Planning in Compute-Aggregate Problems as Optimization Problems on Graphs

Pavel Chuprikov*†, Alex Davydow*, Kirill Kogan†, Sergey I. Nikolenko*, Alexander V. Sirotkin‡*
*Steklov Institute of Mathematics at St. Petersburg, St. Petersburg, Russia
†IMDEA Networks Institute, Madrid, Spain
‡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. We present a formalization of compute-aggregate planning parameterized by the aggregation function.**

Various frameworks split computations into multiple phases: Map-Reduce-Merge [1] extends MapReduce to implement aggregations, Camdoop [2] assumes that an aggregation's output size is a specific fraction of input sizes, Astrolabe [3] collects large-scale system state and provides on-the-fly attribute aggregation, and so on. Other stream processing frameworks support low-latency dataflow computations over a static dataflow graph [4], while [5] explores optimal tree overlays to optimize latency of compute-aggregate tasks. Applications usually have little control over how network transport handles the data, but more fine-grained control may be required in order to avoid, e.g., the *incast problem* [6]. Suppose also that each compute-aggregate task should conform to a budget constraint. The problem thus divides into two: (1) find a "cheapest" plan given network parameters and (2) redistribute aggregations computed in (1) while optimizing desired objectives. In this setting, we can solve the first phase independently of the underlying transport protocols, while the second phase can address such problems as incast. The first phase is also of separate interest since it can represent various economic settings (e.g., energy efficiency) during aggregation and lead to better utilization of network infrastructure. Our primary goal is to identify universal properties of compute-aggregate tasks, leading to unified design principles. We define a model that needs to know only one property: the (approximate) size of two data chunks after aggregation.

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. 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 $C = \{\boxed{x_0}, \boxed{x_1}, \ldots, \boxed{x_k}\}$, each $\boxed{x_i}$ 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 cost function $c : E \to \mathbb{R}_+$ on G associates with each link e 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 three-node example of a compute-aggregate task is presented on Fig. 1a. Costs are shown on the edges, square brackets denote chunks, and the root vertex is marked by t.

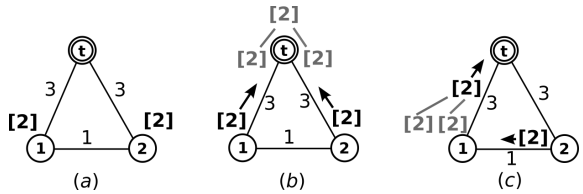

Fig. 1: A sample compute-aggregate task with three vertices t (target), u, and v: (a) the problem; (b) "move to root" plan with cost 12; (c) optimal aggregation plan with cost 8.

The simplest form of an aggregation plan is "move to root": bring everything to the root node t. It can be suboptimal: assume that on Fig. 1 the aggregation function chooses the best chunk, so the aggregated size does not exceed the maximal size of initial chunks. Then "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 the chunks merge, and the resulting chunk of size 2 moves to t with total cost 8. Besides, "move to root" can overload the links near the root. This leads to the idea of intermediate aggregations: aggregate data chunks en route to t. Recent studies [7], [8] 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 the initial data chunks; thus, reducing several chunks to one can significantly reduce storage requirements.

We introduce the function $\mathbf{aggr}$ defined on data chunks, assuming it to be *associative*: $\mathbf{aggr}(\boxed{x}, \mathbf{aggr}(\boxed{y}, \boxed{z})) = \mathbf{aggr}(\mathbf{aggr}(\boxed{x}, \boxed{y}), \boxed{z})$, and *commutative*: $\mathbf{aggr}(\boxed{x}, \boxed{y}) = \mathbf{aggr}(\boxed{y}, \boxed{x})$. The basic principle of *data locality optimization*, which lies at the heart of the *Hadoop* framework [9], 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. An aggregation plan is a sequence of operations $P = (o_0, o_1, \ldots, o_m)$, where each o_i is either

https://doi.org/10.1109/ICNP.2017.8117579

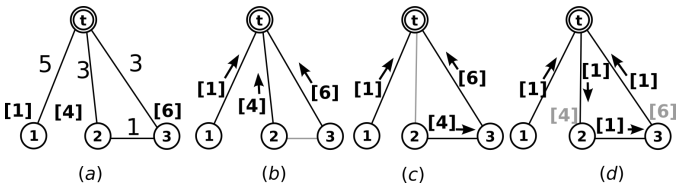

Fig. 2: Different μ lead to different plans: (a) a sample task; (b) optimal plan for $\mu(a,b) = a + b$; (c) optimal plan for $\mu(a,b) = \max(a,b)$; (d) optimal plan for $\mu(a,b) = \min(a,b)$.

$\mathbf{move}(\boxed{x}, v)$, move a chunk x to a vertex v, or $\mathbf{aggr}(\boxed{x}, \boxed{y})$, merge 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 important advantages over "move to root": the TCP-incast problem becomes less pronounced because inbound traffic is spread among different nodes, the total number of transmitted bits is reduced due to earlier aggregations, storage capacity is now less of a constraint since less data has to be collected per node, and data transmission cost is also reduced (cf. examples on Fig. 1). In order to formally define the optimization problem for aggregation, however, we have to know, given $\boxed{x}$ and $\boxed{y}$, the size of their aggregation result $\boxed{xy}$. 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 $\mathbf{aggr}$ is assumed to be associative and commutative, every aggregation size function 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;
- $\max(a,b) \le \mu(a,b) \le a + b$ 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 along edge $2 \to 3$ and merge, then move the resulting chunk of size 6 to the root (Fig. 2c). Finally, for $\mu(a,b) = \min(a,b)$ it is best to leave large chunks in place and traverse the graph with the smallest chunk (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* $\mathrm{cost}(P)$ *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. ■

We have introduced a model to find a schedule of aggregations that satisfies constraints rather than directly optimizes desired objectives, formulated the basic optimization problem and pinpointed how it depends on the aggregation function μ. We believe that this approach will allow to decouple optimization problems from underlying transports and provide fine-grained control to exploit network infrastructure.

Acknowledgments: This work was supported by the Russian Science Foundation grant no. 17-11-01276, "Networking and distributed systems and algorithms and related fundamental problems".

REFERENCES

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

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

[3] R. van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Trans. Comp. Syst.*, vol. 21, no. 2, pp. 164–206, 2003.

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

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

[6] Y. Zhang and N. Ansari, "On architecture design, congestion notification, TCP incast and power consumption in data centers," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 1, pp. 39–64, 2013.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[8] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, "The case for evaluating mapreduce performance using workload suites," in *MASCOTS*, 2011, pp. 390–399.
[9] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.