because TCAM and CCAM are strict subsets of CAM, `SPT` is optimal for them too. For TCAM there is no need to calculate shortest paths since paths are unique, and the running time becomes linear. □

We have found that for $\mu(x, y) \in (-\infty, \min\{x, y\}]$ the problem is inapproximable (Theorem 7), and for $\mu(x, y) \in [x+y, \infty)$ there is an optimal algorithm (Theorem 8). We split the remaining range $[\min\{x, y\}, x+y]$ at $\max\{x, y\}$ for two reasons. First, in practice max is a valid bound for many applications: set intersection, set union, outer join (symmetric or asymmetric); thus, the infrastructure often knows on which side of max μ lies. Second, theoretic results below show that max is an interesting demarcation line for worst-case guarantees: below max chunk sizes are a primary factor, and above max the graph structure begins to dominate.

If $\mu(x, y) \in [\min(x, y), \max(x, y)]$, we can replace the ratio $W_C[\mu]/w_c[\mu]$ (Theorem 3), which depends on μ, with a simpler one that depends only on chunk sizes. In the next theorem $W_C = \max_{x \in C}\{\text{size}(x)\}$, $w_c = \min_{x \in C}\{\text{size}(x)\}$.

Theorem 9. *If $\min\{a, b\} \leq \mu(a, b) \leq \max\{a, b\}$ for all a, b and there exists a polynomial α-approximate algorithm for* MSTT, *then there exists a polynomial algorithm that solves* CAM[μ] *with approximation factor $\alpha \frac{W_C}{w_C}$.*

Using Theorems 4 and 5, it is possible to improve the result of the previous theorem for CCAM and TCAM.

Corollary 10. *If* $\min\{a,b\} \le \mu(a,b) \le \max\{a,b\}$ *then there exist polynomial algorithms that solve* CCAM$[\mu]$ *and* TCAM$[\mu]$ *with approximation factor* $\frac{W_C}{w_C}$.

For $\mu(x,y) \in [\max\{x,y\}, x+y]$, the last remaining range, we employ a mix of `SPT` and `steiner_rec`: merge chunks above a certain threshold with `steiner_rec`, and below this threshold with `SPT`. The end result is a worst-case bound independent of both μ and chunk sizes, as presented in the following theorem.

Theorem 11. *If for all* a *and* b $\max(a,b) \le \mu(a,b) \le a+b$, *then there exists a* $2N\sqrt{\alpha V \frac{c_{\max}}{c_{\min}}}$*-approximate polynomial algorithm for* CAM$[\mu]$, *which we call* `RECH_MStTSplit`, *where* V *is the number of vertices in* G, N *is the number of chunks,* $c_{\max}$ *is the cost of the most expensive edge in* G, $c_{\min}$ *is the cost of the cheapest edge, and* α *is an approximation factor for* MSTT.

Proof. The idea of the algorithm is as follows. We split all chunks C into two sets: chunks with weight at least δM go into set C_1 and chunks with weight smaller than δM go into C_2, where M is the weight of the maximal chunk and δ is a constant to be defined later, so $C = C_1 \cup C_2$. Next we solve two separate CAM problems. For C_1 we run the general algorithm from Theorem 3, and for C_2 we run the algorithm from Theorem 8 that we used for μ such that $\mu(a,b) \ge a+b$.

The first algorithm yields an $\frac{\alpha N}{\delta}$-approximate solution, and the total weight of the second solution does not exceed $\delta MVNc_{\max}$, where N is the number of chunks. Let W be the weight of the optimal solution. Now, since $\max(a,b) \le \mu(a,b)$, and W is at least the weight of the optimal solution for C_1, we can conclude that the weight of the solution for C_1 is at most $\frac{\alpha V}{\delta}W$. On the other hand, since $W \ge Mc_{\min}$, the weight of the solution for C_2 is at most $\delta V^2 \frac{c_{\max}}{c_{\min}}W$. Now if we choose $\delta = \sqrt{\frac{ac_{\min}}{Vc_{\max}}}$ to minimize the total result, the total weight of both solutions will be $2N\sqrt{\alpha V \frac{c_{\max}}{c_{\min}}}W$. □

To improve the above theorem we cannot apply Theorem 4 to get rid of α for CCAM or TCAM since it uses the Steiner tree only for a subset of chunks. But, remarkably, we can do better for TCAM: the following theorem proves that between max and "+" `steiner_rec` is optimal for TCAM.

Theorem 12. *There exists a polynomial optimal algorithm for the* TCAM$[\mu]$ *problem for any* μ *such that* $\forall a,b$ $\max(a,b) \le \mu(a,b) \le a+b$.

Proof. The algorithm is similar to Theorem 3: move chunks towards the vertex t, merging them in intermediate nodes.

Consider an arbitrary subtree T of G. All data chunks from T have to be eventually moved upwards using parent edge e (if $T \ne G$). Minimal cost of this operation is clearly $\le s_T$, where s_T is the size of the aggregation result of all chunks from $C|_T$. Can it be less?

Assume the opposite: there is a set of data chunks X that will be moved upwards through e s.t. $C_T \subseteq X^* = \bigcup_{x \in X} x^*$ and $\sum_{x \in X} \text{size}(x) < s_T$, where x^* is the set of initial chunks that contributed to x. If $C_T \subsetneq X^*$, we can throw away $X^* \setminus C_T$ without any increase in the cost because μ is at least max. Also, since μ does not exceed the sum, we can aggregate X with no cost increase. The resulting chunk has size s_T, which is a contradiction: by construction, each upward edge e from a subtree T will add exactly s_T. □

In the next subsection we refine ranges to points and investigate situations where μ is a specific function, that is, when actual aggregation size is expected to be close to that function.

4.3. Specific aggregation size functions

The discussion above has shown a way to gradually exploit available knowledge to improve worst-case guarantees and runtime performance while not being tied to any particular big data application. Yet, to achieve the best performance one might have to use specifically tailored algorithms. We capture such algorithms by considering situations where the aggregation size function μ is known precisely. Since μ does not completely define an application, these algorithms can still be reused across multiple applications.

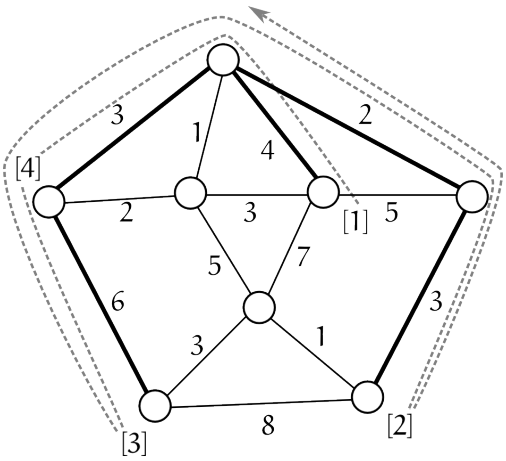


Figure 6: Sample solution from Theorem 13. Steiner tree approximation is shown with bold lines, and the resulting path of the smallest chunk is shown by a dotted arrow. Note that this chunk never visits one edge more than twice.

We present two particular examples where exact knowledge of μ leads to algorithms whose approximation factor is constant, that is, independent of the chunk set C, the graph G, and μ.

These examples deal with intermediate points between the ranges discussed in the previous subsection: (1) below min, (2) from min to max, (3) from max to sum, and (4) above sum. Out of the three delimiting points between these ranges—min, max, and sum—sum has already been covered by an optimal algorithm from Theorem 8. In this section, we concentrate on min and max.

Theorem 13. *If there exists a polynomial α-approximate algorithm for* MSTT*, then there exists a polynomial 2α-approximate algorithm for* CAM[min].

Proof. Given an instance (G, t, C) of the CAM[min] problem, first we find an α-approximation T to the MSTT instance $(G, V' = \{t\} \cup \{\mathrm{v}(x) : x \in C\})$. Then, we construct an aggregation schedule by taking a data chunk with the smallest size and walking it through T. The resulting cost does not exceed $2m \cdot w(T)$, where m is the size of the smallest chunk. Similar to Theorem 17, an aggregation schedule defines a subgraph $H \supseteq V'$, and so incurs the cost of at least $m \cdot w(H)$. A sample solution for this algorithm is shown on Fig. 6. □

Corollary 14. *There exists a polynomial 2-approximate algorithm for* CCAM[min].

TCAM[min] can be solved in polynomial time with dynamic programming.

Theorem 15. *There exists an optimal algorithm for* TCAM[min] *running in polynomial time.*

Proof. The optimal algorithm uses dynamic programming. Consider an instance $(G = (V, E), t, C)$ of the TCAM[min] problem, where G is a tree with a root t. For every vertex $v \in V$ we compute $\mathrm{mc}(v)$, the size of the smallest chunk in a subtree T_v rooted at v, and for every $c \in C$ we compute $\mathrm{dp}(v, \mathrm{size}(c))$, an optimal solution for T_v with an additional chunk of size $\mathrm{size}(c)$ at v. We can find $\mathrm{mc}(v)$ for every vertex in linear time by running depth first search. If $\mathrm{dp}(v, \mathrm{size}(c))$ are known for every $u \in \texttt{children}(v)$ then $\mathrm{dp}(v, \mathrm{size}(c))$ can be computed as

$$\mathrm{dp}(v, \mathrm{size}(c)) = \sum\nolimits_{u \in \texttt{ch}(v)} \min\{2 \cdot \mathrm{size}(c) \cdot \mathrm{d}(v, u) + \mathrm{dp}(u, \mathrm{size}(c)), \mathrm{mc}(u) \cdot \mathrm{d}(v, u) + \mathrm{dp}(u, \mathrm{mc}(u))\}.$$

Now $\mathrm{dp}(t, \mathrm{mc}(t))$ contains the cost of an optimal aggregation plan, which can be found with backtracking. □

Our second approximate algorithm, which solves CAM[max], is also based on the MSTT problem, but with a different construction.

Theorem 16. *If there exists a polynomial α-approximate algorithm for* MStT*, then there exists a polynomial 4α-approximate algorithm for* CAM[max]*, which we call* `RECH_MStTMax`*.*

Proof. Let the maximal chunk size be equal to M. We separate data chunks into several subsets: the first with chunks with sizes in $(\frac{M}{2}, M]$, the second in $(\frac{M}{4}, \frac{M}{2}]$, and so on. The idea is to build an approximate solution for the first subset, extend it to a solution for the first two subsets, and so on.

First, we build a Steiner tree for the root and chunks in the first subset and solve the problem on this tree. This solution is at least 2α-competitive, where α is the MStT approximation factor: edges used by a different solution must connect every chunk to the root, so their total cost is at least the cost of a minimum Steiner tree, and their sizes are at least $M/2$. Next, we merge the tree obtained on the first iteration into a single vertex, throw away the first subset, build a Steiner tree for the second subset, solve the problem for this tree, and so on. Suppose that there were k such subsets. Since we move chunks of size at most $\frac{M}{2^{i-1}}$, and merging vertices does not increase the weight of a Steiner tree, the cost of the ith subset does not exceed $\frac{M}{2^{i-1}}\alpha\mathrm{ST}(i, i-1)$, where $\mathrm{ST}(i,j)$ is the optimal Steiner tree weight for chunks with sizes in $(\frac{M}{2^i}, \frac{M}{2^j}]$. Thus, the total cost does not exceed $2\alpha M \sum_{i=1}^{k} \frac{\mathrm{ST}(i,i-1)}{2^i}$.

For the lower bound, we count the cost of all data movements across every edge. The total cost of all edges with chunks of mass at least $M/2$ moved along them is bounded by $\mathrm{ST}(1,0)$, so the cost is bounded by $\frac{M}{2}\mathrm{ST}(1,0)$; repeating the process for $\frac{M}{4}$, $\frac{M}{8}$ and so on, we get in total $M\sum_{i=1}^{k}\frac{\mathrm{ST}(i,0)}{2^i}$. Some of the edges are counted more than once: an edge with the largest chunk of size $\frac{M}{2^j}$ moved along it has been counted once with a factor of $\frac{M}{2^j}$, once with $\frac{M}{2^{j+1}}$, and so on, but the real lower bound for this edge is $\frac{M}{2^j}$. Thus for every edge we have an extra factor of $1+\frac{1}{2}+\frac{1}{4}+\ldots\leq 2$, and the optimal cost is at least $\frac{M}{2}\sum_{i=1}^{k}\frac{\mathrm{ST}(i,0)}{2^i}$. Since $\mathrm{ST}(i,0)\geq\mathrm{ST}(i,i-1)$, the total approximation factor is $\frac{2\alpha M}{M/2}=4\alpha$. □

Unfortunately, we cannot take advantage of Theorem 4 to improve the result of Theorem 16 for CCAM: even if there is a chunk in every vertex, we need to build a Steiner tree only for a subset of chunks. But Theorem 12 implies that TCAM[max] has an optimal solution since $\max\{x,y\}$ lies trivially in $[\max\{x,y\}, x+y]$.

However, results for CAM[min] and CAM[max] cannot be significantly improved because both are NP-hard, as we show in the following theorems

Theorem 17. *If there exists $a>0$ such that $\mu(a,a)=a$ then* CAM[μ] *is NP-hard and does not have less than $\frac{19}{18}$-approximate polynomial algorithms even if all edge weights are equal, unless P=NP.*

Proof. The proof is by reduction from the MStT problem. Given a MStT instance $(G, w, V'\subseteq V)$, we place data chunks of size a in each vertex of V' except one, which becomes the sink. Any aggregation schedule defines a connected subgraph H of G that contains all vertices from V'. The minimal cost is $a\cdot w(H)$, where $w(H)=\sum_{e\in E(H)}w(e)$, since all transmitted data chunks have size a, and we can always avoid transmitting more than one chunk across one link. Any spanning tree of H defines a Steiner tree for V', and vice versa, any Steiner tree T defines an aggregation schedule with cost $a\cdot w(T)$. □

Theorem 18. *The* CAM[min] *problem is NP-hard even if all edge weights are equal, and each vertex is required to contain a data chunk.*

Proof. This time we reduce the Hamiltonian cycle problem. Given a Hamiltonian cycle problem instance G, we choose the sink arbitrarily, place a chunk of size 1 in the sink and chunks of size $|V|^2$ in every other vertex. The optimal solution travels with weight 1 along a Hamiltonian cycle. □

5. Evaluation

To evaluate the relative performance of the proposed heuristics, we have compared them in practical settings on network topologies generated by the *topobench* library [36]. Most of the topologies available in *topobench* are designed for network switches; since we are interested in compute-aggregate tasks which are usually specific for servers and datacenters, out of all topologies in the library we analyzed topologies that

can serve for both server adjacency and switch adjacency, namely *JellyFish* [37], *Hypercube*, and *Small World Datacenter Ring* (SWDC Ring) [38]. For example, in *FatTree* servers are connected only to the leaf routers, so these networks do not really reflect server-to-server topology.

We compare seven algorithms:

- `CAMH`, a greedy algorithm that on every step chooses a pair of chunks $\boxed{x}$ and $\boxed{y}$ and vertex v such that after $\boxed{x}$ and $\boxed{y}$ are merged at v the cost of moving all resulting chunks along shortest paths plus the cost of moving $\boxed{x}$ and $\boxed{y}$ to v is minimized;
- `MCAMH`, a variation of `CAMH` where $v \in \{\mathrm{v}(\boxed{x}), \mathrm{v}(\boxed{y})\}$;
- $\texttt{MCAMH}_{\leq}$, a restriction of `MCAMH` where we always move the smaller chunk;
- `RECH_MST`, a variation of Algorithm 2 that merges along the minimum *spanning* tree;
- `RECH_MStT` (Algorithm 2);
- `RECH_SPT`, another variation of Algorithm 2 that uses a tree of shortest paths;
- `RECH_MStTMax` (Theorem 16);
- `RECH_MStTSplit` (Theorem 11).

Each of Figs. 7–9 shows one sample topology of each kind and analyzes heuristic performance with respect to the properties of compute-aggregate tasks. These properties are defined by the following parameters:

- number of initial data chunks $n_C = |C|$,
- maximal edge weight W, and
- the range s of allowed data chunks.

The range s is defined by the median center of the interval s_{avg} and its radius Δ_s, so $s = [s_{\text{avg}} - \Delta_s, s_{\text{avg}} + \Delta_s]$. To generate CAM instance for a given network topology, we uniformly select the weights from 1 to W and uniformly place n_C data chunks, each of which gets size uniformly from s. The three figures correspond to three practical aggregation functions:

- $\mu(x, y) = \exp(\log|x| + \log|y|)$ (Fig. 7), which roughly estimates the result of joining two dictionaries (*WordCount*),
- $\mathbf{aggr}(x, y) = \mathrm{SetUnion}(x, y)$ (Fig. 8), and
- $\mathbf{aggr}(x, y) = \mathrm{ChooseBest}(x, y)$ (Fig. 9), which compares the (randomly chosen) priorities of two chunks.

For each point on the graph, we ran 1000 random experiments, taking the average cost. We have made an implementation of all algorithms and sample topologies available at [39].

The results show that `RECH_MStT`, `RECH_MStTMax`, and `RECH_MStTSplit` have the best results among tree-based heuristics. This is expected: unlike `RECH_MST`, `RECH_MStT` works on the local level and does not optimize the part of the tree not directly involved in aggregation, but also does not optimize every chunk separately as `RECH_STP` does.

Among potential-based heuristics, `CAMH` clearly wins for WordCount (it is even better than `RECH_STP`) but fares much worse for SetUnion and ChooseBest. `CAMH`, `MCAMH`, and $\texttt{MCAMH}_{\leq}$ use the value of μ, so their behaviour changes significantly between SetUnion and ChooseBest: the latter can bring a larger improvement if we get lucky and get a light high-priority chunk. The relative performance of other algorithms does not change significantly with aggregation function. `CAMH`, `MCAMH`, and $\texttt{MCAMH}_{\leq}$ represent a tradeoff between expressivity (larger search space) and a higher chance to end up in a worse solution. `MCAMH` serves as the middle ground and takes the leading place almost everywhere among these three heuristics; `CAMH` wins on

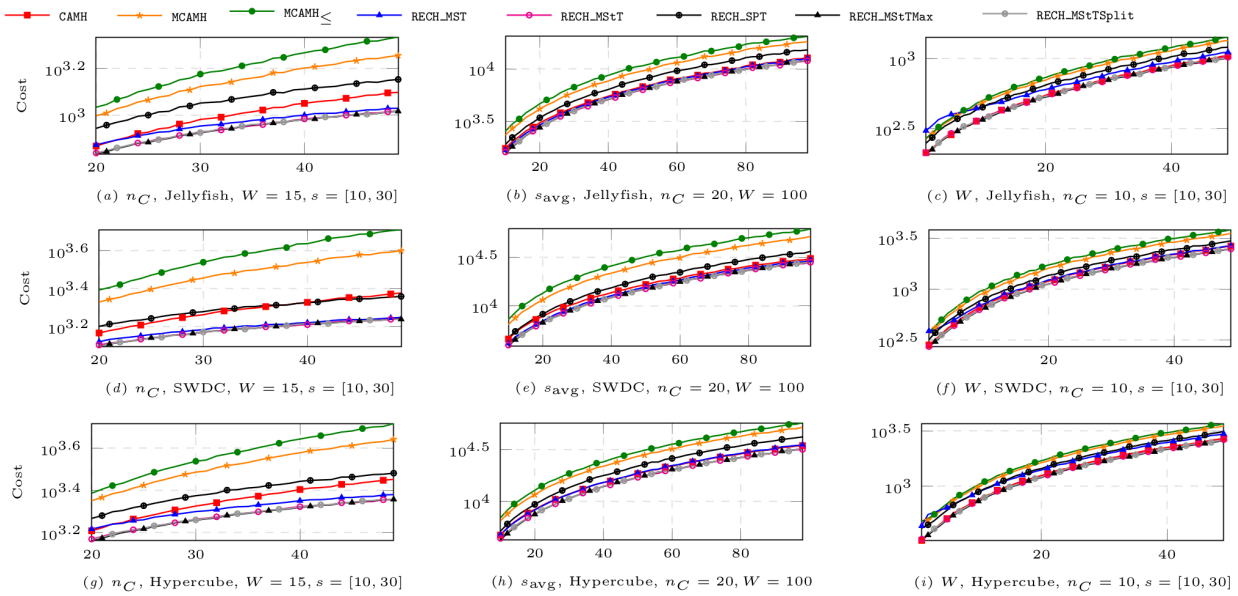


Figure 7: Compute-aggregation costs for $\mu(x, y) = \exp(\log x + \log y)$. Rows correspond to topologies: (a-c) JellyFish; (d-f) SWDC Ring; (g-i) Hypercube. Columns correspond to the horizontal axis: (a,d,g) number of chunks; (b,e,h) average chunk size s_{avg} ($\Delta_s = 10$); (c,f,i) maximal edge weight.

WordCount, where merges are more regular and predictable, and the local behavior of **aggr** does not mislead `CAMH`.

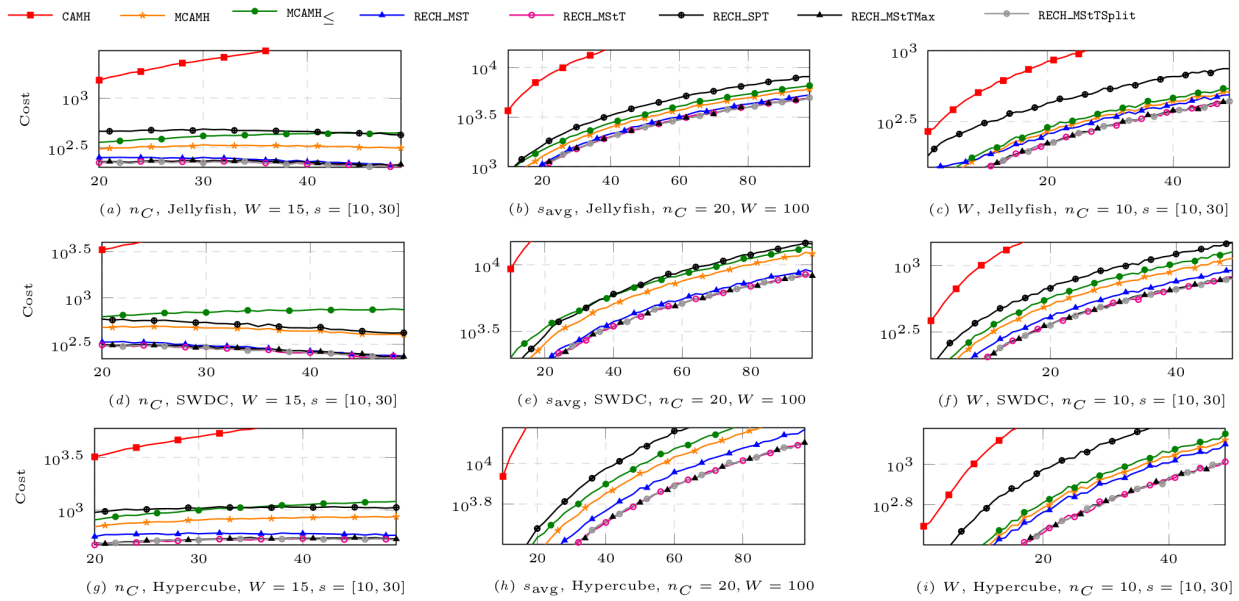
In general, heuristics based on Steiner trees are the best throughout all experiments, with `RECH_MStT`, `RECH_MStTMax`, and `RECH_MStTSplit` occupying the top three places almost everywhere, even though they do not pay that much attention to **aggr**. `RECH_MStTMax` starts losing for the ChooseBest aggregation function since it was designed specifically for **aggr** = max, and ChooseBest is very different. As the number of chunks grows, `RECH_MST` becomes closer to `RECH_MStT` and its variations: chunks cover more vertices, and Steiner trees become more similar to spanning trees. `RECH_SPT` is worse than `RECH_MstT`, as expected; it even loses to `MCAMH` and $\texttt{MCAMH}_{\leq}$ in many cases but regains some ground for more chunks. Interestingly, `RECH_SPT` works better on SWDC Ring; this may be because this topology is bounded (it is a cycle with four random chords coming out of each vertex), so the tree of shortest paths covers most edges in the cycle and more chunks.

There are two main conclusions to be drawn from the evaluation study. First, there is a substantial difference in the *relative* performance of algorithms for different choices of **aggr** (e.g., `RECH_SPT` vs `CAMH` for SetUnion and ChooseBest) even if all other parameters (such as, e.g., the initial chunk distribution) are kept the same, which emphasizes the need for extra information about applications and supports our choice of an aggregation size function as a basis for the presented taxonomy (see also Table 1). Second, algorithms based on Steiner tree heuristics have the best performance and are almost indistinguishable from each other. Not only does it confirm analytic results from Section 4, but it also shows the superiority of precisely those algorithms that pay the most attention to the network topology and intermediate aggregations; both aspects are either neglected or treated trivially in traditional designs.

In general, the results of our evaluation study reiterate on the importance of holistic application/network optimization that we advocate for, and pave the way for designers of compute-aggregate infrastructures to extend the taxonomy presented in Table 1 if better performance is needed.

6. Conclusion

In this work, we explore infrastructural improvements in datacenters dealing with compute-aggregate jobs. We propose to divide the aggregation plan for a compute-aggregate job into two phases: on the first



(a) n_C, Jellyfish, $W = 15, s = [10, 30]$ (b) s_{avg}, Jellyfish, $n_C = 20, W = 100$ (c) W, Jellyfish, $n_C = 10, s = [10, 30]$

(d) n_C, SWDC, $W = 15, s = [10, 30]$ (e) s_{avg}, SWDC, $n_C = 20, W = 100$ (f) W, SWDC, $n_C = 10, s = [10, 30]$

(g) n_C, Hypercube, $W = 15, s = [10, 30]$ (h) s_{avg}, Hypercube, $n_C = 20, W = 100$ (i) W, Hypercube, $n_C = 10, s = [10, 30]$

Figure 8: Compute-aggregation costs for $\mathbf{aggr}(x, y) = \text{SetUnion}(x, y)$. Rows show topologies: (a-c) JellyFish; (d-f) SWDC Ring; (g-i) Hypercube; columns, X-axis: (a,d,g) no. of chunks; (b,e,h) avg. chunk size s_{avg} ($\Delta_s = 10$); (c,f,i) max edge weight.

phase, find the cheapest plan of data aggregation across the network, and on the second phase find an optimal scheduling of aggregations in time that would avoid hotspots and optimize desired objectives. The main contribution of this work is a universal taxonomy of aggregation functions that guides the choice of optimal plan for any particular application.

In our studies of the first phase, we have found that there is no universal solution that could be aplied to all aggregation functions, while choosing a suboptimal plan can increase aggregation costs by a factor of 10x and more; we have provided rigorous theoretical results that support this conclusion. Therefore, this leaves us with the second phase of the aggregation plan to explore; as the main direction for future work we see the development of a proactive approach of avoiding hotspots. In addition, we plan to identify universal characteristics of compute-aggregate tasks that would allow to unify the design principles of "ideal" aggregations.

References

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: Dynamic flow scheduling for data center networks, in: USENIX, 2010, pp. 281–296.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber, Bigtable: A distributed storage system for structured data (awarded best paper!), in: OSDI, 2006, pp. 205–218.

[3] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, Operating Systems Review 44 (2) (2010) 35–40.

[4] Y. Chen, R. Griffith, J. Liu, R. H. Katz, A. D. Joseph, Understanding TCP incast throughput collapse in datacenter networks, in: WREN, 2009, pp. 73–82.

[5] Y. Zhang, N. Ansari, On architecture design, congestion notification, TCP incast and power consumption in data centers, IEEE Communications Surveys and Tutorials 15 (1) (2013) 39–64.

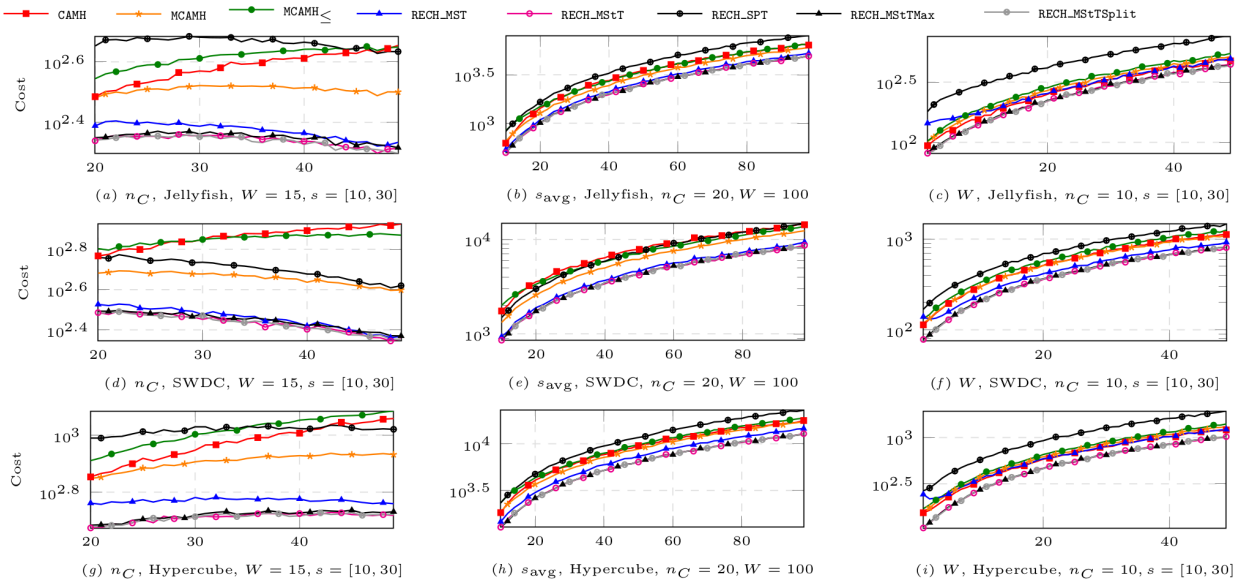


Figure 9: Compute-aggregation costs for $\mathbf{aggr}(x, y) = \text{ChooseBest}(x, y)$. Rows show topologies: (a-c) JellyFish; (d-f) SWDC Ring; (g-i) Hypercube; columns, X-axis: (a,d,g) no. of chunks; (b,e,h) avg. chunk size s_{avg} ($\Delta_s = 10$); (c,f,i) max edge weight.

[6] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[7] Y. Chen, A. Ganapathi, R. Griffith, R. H. Katz, The case for evaluating mapreduce performance using workload suites, in: MASCOTS, 2011, pp. 390–399.

[8] R. Rojas-Cessa, Y. Kaymak, Z. Dong, Schemes for fast transmission of flows in data center networks, IEEE Commun. Surv. Tutorials 17 (3) (2015) 1391–1422.

[9] P. Costa, A. Donnelly, A. I. T. Rowstron, G. O'Shea, Camdoop: Exploiting in-network aggregation for big data applications, in: NSDI, 2012, pp. 29–42.

[10] H. Yang, A. Dasdan, R. Hsiao, D. S. P. Jr., Map-reduce-merge: simplified relational data processing on large clusters, in: SIGMOD, 2007, pp. 1029–1040.

[11] Y. Yu, P. K. Gunda, M. Isard, Distributed aggregation for data-parallel computing: interfaces and implementations, in: SOSP, 2009, pp. 247–260.

[12] R. van Renesse, K. P. Birman, W. Vogels, Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining, ACM Trans. Comput. Syst. 21 (2) (2003) 164–206.

[13] N. Jain, M. Dahlin, Y. Zhang, D. Kit, P. Mahajan, P. Yalagandula, STAR: self-tuning aggregation for scalable monitoring, in: VLDB, 2007, pp. 962–973.

[14] P. Yalagandula, M. Dahlin, A scalable distributed information management system, CCR 34 (4) (2004) 379–390.

[15] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, J. Currey, Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language, in: OSDI, 2008, pp. 1–14.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: NSDI, 2012, pp. 15–28.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, M. Abadi, Naiad: a timely dataflow system, in: SIGOPS, 2013, pp. 439–455.

[18] S. Venkataraman, I. Roy, A. AuYoung, R. S. Schreiber, Using R for iterative and incremental processing, in: HotCloud, 2012.

[19] P. A. Tucker, D. Maier, T. Sheard, L. Fegaras, Exploiting punctuation semantics in continuous data streams, IEEE Trans. Knowl. Data Eng. 15 (3) (2003) 555–568.

[20] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: SIGMOD, 2010, pp. 135–146.

[21] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: Fault-tolerant stream processing at internet scale, PVLDB 6 (11) (2013) 1033–1044.

[22] W. Culhane, K. Kogan, C. Jayalath, P. Eugster, Optimal communication structures for big data aggregation, in: INFOCOM, 2015, pp. 1643–1651.

[23] P. Štefanič, M. Cigale, A. C. Jones, L. Knight, I. Taylor, C. Istrate, G. Suciu, A. Ulisses, V. Stankovski, S. Taherizadeh, G. F. Salado, S. Koulouzis, P. Martin, Z. Zhao, Switch workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications, Future Generation Computer Systems 99 (2019) 197 – 212. doi:https://doi.org/10.1016/j.future.2019.04.008.
URL http://www.sciencedirect.com/science/article/pii/S0167739X1831094X

[24] Z. Zhao, P. Martin, A. Jones, I. Taylor, V. Stankovski, G. F. Salado, G. Suciu, A. Ulisses, C. de Laat, Developing, provisioning and controlling time critical applications in cloud, in: Z. Á. Mann, V. Stolz (Eds.), Advances in Service-Oriented and Cloud Computing, Springer International Publishing, Cham, 2018, pp. 169–174.

[25] P. Stefanic, M. Cigale, A. C. Jones, L. Knight, D. Rogers, F. Quevedo Fernandez, I. Taylor, Application-infrastructure co-programming: managing the entire complex application lifecycle, in: CEUR Workshop Proceedings, Vol. 2357, CEUR Workshop Proceedings, 2019.

[26] D. Bruneo, T. Fritz, S. Keidar-Barner, P. Leitner, F. Longo, C. Marquezan, A. Metzger, K. Pohl, A. Puliafito, D. Raz, A. Roth, E. Salant, I. Segall, M. Villari, Y. Wolfsthal, C. Woods, Cloudwave: Where adaptive cloud management meets devops, in: 2014 IEEE Symposium on Computers and Communications (ISCC), Vol. Workshops, 2014, pp. 1–6.

[27] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, M. Wójcik, Re-architecting datacenter networks and stacks for low latency and high performance, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 29–42. doi:10.1145/3098822.3098825.
URL https://doi.org/10.1145/3098822.3098825

[28] S. Ghosh, A. Mukherjee, S. K. Ghosh, R. Buyya, Mobi-iost: Mobility-aware cloud-fog-edge-iot collaborative framework for time-critical applications, IEEE Transactions on Network Science and Engineering (2019) 1–1.

[29] A. N. Toosi, R. N. Calheiros, R. Buyya, Interconnected cloud computing environments: Challenges, taxonomy, and survey, ACM Comput. Surv. 47 (1). doi:10.1145/2593512.
URL https://doi.org/10.1145/2593512

[30] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.

[31] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65. doi:10.1145/2934664.
URL http://doi.acm.org/10.1145/2934664

[32] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, L. Zhou, Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs, in: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, ACM, New York, NY, USA, 2014, pp. 44–53. doi:10.1145/2591062.2591177.
URL http://doi.acm.org/10.1145/2591062.2591177

[33] T. White, Hadoop: The Definitive Guide, 1st Edition, O'Reilly Media, Inc., 2009.

[34] C. Kaklamanis, M. Chlebík, J. Chlebíková, Algorithmic aspects of global computing the steiner tree problem on graphs: Inapproximability results, Theoretical Computer Science 406 (3) (2008) 207 – 214.

[35] J. Byrka, F. Grandoni, T. Rothvoß, L. Sanità, An improved lp-based approximation for steiner tree, in: Proceedings of the Forty-second ACM Symposium on Theory of Computing, STOC '10, ACM, New York, NY, USA, 2010, pp. 583–592. doi:10.1145/1806689.1806769.
URL http://doi.acm.org/10.1145/1806689.1806769

[36] S. A. Jyothi, A. Singla, P. B. Godfrey, A. Kolla, Measuring and Understanding Throughput of Network Topologies, Tech. rep., http://arxiv.org/abs/1402.2531 (2014).

[37] A. Singla, C.-Y. Hong, L. Popa, P. B. Godfrey, Jellyfish: Networking Data Centers Randomly, in: USENIX, 2012.

[38] J.-Y. Shin, B. Wong, E. G. Sirer, Small-world datacenters, in: SOCC, SOCC '11, 2011, pp. 2:1–2:13.

[39] P. Chuprikov, S. Nikolenko, Formalizing compute-aggregate problems in cloud computing code, https://gitlab.com/pschuprikov/compute_aggregate.