Formalizing Compute-Aggregate Problems in Cloud Computing*

Pavel Chuprikov[1,2], Alex Davydow[1], Kirill Kogan[2], Sergey Nikolenko[1], and Alexander Sirotkin[1,3]

[1] Steklov Institute of Mathematics at St. Petersburg, Russia adavydow@gmail.com, sergey@logic.pdmi.ras.ru
[2] IMDEA Networks Institute, Madrid, Spain pavel.chuprikov@imdea.org, kirill.kogan@imdea.org
[3] National Research University Higher School of Economics, St. Petersburg, Russia avsirotkin@hse.ru

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 to optimize desired objectives: cost, latency, and others. 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, and support our results with an extensive experimental study.

Keywords: Compute-aggregate problems · Cloud computing · Competitive analysis.

1 Introduction

Data centers store data at different interconnected locations. Modern big data applications are highly distributed, and requests need to satisfy various objectives: latency, cost efficiency, etc. [2, 6, 14]. *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* [8, 27], 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

* This work was supported by the Russian Science Foundation grant 17-11-01276.

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: (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 in (1) 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 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 jobs is 40.3% of the initial data sizes in Google [11], 8.2% in Yahoo, and 5.4% in Facebook [7].

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. Section 2 summarizes prior art, Section 3 introduces the model. In Section 4 we classify aggregate functions with regard to their effect on input data chunks and study their computational properties, i.e., hardness and approximability of optimization problems. Section 5 presents our experimental results, and Section 6 concludes the paper.

2 Related work

Various frameworks split computations into multiple phases: Map-Reduce-Merge [23] extends MapReduce to implement aggregations, Camdoop [9] assumes that an aggregation's output size is a specific fraction of input sizes, Astrolabe [17] collects large-scale system state and provides on-the-fly attribute aggregation, and so on. Like other data-flow systems [11, 24, 25], Naiad [16] offers the low latency of stream processors together with the ability to perform iterative and incremental computations. The work [25] introduces a distributed memory ab-

straction for fault-tolerant in-memory computation on large clusters, with orders of magnitude better latency than disk accesses. Other stream processing frameworks support low-latency dataflow computations over a static dataflow graph [1, 15, 20], while [10] explores optimal tree overlays to optimize latency of compute-aggregate tasks under specified budget constraints.

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, 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 the 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 $c(e) \cdot \mathrm{size}(\boxed{x})$. 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

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 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

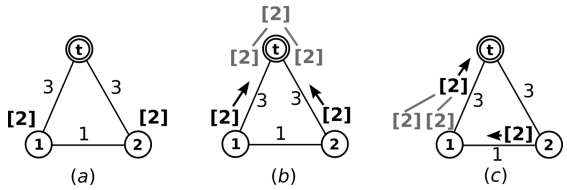

Fig. 1. A sample compute-aggregate task with three vertices t (target), u, and v: (a) the problem; (b) "move to root" plan with cost 12; (c) optimal aggregation plan with cost 11. Transmission cost of every edge is specified near the middle of the edge (e.g., $c(u, v) = 1$).

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 [5, 26] 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 [7, 11] 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 [22]. 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 [11]. 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 [21], 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 either $\mathbf{move}(\boxed{x}, v)$, which moves a chunk x to a vertex v, or $\mathbf{aggr}(\boxed{x}, \boxed{y})$, 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: $\mathrm{v}(\boxed{z}) = t$. E.g., Figs. 1b and 1c show aggregation plans for the problem on Fig. 1a. Aggregation plans are fully decoupled from the transport layer, producing instructions and constraints that the transport layer must satisfy.

An aggregation plan has an associated transmission cost $\mathrm{cost}(P)$, which is the sum of costs of all operations in P; here $\mathrm{cost}(\mathbf{aggr}(\boxed{x}, \boxed{y})) = 0$ (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.

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

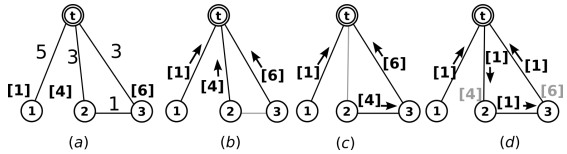

Fig. 2. Different μ lead to different plans: (a) a sample task; (b) optimal plan for $\mu(a,b) = a + b$; (c) optimal plan for $\mu(a,b) = \max(a,b)$; (d) optimal plan for $\mu(a,b) = \min(a,b)$.

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 $\text{size}(\boxed{xy}) = \mu(\text{size}(\boxed{x}), \text{size}(\boxed{y}))$. 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, μ should also have these properties. Some examples of μ for practical problems include: $\mu(a,b) = \text{const}$ 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; $\max(a,b) \le \mu(a,b) \le a + b$ for set union (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 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 both the aggregation size function is well-behaved and there are constraints on the graph structure, there is not much we could 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.

4 A Taxonomy of Aggregation Functions

There are different types of big data applications, a large variation in data-center network topologies, and countless data distributions, 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). E.g., if the network graph is a tree, it might be possible to construct an aggregation plan in linear time; or a better (in terms of the resulting cost) algorithm 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

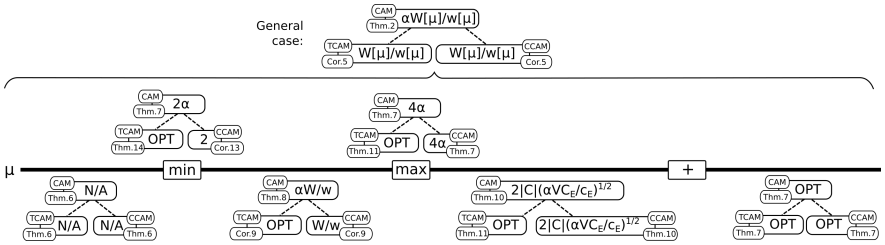

Fig. 3. The structure of our results in relation to aggregation size function μ and problem instance (CAM, TCAM, CCAM).

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). Our theoretical results are summarized in Fig. 3; the horizontal axis corresponds to how fast μ grows, and each small tree of results shows approximation ratios for CAM, TCAM, and CCAM, referring to specific theorems below.

4.1 General case

In this subsection we assume no constraint on the behavior of an aggregation size function μ. As an example, consider a simpler setting where all chunks have size x, and $\mu(x, x) = x$. In this case, when paths of two chunks intersect, it 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 [13], MSTT, that has many constant approximation algorithms. Using one of those we build our first aggregation plan construction algorithm `steiner_rec` (Alg. 2). If there is a polynomial α-approximate algorithm for MSTT, then `steiner_rec` 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, $w_C[\mu] = \min_{C' \subseteq C}\{\mu(C')\}$.

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**$(x, \mathrm{parent}(v))$)
    C.`update`(**move**$(x, \mathrm{parent}(v))$)
return P

Theorem 2. *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. □

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 known approximation ratio of $\ln 4 + \varepsilon \leq 1.39$ [4]. Although `steiner_rec` does not depend on either μ or chunk sizes, the approximation factor in Theorem 2 includes both. This result improves for special cases of TCAM and CCAM.

Theorem 3. *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 4. *If G is a tree,* 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 3 and Theorem 4 essentially say that in these special cases we have 1-approximation algorithms for MSTT. Theorem 2 and this observation together imply the following.

Corollary 1. *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 2 and Corollary 1 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 5. *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 ∞. 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$[\mu]$ implies a constant approximation of Set Cover which is impossible unless $P = NP$. For TCAM$[\mu]$, 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$[\mu]$ 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.

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. Theorem 5 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.

Theorem 6. *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 5), and for $\mu(x, y) \in [x + y, \infty)$ there is an optimal algorithm (Theorem 6). 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 starts 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 2), 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 7. *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 $\alpha \frac{W_C}{w_C}$.*

Corollary 2. *If $\min\{a, b\} \leq \mu(a, b) \leq \max\{a, b\}$ then there exist polynomial algorithms that solves* CCAM[μ] *and* TCAM[μ] *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`; below, with `SPT`.

Theorem 8. *If for all a and b $\max(a, b) \le \mu(a, b) \le a + b$, then there is an $2NV^{1/2}\sqrt{\alpha \frac{c_{\max}}{c_{\min}}}$-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, $c_{\min}$, 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 2, and for C_2 we run the algorithm from Theorem 6 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 $\delta = \frac{\sqrt{\alpha c_{\min}}}{V^{1/2}\sqrt{c_{\max}}}$ to minimize the total result, the total weight of both solutions will be $2NV^{1/2}\sqrt{\alpha \frac{c_{\max}}{c_{\min}}} W$. □

To improve the above theorem we cannot apply Theorem 3 to get rid of α for CCAM or TCAM since it uses 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. The algorithm is similar to Theorem 2.

Theorem 9. *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 2: 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. □

4.3 Specific aggregation size functions

Sometimes we can improve performance further if we know μ exactly. This is especially interesting for the "junction points" between previous results; Theorem 6 covers sum, here we concentrate on min and max.

Theorem 10. *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 13, 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. 4. □

Corollary 3. *There exists a polynomial 2-approximate algorithm for* CCAM[min].

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

Theorem 11. *There is 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 $\mathrm{dp}(v, \mathrm{size}(c)) = \sum_{u \in \mathtt{ch}(v)} \min\{2 \cdot \mathrm{size}(c) \cdot \mathrm{d}(v, u) + \mathrm{dp}(u, \mathrm{size}(c)), \mathrm{mc}(u) \cdot \mathrm{d}(v, u) + \mathrm{dp}(u, \mathrm{mc}(u))\}$. 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, for CAM[max], is also based on the MStT problem, but with a different construction.

Theorem 12. *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 $(M/2, M]$, the second in $(M/4, M/2]$, 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

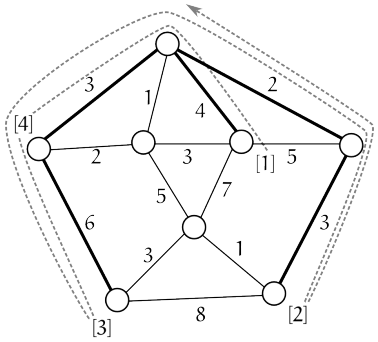

Fig. 4. Sample solution from Theorem 10. Steiner tree approximation is shown with bold lines, and the resulting path of the smallest chunk is shown by a dotted arrow. Note that this chunk never visits one edge more than twice.

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 $M/2^{i-1}$, and merging vertices does not increase the weight of a Steiner tree, the cost of the ith subset does not exceed $M/2^{i-1}\alpha\mathrm{ST}(i, i-1)$, where $\mathrm{ST}(i,j)$ is the optimal Steiner tree weight for chunks with sizes in $(M/2^i, M/2^j]$. Thus, the total cost does not exceed $2\alpha M \sum_{i=1}^{k} \mathrm{ST}(i, i-1)/2^i$. 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 $M \sum_{i=1}^{k} \mathrm{ST}(i,0)/2^i$. Some of the edges are counted more than once: an edge with the largest chunk of size $M/2^j$ moved along it has been counted once with a factor of $M/2^j$, once with $M/2^{j+1}$, and so on, but the real lower bound for this edge is $M/2^j$. Thus for every edge we have an extra factor of $1 + 1/2 + 1/4 + \ldots \leq 2$, and the optimal cost is at least $M/2 \sum_{i=1}^{k} \mathrm{ST}(i,0)/2^i$. Since $\mathrm{ST}(i,0) \geq \mathrm{ST}(i,i-1)$, the total approximation factor is $\frac{2\alpha M}{M/2} = 4\alpha$. □

Theorem 9 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.

Theorem 13. *If there exists $a > 0$ s.t. $\mu(a,a) = a$ 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 the 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 14. *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 [12]. We analyzed topologies that can serve for both server adjacency and switch adjacency, namely *JellyFish* [19], *Hypercube*, and *Small World Datacenter Ring* (SWDC Ring) [18]. 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 Alg. 2 that merges along the minimum *spanning* tree; `RECH_MStT` (Alg. 2); `RECH_SPT`, another variation of Alg. 2 that uses a tree of shortest paths; `RECH_MStTMax` (Thm. 12); `RECH_MStTSplit` (Thm. 8). An implementation of all algorithms and sample topologies can be found at [3]. Selected results are shown on Fig. 5; the plots show the number of data chunks $n_C = |C|$, maximal edge weight W, and range s of chunk sizes, $s = [s_{\text{avg}} - \Delta_s, s_{\text{avg}} + \Delta_s]$. To generate a CAM instance for a given topology, we uniformly select weights from 1 to W and uniformly place n_C data chunks in the vertices, each of size chosen uniformly from s. We used three aggregation functions: $\mu(x, y) = \exp(\log|x| + \log|y|)$ (Fig. 5a-c), which roughly estimates joining two dictionaries, WordCount, $\mathbf{aggr}(x, y) = \text{SetUnion}(x, y)$ (Fig. 5d-i), and $\mathbf{aggr}(x, y) = \text{ChooseBest}(x, y)$ (Fig. 5j-o), which compares the (random) priorities of two chunks. For each point, we averaged 1000 random experiments.

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

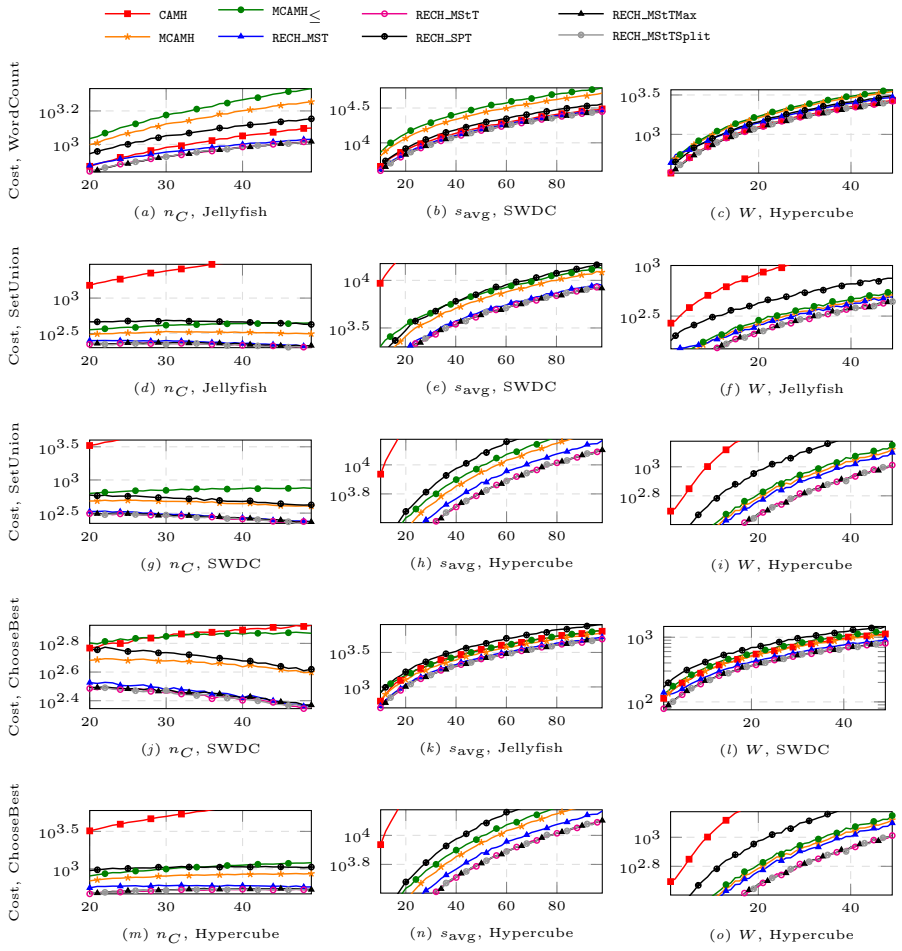

Fig. 5. Costs for different aggregation functions: (a-c) $\mu(x, y) = \exp(\log x + \log y)$; (d-i) $\mathbf{aggr}(x, y) = \text{SetUnion}(x, y)$; (j-o) $\mathbf{aggr}(x, y) = \text{ChooseBest}(x, y)$. Labels show X-axis value and topology. Parameters in left column: $W = 15$, $s = [10, 30]$; middle column: $n_C = 20$, $W = 100$; right column: $n_C = 10$, $s = [10, 30]$.

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 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, for a given topology heuristics designed for specific aggregation functions perform significantly better in their own domain. Second, heuristics designed for the same range of aggregation functions for a specific topology perform better than more general heuristics independent from network topologies. This confirms analytic results from Section 4 and allows designers of compute-aggregate infrastructures to extend the taxonomy proposed in Fig. 3 if better performance is needed. This demonstrates a fundamental tradeoff between simplicity of infrastructure and its performance.

6 Conclusion

In this work, we have introduced a model to find a schedule of aggregations that satisfies budget constraints rather than directly optimizing desired objectives such as latency or throughput. We believe that this approach will allow to decouple optimization problems from underlying transports and provide fine-grained control to exploit network infrastructure. Our primary contribution is a classification of aggregation functions with a theoretical and practical analysis that together lead to unified design principles of "perfect" aggregations.

References

1. Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: Fault-tolerant stream processing at internet scale. PVLDB **6**(11), 1033–1044 (2013)
2. Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., Vahdat, A.: Hedera: Dynamic flow scheduling for data center networks. In: USENIX. pp. 281–296 (2010)
3. Anonymous: Formalizing compute-aggregate problems in cloud computing code. https://github.com/CrKJdwbRAe/infocom2017
4. Byrka, J., Grandoni, F., Rothvoß, T., Sanità, L.: An improved lp-based approximation for steiner tree. In: Proceedings of the Forty-second ACM Symposium on Theory of Computing. pp. 583–592. STOC '10, ACM, New York, NY, USA

(2010). https://doi.org/10.1145/1806689.1806769, http://doi.acm.org/10.1145/1806689.1806769
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink^{TM}: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A distributed storage system for structured data (awarded best paper!). In: OSDI. pp. 205–218 (2006)
7. Chen, Y., Ganapathi, A., Griffith, R., Katz, R.H.: The case for evaluating mapreduce performance using workload suites. In: MASCOTS. pp. 390–399 (2011)
8. Chen, Y., Griffith, R., Liu, J., Katz, R.H., Joseph, A.D.: Understanding TCP incast throughput collapse in datacenter networks. In: WREN. pp. 73–82 (2009)
9. Costa, P., Donnelly, A., Rowstron, A.I.T., O'Shea, G.: Camdoop: Exploiting in-network aggregation for big data applications. In: NSDI. pp. 29–42 (2012)
10. Culhane, W., Kogan, K., Jayalath, C., Eugster, P.: Optimal communication structures for big data aggregation. In: INFOCOM. pp. 1643–1651 (2015)
11. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
12. Jyothi, S.A., Singla, A., Godfrey, P.B., Kolla, A.: Measuring and Understanding Throughput of Network Topologies. Tech. rep. (2014), http://arxiv.org/abs/1402.2531
13. Kaklamanis, C., Chlebk, M., Chlebkov, J.: Algorithmic aspects of global computing the steiner tree problem on graphs: Inapproximability results. Theoretical Computer Science **406**(3), 207 – 214 (2008)
14. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. Operating Systems Review **44**(2), 35–40 (2010)
15. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD. pp. 135–146 (2010)
16. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: SIGOPS. pp. 439–455 (2013)
17. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans. Comput. Syst. **21**(2), 164–206 (2003)
18. Shin, J.Y., Wong, B., Sirer, E.G.: Small-world datacenters. In: SOCC. pp. 2:1–2:13. SOCC '11 (2011)
19. Singla, A., Hong, C.Y., Popa, L., Godfrey, P.B.: Jellyfish: Networking Data Centers Randomly. In: USENIX (2012)
20. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. Knowl. Data Eng. **15**(3), 555–568 (2003)
21. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Inc., 1st edn. (2009)
22. Xiao, T., Zhang, J., Zhou, H., Guo, Z., McDirmid, S., Lin, W., Chen, W., Zhou, L.: 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. pp. 44–53. ICSE Companion 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2591062.2591177, http://doi.acm.org/10.1145/2591062.2591177
23. Yang, H., Dasdan, A., Hsiao, R., Jr., D.S.P.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD. pp. 1029–1040 (2007)

24. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI. pp. 1–14 (2008)
25. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI. pp. 15–28 (2012)
26. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: A unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664, `http://doi.acm.org/10.1145/2934664`
27. Zhang, Y., Ansari, N.: On architecture design, congestion notification, TCP incast and power consumption in data centers. IEEE Communications Surveys and Tutorials **15**(1), 39–64 (2013)