ROME: All Overlays Lead to Aggregation, but Some Are Faster than Others

MARCEL BLÖCHER, TU Darmstadt, Germany
EMILIO COPPA, Sapienza University of Rome, Italy
PASCAL KLEBER, TU Darmstadt, Germany
PATRICK EUGSTER, Università della Svizzera italiana (USI), Switzerland and Purdue University, USA
WILLIAM CULHANE, Imperial College London, UK
MASOUD SAEIDA ARDEKANI, Purdue University, USA

Aggregation is common in data analytics and crucial to distilling information from large datasets, but current data analytics frameworks do not fully exploit the potential for optimization in such phases. The lack of optimization is particularly notable in current "online" approaches that store data in main memory across nodes, shifting the bottleneck away from disk I/O toward network and compute resources, thus increasing the relative performance impact of distributed aggregation phases.

We present ROME, an aggregation system for use within data analytics frameworks or in isolation. ROME uses a set of novel heuristics based primarily on basic knowledge of aggregation functions combined with deployment constraints to efficiently aggregate results from computations performed on individual data subsets across nodes (e.g., merging sorted lists resulting from top-k). The user can either provide minimal information that allows our heuristics to be applied directly, or ROME can autodetect the relevant information at little cost. We integrated ROME as a subsystem into the Spark and Flink data analytics frameworks. We use real-world data to experimentally demonstrate speedups up to 3× over single-level aggregation overlays, up to 21% over other multi-level overlays, and 50% for iterative algorithms like gradient descent at 100 iterations.

Marcel Blöcher work done at TU Darmstadt. Now at SAP.
Pascal Kleber work done at TU Darmstadt. Now at Deutsche Bundesbank.
William Culhane work done at Imperial College London. Now at Google.
Masoud Saeida Ardekani work done at Purdue University. Now at Google.

This article integrates and extends previous work, notably an initial description of the idea of leveraging aggregation function characteristics for optimizing corresponding phases [17] and an exploration of theoretical foundations for such optimized aggregation [18]. The present article presents a detailed design of a corresponding aggregation system, called "ROME." Beyond integrating relevant portions of our prior works [17, 18], we describe in this article ROME's architecture, APIs, as well as novel heuristics for real-life deployment, and evaluate its performance while integrated into well-known frameworks Apache Spark [4] and Apache Flink [3]. This work has been co-funded by the German Research Foundation (DFG) as part of the projects B2 in the Collaborative Research Center (CRC) 1053 "MAKI," US NSF grants 1618923 "Elastic and Robust Cloud Programming" and 1421910 "Practical Assured Big Data Analysis in the Cloud," and ERC grant FP7-617805 "LiveSoft."

Authors' addresses: M. Blöcher and P. Kleber, TU Darmstadt, Hochschulstraße 10, 64289 Darmstadt, Germany; emails: {bloecher.marcel, pascal.kleber.pk}@gmail.com; E. Coppa, Sapienza University of Rome, Via Ariosto 25, Rome, Italy; email: coppa@diag.uniroma1.it; P. Eugster, Università della Svizzera italiana, Via la Santa 1, 6962 Lugano-Viganello, Switzerland and Purdue University, 305 N. University Street, West Lafayette, IN 47907, United States; email: eugstp@usi.ch; W. Culhane, Imperial College London, South Kensington Campus, London SW7 2AZ, UK; email: wjculhaneiv@gmail.com; M. S. Ardekani, Purdue University, 305 N. University Street, West Lafayette, IN 47907, United States; email: m@masoud.io.

0734-2071/2022/07-ART4 $15.00
https://doi.org/10.1145/3516430

CCS Concepts: • **General and reference** → **Performance**; • **Theory of computation** → **Distributed algorithms**; • **Software and its engineering** → **Cloud computing;**

Additional Key Words and Phrases: Big data aggregation overlay, distributed algorithms

ACM Reference format:
Marcel Blöcher, Emilio Coppa, Pascal Kleber, Patrick Eugster, William Culhane, and Masoud Saeida Ardekani. 2022. ROME: All Overlays Lead to Aggregation, but Some Are Faster than Others. *ACM Trans. Comput. Syst.* 39, 1–4, Article 4 (July 2022), 33 pages.
https://doi.org/10.1145/3516430

1 INTRODUCTION

Data analytics is a core challenge of our era. Extracting and distilling information from large datasets is now commonplace in academia (e.g., astrophysics, chemistry), enterprise (e.g., advertising, recommendation), and government (e.g., intelligence services, demographic surveys) [5, 15, 42].

1.1 Aggregation

Data analytics systems typically split data across nodes and compute information from subsets of the data, then aggregate the partial results.

Many aggregation functions are associative [59], so a natural choice is to aggregate results along an *overlay (network)* such as a tree connecting leaf nodes (where original computations occur) to a root (where the final result will be available). It becomes clear that there are simple customizations to such aggregation trees created for a broad range of aggregation functions. Exactly which customizations are applicable—most prominently affecting *fan-in* of the tree—depends on characteristics of the aggregation function that affect the size of the data.

1.2 Optimizing Aggregation

The instinctual decision to maximize parallelism via bushy trees with small fan-ins [7] where many nodes at the lowest level run simultaneously is suboptimal, say, for sorting, because every element is seen at each level, leading to a high amount of repeated work. Conversely, a system with a fixed single coordinator performing aggregation (e.g., as in Cao and Wang's work [8]) has non-optimal latency for applications such as deduplication where data filtering removes burdensome repetition.

1.3 In Perspective

Intuitively, the problem of aggregation can be considered the "inverse" of *multicast*: In the latter case, data is typically propagated along a tree, and the function applied at every node in the tree, in the simplest case, copies in the input to all outgoing links; in more complex scenarios, nodes can perform transformation functions on data prior to forwarding it (e.g., for interoperability), or collect acknowledgments or negative acknowledgments (for reliability).

While much effort has been invested in optimizing multicast *spanning trees*, aggregation has received less attention, although aggregation is very common in practice, and its impact is increasing in the era of data analytics. While the familiar problems—e.g., merging sorted input or combining of word counts—have outputs at least as large as each input, the average MapReduce job at Google [19] and the common "aggregate" jobs at Facebook and Yahoo! [10] actually decrease the amount of data. Despite the large potential performance impact of simple traits of aggregation trees, most systems described in literature performing aggregation are agnostic to them.

1.4 Leveraging Aggregation Function Characteristics

We propose to automatically tailor aggregation overlays to *specific problems*, using the ratio R of the output size of the aggregation function at hand to its input size, which is easy to find or estimate

Table 1. Some Common Aggregation Functions and Their Size Ratios of Output to *One* Input ($|a|$ Means Size of a)

Common problems	$R = \frac{\lvert\text{Output}\rvert}{\lvert\text{Input}\rvert}$
Production jobs at Facebook, Yahoo! [10], Google [19]; sieve and deduplication algorithms	<1
Top-k on pre-partitioned data, k-means clustering, square matrix multiplication, word count with fixed dictionary, gradient descent with mini-batching	=1
Top-k on arbitrary data, word count on mismatched dictionaries, sort, generate n-grams	>1

for many aggregation functions. For example, a *top-k aggregate* (top-k for short) overlay reports the k groups according to a selection criteria on the group's score [25]. If all data belonging to a group resides on a single partition stored at a single leaf node such that the leaf nodes can calculate scores for each group, then the aggregation function produces output of size k—a ratio of $R = 1$ (i.e., k/k). If some group's data is spread across all partitions, however, then the aggregators must merge sets for the scores to be calculated at the root, resulting in output of each aggregator that is bigger than its input. Table 1 shows common aggregation problems grouped by relevant ratios.

To better illustrate top-k on pre-partitioned data and top-k on arbitrary data, and the difference between them, consider the following example with two worker nodes w_1, w_2, and $k = 2$: With pre-partitioned data, assume w_1 gets $\{(a,2),(b,3),(c,5)\}$. It suffices for w_1 to report its own top-2 $\{(b,3),(c,5)\}$, as w_2 will not report anything for any same keys as w_1. The final top-2 can thus contain *at most* w_1's top-2 b and c. Hence, no worker needs to report more than k in the general case, from which k are selected at every step of the aggregation so $R = 1$. However, in the case of top-k with arbitrary data, w_2 might actually have $\{(a,3),(d,2),(e,4)\}$, and so if w_2 reports only its top-2 as $\{(a,3),(e,4)\}$ taking the combined top-2 from both workers' top-2 will yield $\{(c,5),(e,4)\}$ although a's combined count is $2 + 3 = 5$, which is more than that of e, and so the result really should be $\{(c,5),(a,5)\}$. Hence, workers in the general case must report all their values, and all aggregated values passed on at every level, clearly leading to $R > 1$.

Research that underscores the importance of minimal aggregation latency largely ignores the impact of overlays [28, 32]; existing big data analytics frameworks mostly use one-size-fits-all overlays, leading to unnecessarily high latency for distributed data aggregation. When customizable overlays are available in a framework, users must *manually* configure them properly for a particular problem. Such static optimization might become simply impossible with non-trivial aggregation functions (e.g., composed functions) and jobs, or when the same code/code portion is executed on several distinct datasets with different data skew.

1.5 Solution Outline

In this article, we present a novel holistic approach for optimizing aggregation implemented in a system called ROME (*Robust Aggregation Overlays minimizing Execution Time*). In short, our approach is threefold:

(1) an *Analysis* stage obtains relevant constraints from applications, based on which,
(2) an *Overlay* stage determines theoretically (near-)optimal overlay trees for idealized settings; finally,
(3) a *Mapping* stage applies several heuristics to tailor these overlays to real-life (non-ideal) deployment constraints at hand.

Like most currently popular systems for data analytics [2, 53, 60], ROME works in-memory to significantly reduce latency and unpredictable timings. It is designed for easy integration into existing data analytics systems.

1.6 Contributions

The contributions of this article are as follows:

- We define a model for a class of problems termed *compute-aggregate* whose distributed execution can be optimized by manipulation of an underlying aggregation tree without requiring revision to applications themselves.
- We elucidate why current data analytics frameworks, while well-suited to compute-aggregate, are currently addressing these problems inefficiently by design, and we introduce a framework to rectify that.
- We identify parameters for aggregation trees and prove their optimal values via mathematical models for various cases.
- We adapt these proven heuristics for creating near-optimal aggregation overlays by simplifying them to only use the size of data output by an aggregation function divided by the size of its input—a value that can be determined on-the-fly. We also develop a load-balancing technique and other practical alterations to reduce network usage during aggregation, as well as reaggregation (e.g., for incremental computations).
- We introduce ROME, a full-featured system implementing our heuristics. We describe its architecture and API. We then integrate ROME as a subsystem into two common data analytics frameworks, Apache Flink [3] and Apache Spark [4], to increase the efficiency of end-to-end data analytics.
- We evaluate ROME through these two systems. We empirically demonstrate that our chosen overlays are uniformly better than any predetermined overlay. Our microbenchmarks show differences of up to 86% between targeted and naïve overlays, even with no memory constraints and smaller deployments than typical in real-world use. Both of these factors could result in bigger differences in practice. Within real-world systems, job execution time shrinks by a factor of up to 3 over systems using single-level aggregation overlays such as `reduce` in Flink and Spark, by 21% over the `treeReduce` in Spark, and 16% if using Spark's feature to *manually* configure an overlay. When running an iterative algorithm like gradient descent, total runtime improves by 50% at 100 iterations over Spark's Mllib implementation, which also uses a multi-level aggregation overlay. We also show that using optimized overlays can reduce the memory requirements over those used to complete jobs in Flink or Spark deployments.

1.7 Roadmap

The rest of the article is structured as follows: Section 2 defines the model considered and how it relates to the current landscape of data analytics. Section 3 presents an overview of our approach. Section 4 provides optimality proofs for fan-in values F for specific values of R and scenarios. Section 5 presents the implementation of our ROME system, including its architecture, API, handling of faults, as well as how it integrates into Flink and Spark. Section 6 presents experiments with both of these prototypes. We position our work to prior art in Section 7 and conclude with Section 8.

2 MODEL

This section outlines the compute-aggregate family of problems considered and how it fits into the landscape of data analytics.

2.1 Problem Definition

Intuitively, compute-aggregate problems consist of two phases (see Figure 1): a *(i)* compute phase processes distributed subsets of input in parallel; a subsequent *(ii)* aggregation phase combines the results of the first phase to obtain a final output.

More formally, using the notation summarized in Table 2:

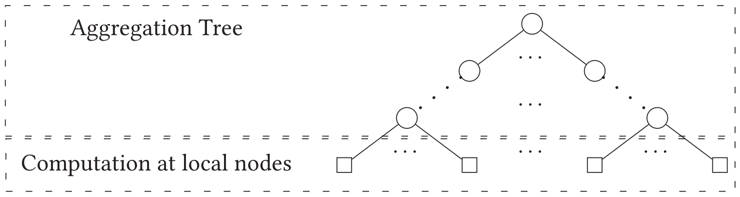


Fig. 1. Visual representation of the computation and aggregation phases.

Table 2. Notation Used

Token	**Meaning**
n	Number of computation/leaf nodes.
$f()$	Initial computation function.
$g()$	Aggregation function.
$h()$	(Composed) function to compute and aggregate.
z	Data formatted for input/computation.
x	Data formatted for aggregation.
F	Fan-in of the tree, making the height $O(\log_F n)$.
$g(\overline{x})$	The aggregation function for a set of inputs $\overline{x}$.
$g^t(\overline{x})$	Returns the time taken for $g(\overline{x})$ (with communication).
t	Time per unit of data for linear $g^t(\overline{x})$; $g^t(\emptyset) = 0$.
x_0	Output from one computation node.
R	Ratio of the final aggregate output size to $\lvert x_0 \rvert$.
R_1	Ratio of output sizes of individual levels.

Definition 1. A compute-aggregate task produces output $h(\overline{z})$ from input $\overline{z} = z_1, \ldots, z_n$ where $h(\overline{z})$ is decomposable into computations on partial inputs, $f(z_1), \ldots, f(z_n)$, and an aggregation function $g()$ such that $h(\overline{z}) \equiv g(x_1, \ldots, x_n)$ with $\forall i \in [1..n]\ x_i = f(z_i)$.

Intuitively, each computation node contains some subset of the intial data. After computation (i), a system aggregates (ii) the results along an *aggregation tree* (henceforth simply tree) communication structure to create the final output. With the exception of passing the results of the computation to the aggregation tree the two phases are independent from each other. The two phases are visually represented in Figure 1.

We consider optimizing the aggregation phase. Optimizing computation requires knowledge about the data, data structures, and computation for each specific problem. We show optimizing the tree often only requires knowing very basic information about the aggregation function.

Aggregation can be triggered by the completion of the computation phase or run periodically on the current state of the data, as long as the data is formatted for aggregation. Aggregation applies some function g to all of the outputs of the computation nodes, $g(f(z_1), \ldots, f(z_n))$. This does not have to be done in a single step. Aggregation can be applied to the results of previous aggregation. When aggregation begins, each output from a leaf is sent to a single aggregation node. The aggregation is applied to all inputs received at the node, and the node outputs the aggregated result. The outputs from those nodes, if there are indeed multiple such nodes, are in turn aggregated. The final aggregate result contains exactly one path to each leaf, so each computation output is included exactly once, resulting in an explicit tree structure. Figure 2 shows how 16 leaf nodes can be placed in four different trees that only differ in their fan-ins.

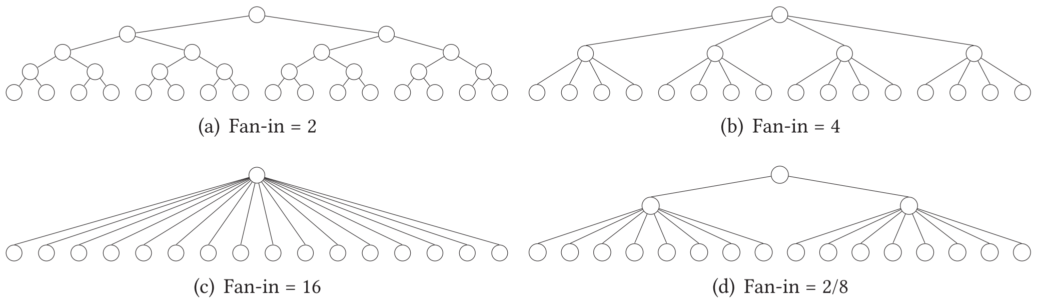


Fig. 2. Four aggregation trees with 16 leaves.

2.2 Function Requirements

We consider aggregation functions that take $\overline{x} = x_1 \ldots x_m$ and output an aggregate $x^{1..m}$, i.e., $x^{1..m} = g(\overline{x})$. Functions must be able to handle any number of inputs for the increased fan-in to be effective, as there is no advantage to having multiple inputs available if they cannot be used.

Functions are cumulative, commutative, and associative. This essentially means inputs may be aggregated in any order with any group of inputs, including those that are outputs of non-root nodes of the tree. Definitions 2, 3, and 4 capture the properties more precisely. Definitions require equivalency (≡), not necessarily identical output. For example, if a system is supposed to output the single word with the maximum number of occurrences (word count) and two words are tied for that distinction, either word may be returned.

Definition 2 (Cumulative Aggregation). $g(g(\overline{x}), g(\overline{x}')) \equiv g(\overline{x}, \overline{x}')$

Definition 3 (Commutative Aggregation). $g(\overline{x}', \overline{x}) \equiv g(\overline{x}, \overline{x}')$

Definition 4 (Associative Aggregation). $g(g(\overline{x}, \overline{x}'), \overline{x}'') \equiv g(\overline{x}, g(\overline{x}', \overline{x}'')) \equiv g(\overline{x}, \overline{x}', \overline{x}'')$

2.3 In Perspective

Before detailing how we optimize aggregation, we put it in the perspective of state-of-the-art data analytics systems, which can natively aggregate data (see Table 3).

Aggregation in many big data analytics frameworks follows the MapReduce [19] approach, and map data to disjoint processing partitions. This *aggregation by partition* model still inspires a significant portion of data analytics. It works well for problems like word count where aggregation can be partitioned. We show this is inefficient for problems such as top-k in Section 5.

For *total aggregation* jobs where all data must be compared (e.g., top-k, sort), at least transitively, to each other to find a global result, aggregation by partition must use only a single partition. That is, users must either create a single reducer (fan-in of n) or run iterations of the problem to prune data at the cost of remapping at each iteration. While Flink [3] addresses some issues with MapReduce (e.g., processing in-memory for lower latency), it still suffers from similar aggregation limitations. Aggregation by partition and total aggregation are disjoint in the big data problem space, as shown in Figure 3, even if the tools for one may be applied to the other.

Spark [4] also uses an in-memory model. Additionally, it adds an extra *aggregator* functionality to the MapReduce model, allowing aggregation to be distributed across multiple reducers. These aggregators require "add-only" semantics, which require monotonic operators without supporting transitive operators (i.e., enforcing non-transitive operators). This is insufficient for a wide range

Table 3. System Comparison

System	Fan-in	In-memory	Aggregation types
MapReduce [19]	Manually configurable		By partition
Flink [3]	Manually configurable	✓	By partition
Spark [4]	Manually configurable *or* tree template by height	✓	By partition *or* total aggregation *or* add-only semantics
ROME	Adaptive *or* self-adaptive	✓	Total aggregation

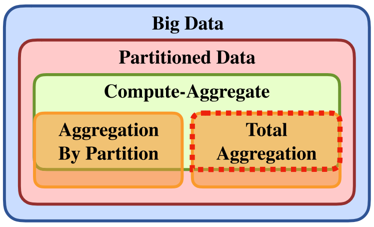


Fig. 3. Aggregation by partition and total aggregation in the data analytics problem space.

of problems like top-k sorting, which rely on score comparisons *across* nodes to be transitive. Consequently, for aggregation methods more complex than a monotonically increasing counter, Spark must use its `reduce` operator, effectively limiting the deployment to a fan-in of n.

The `treeReduce` functionality (added in 2015) in Spark enables user-specified aggregation topologies, however with several drawbacks. First, the topology in `treeReduce` is defined by *height* rather than *fan-in* using a *scale* factor internally

$$scale := \max(2, \lceil partitions^{1/height} \rceil).$$

When a user calls `treeReduce`, Spark runs partial aggregation rounds as long as

$$remaining\ partitions > scale + \lceil remaining\ partitions/scale \rceil$$

evaluates positively, and updates

$$remaining\ partitions := remaining\ partitions/scale$$

accordingly. As a consequence, the user sets only an upper bound of the total height of the aggregation tree. Hence, Spark is forced to fit the aggregation overlay to the number of partitions, workers, and given (max) height, which ignores the actual aggregation function.

Second, Spark does not provide guidance on how to pick an appropriate value. Simply setting height, i.e., the upper bound, to a very large value will cause Spark to build up aggregation trees of maximum height, always enforcing a fan-in of 2. Informed users may use our heuristics presented shortly based on the ratio R of final output to one input, and the number n of nodes across which data is originally distributed, to manually determine a height. However, one must recalculate the height when the application is deployed with a different number of worker nodes or workload. Even then, resulting overlays may be skewed by the greedy hash partitioner Spark uses to form an aggregation tree based on a rough fan-in derived from the provided height.

Clearly, manual calculation of an aggregation overlay requires that R and n are known before the application is deployed, which can be hard. Thus, the ability of ROME to autodetect R, as we

Table 4. Overview of Optimizations and Heuristics Devised by ROME

Heuristic	**Section**	**Input**	**Output**	**Description**
Autodetection of R	Section 3.2	n: # of nodes, $R[]$: ratios	$\mathcal{G}$: overlay	Actual ratios $R[1..n]$ are measured on nodes of first level ($F = 2$) to build overlay $\mathcal{G}$ for upper levels.
Optimal fan-in	Section 3.3.1	R: ratio	F: fan-in	Optimal fan-in F is computed using R based on Table 5.
Fan-in rounding	Section 3.3.2	F: fan-in	F': fan-in	Fan-in F is rounded up ($F' = \lceil F \rceil$) to ensure a discrete value.
Balancing mechanism	Section 3.4.1	$\mathcal{G}$: overlay	$\mathcal{G}'$: overlay	$\mathcal{G}'$ is constructed from $\mathcal{G}$ to minimize performance skew across nodes.
Node colocation	Section 3.4.2	$\mathcal{G}$: overlay	$\mathcal{G}'$: overlay	Nodes at different levels are colocated to reduce communication.
Root node bypass	Section 3.4.3	$\mathcal{G}$: overlay	$\mathcal{G}'$: overlay	$\mathcal{G}'$ replaces root node of $\mathcal{G}$ by worker of the third-party system (e.g., Spark).
Append-only updates	Section 3.4.4	$\mathcal{G}$: overlay, V: node	$\mathcal{G}'$: overlay, V': node	New node V with append-only data is added to overlay $\mathcal{G}$ and aggregation recomputed at root V'.
Infrequent updates	Section 3.4.4	$\mathcal{G}$: overlay, V: node	$V[]$: nodes	Values are re-computed at nodes $V[]$ ($V[1] = V$).

R, F, $\mathcal{G}$, and V represent characteristic ratios, fan-ins, overlay graphs, and nodes, respectively. Note that for presentation simplicity, input and output types are conceptual, i.e., actual heuristics might operate at other levels of abstraction inside the implementation.

shall detail in Section 3.2, and use a properly determined topology based on fan-in at execution time is a more applicable way and leads to lower job latency.

3 OPTIMIZING COMPUTE-AGGREGATE

While the aggregation function $g()$ is fixed for a given problem, the overlay over which to apply it is configurable. Our goal is thus to find an aggregation overlay yielding minimal latency, using the set of available resource to its best. This section discusses our threefold approach towards that goal. To help the reader quickly identify our optimization techniques proposed throughout the section, we summarize them in Table 4.

3.1 Optimizing Overlays

To optimize aggregation overlays, ROME applies a three-stage approach as shown in Figure 4 to an aggregation job submitted by an application. In short:

(1) The *Analysis* stage obtains relevant constraints for the job. This consists first and foremost in the mentioned characteristic R-ratio *between the aggregation function output and a single input* (cf. Table 2).
(2) An *Overlay* stage uses R to determine the *fan-in* of a (nearly) optimal overlay tree for an idealized setting.
(3) A *Mapping* stage applies several heuristics to tailor such an overlay to the real-life deployment constraints at hand; this includes catering for workload rebalancing, trees with non-fractional height, limited resources of on-path aggregation nodes, and strategic reuse of resources.

The job is then executed by ROME. In case the job was submitted through a data analytics framework that ROME is integrated into, this execution happens in a concerted manner with the framework, as we shall elaborate on shortly.

Next, we discuss the three phases in more detail.

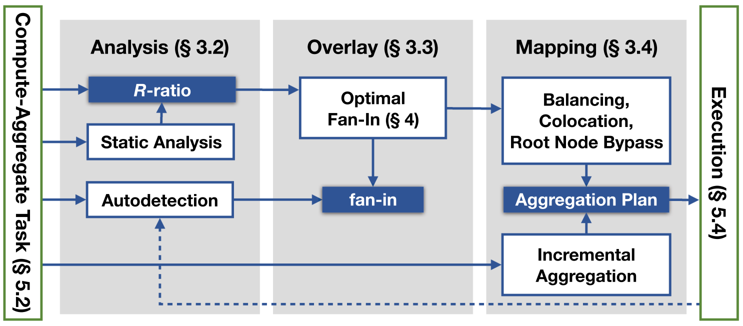


Fig. 4. ROME's three-staged approach.

3.2 Analysis Stage

The first stage is concerned with obtaining characteristic information from jobs that allow the necessary aggregation to be optimized. Chiefly, this consists in the R-ratio. Other parameters used in later stages such as for incremental aggregation (cf. Figure 4) are (currently) obtained explicitly as parameters of the job.

Static analysis. R-ratios are well-known for a number of aggregation scenarios such as the ones presented in Table 1. ROME can thus perform a simple static analysis of aggregation jobs passed to it to determine if they consist in/use any of these functions directly. Depending on the integration of ROME into a larger framework, that framework can share a map with ROME that outlines any of the framework's own known/pre-defined aggregation functions and their corresponding R-values (or correspondences to ROME's pre-defined ones). Simple embeddings of these functions inside composite functions, e.g., inside loops, allow for automatic inference of R in a good number of cases.

Autodetecting R. The static inference of R from a complex aggregation function $g()$ is not always possible. For the very same reason, one cannot automatically extract or synthesize an aggregation function $g()$ from a function $f()$ naïvely applied to an entire dataset, to allow for automatic breakdown of $h()$ into compute and aggregation functions $f()$ and $g()$ and corresponding distributed multi-phase execution (cf. References [34, 39]).

For these cases, ROME supports automatic runtime detection of R. ROME then builds the first level of aggregation with a fan-in of 2. Once nodes at this level complete their local aggregation, R is locally computed and sent to the controller, which builds the heuristic tree for the average computed value of R and the $\frac{n}{2}$ results already computed. We use a fan-in of 2 for initiation, as it allows estimating R very quickly while also preserving as much aggregation as possible for the optimal overlay calculated.

In very rare cases the observed $R = \frac{|\text{Output}|}{|\text{Input}|}$ might change mid-way during the data processing. A hypothetical scenario is when the input data and aggregation function belong to the group of $R \geq 1$ (see Table 1), e.g., word count on mismatched dictionaries, but the data that is aggregated on the first level is "semi" pre-partitioned so the first level observes $R = 1$. We consider this as a very unlikely scenario and so ROME does not consider significant changes of R mid-way during the data processing. However, ROME could check the observed R at each level and trigger a re-balance of the remaining overlay in case of a sudden change. Section 6.3 provides an in-depth analysis of R ratio autodetection of ROME.

Table 5. Optimal Value for F to Minimize the Latency of a Single Input Block, and Values ROME Chooses

R	**Optimal fan-in** (cf. Section 4)	**ROME fan-in**
$R < 1$	2	2
$R = 1$	e	3
$1 < R < n$	$\min\left(n, (1 - \log_n R)^{-\log_R n}\right)$	$\min\left(n, \lceil(1 - \log_n R)^{-\log_R n}\rceil\right)$, cf. Section 3.4.1
$R \geq n$	n	n

User-provided. To avoid autodetection of R and its overheads (or to bypass static analysis), a user can always submit a job with an explicit value for R. As a matter of fact, internally in ROME, a negative R value represents the *absence* of such a pre-defined value. We will elaborate on this later in the context of the ROME system and its API (see Section 5.2).

3.3 Overlay Stage

Our goal is thus to find an aggregation overlay with minimal latency for a given aggregation function. Figure 2 shows four overlays created with different fan-ins, yielding equivalent results for compute-aggregate tasks. Smaller fan-ins like that in Figure 2(a) yield higher parallelism at the lowest levels. Figure 2(c) instead obtains all input at the first level of aggregation. That level will thus take longer than a single level in Figure 2(a), but there are fewer levels to run. Consider the aggregation of occurrences of words in a word count job. Each word is considered at each level. More branching increases parallelism, but at the cost of redundancy at multiple levels. To determine the best tradeoff between parallelism and redundancy, we need to reason on the factors that impact the latency when using an aggregation overlay.

3.3.1 Optimal Fan-in. The aggregation time at a level—composed of the time to receive input from the level just beneath it and the time to create the output for the level—depends on the size of the input, some set of partial results $\overline{x}$. We use $g^t(\overline{x})$ to denote the time required by $g(\overline{x})$ to aggregate input of size $|\overline{x}|$, including communication time. Aggregation on the same level of the overlay happens in parallel, so only the time of a single branch is modeled.

The optimal fan-in for an aggregation tree can intuitively be derived from the given aggregation function $g()$ deployed by considering *two* measures of function complexity: (a) A measure of space complexity in the form of the R-ratio. (b) The time complexity based on $g^t()$ [16]. The next section focuses on proving optimal values for R for minimizing aggregation time in several scenarios of (b). However, somewhat skipping forward and maintaining the bigger picture, these proofs will be based on an idealized setting to stay tractable. We thus have to adapt these heuristics in several ways for application in ROME (see Table 5).

3.3.2 Fractions. The first consideration removes fractional fan-ins. The equations in the theoretical model assumes all variables are continuous; however, aggregation must use discrete inputs in real-life. Hence, in practice, a system as to round up (or down) the fan-in F derived from the model, e.g., considering the ceiling $\lceil F \rceil$. For instance, applications such as top-k sorting on pre-partitioned data that output the same size as one input, i.e., $R = 1$, use a fan-in of 3, while the theory would insist on a fan-in of e. Values in the range $1 < R < n$ are likewise rounded to the smallest fan-in with the same height as the suggested ideal. We round up, because rounding down might increase the height of the overlay.

3.3.3 Time Complexity. In Section 4.3, we identify the parameters for aggregation trees and prove their optimal values via mathematical models for various cases. However, after an in-depth analysis of the results, we can observe that the ratio R is often sufficient to guide a system towards the choice of the near-optimal overlay tree without requiring to take into account more complex parameters. For instance (as proved in Section 4.3), the optimal fan-in when $R = 1$ is 2 regardless of the time complexity of the aggregation function, i.e., $g^t\,()$: The linear and super-linear cases are formally proved, while the sub-linear case would be dominated by the linearity resulting from the communication time, hence this last case can hardly be observed in real-world scenarios. Hence, while we do consider time complexity in our theoretical analysis, our heuristics devised in ROME omits this aspect based on our analysis results.

3.4 Mapping Stage

Further heuristics are required in practice to deal with a non-ideal setting, e.g., to deal with non-full trees, and to effectively map a conceptual overlay to an actual application topology.

3.4.1 Balancing Mechanism. A theoretical model can easily assume that overlay trees are perfectly balanced using continuous variables. This often requires fractional tree heights. Applying fan-ins obtained after rounding blindly creates trees where some nodes have more children than others; thus, the coarse-grained heuristics usually result in unbalanced trees that can be tuned further.

Figure 5 shows a simple example of imbalance skewing performance. In Figure 5(a), the model chooses a fan-in of 4 and expects the height of the overlay to be $\log_4 9 \approx 1.58$. The actual height of the associated overlay is 2. There are also nodes with fewer than 4 children, creating a performance skew between branches. Figure 5(b) also has a height of 2. However, the lowest level of the longest branch has a fan-in of 3 instead of 4. Because the input size is reduced the corresponding node will run faster, and there is no increase in height to offset this. Thus, the latency of the slowest branch is decreased.

To analyze this more formally, let us consider an aggregation function $g\,(x_1 \ldots x_\alpha)$. Let c be the amount of time to process one input to the aggregation. If $g\,(x_1 \ldots x_\alpha)$ is linear on the size of its combined inputs, then the latency of a single run of the function is simply $c\,\alpha$. For an unbalanced tree using a heuristic fan-in α, we model the branch with the highest latency, i.e., with most input. This branch has ϵ levels using α, and $(\lceil \log_\alpha n \rceil) - \epsilon$ levels with a fan-in of no more than $\alpha - 1$. Consequently, a total time for the branch can be estimated as follows:

$$\sum_{k=0}^{\epsilon} R^k c\alpha + \sum_{k=\epsilon+1}^{\lceil \log_\alpha n \rceil - 1} R^k c\,(\alpha - 1).$$

Note that aggregation only makes sense for $\alpha \geq 2$, which implies this formulation holds for $R \geq 1$. When $R > 1$, the formulation simplifies to

$$c\left(\alpha \frac{R^{\epsilon+1} - 1}{R - 1} + (\alpha - 1)\left(\frac{R^{\lceil \log_\alpha n \rceil} - R^{\epsilon+1}}{R - 1}\right)\right).$$

Also, when $R = 1$, it simplifies to

$$c\left(\epsilon + 1 + \lceil \log_\alpha (n) \rceil\,(\alpha - 1)\right).$$

These equations are minimal when ϵ is minimized, i.e., when the entire path uses the smaller fan-in.

As long as imbalance remains, we can apply this logic inductively to decrease the fan-in of a node on whichever is the highest latency branch until there are no nodes that have a different number of children than other nodes at the same height. This inductive process creates as balanced an aggregation overlay as possible for a given height and number of leaf nodes.

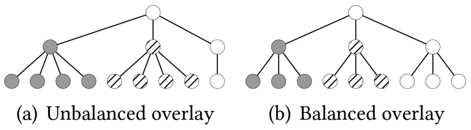


Fig. 5. The effect of the balancing mechanism.

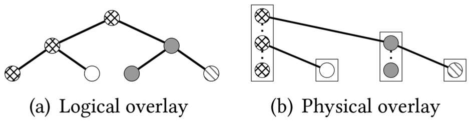


Fig. 6. Reusing workers to reduce resources and latency.

ROME thus implements an explicit balancing mechanism. The system first finds the non-fractional height of a tree using the heuristic. For a heuristically determined fan-in α, this is simply $\lceil \log_\alpha n \rceil$. By our inductive reasoning, ROME finds the smallest fan-in, which creates an overlay with the same height as the heuristic. Thus, the balancing mechanism uses the equation $\lceil \log n / \lceil \log_\alpha n \rceil \rceil$ as the practical fan-in where α is the value returned by the original heuristic. Observe that this only affects $1 \leq R < n$; there is no fan-in less than 2, and any fan-in less than n results in greater tree height.

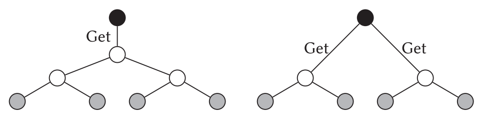
3.4.2 Colocation. Data-intensive aggregations can spend a significant amount of time just sending serialized objects over the network. A natural heuristic for ROME is to colocate distinct aggregation nodes that communicate with each other at the same worker as long as this does not create resource contention. For example, consider the aggregation tree with 4 leaves and fan-in of 2 in Figure 6(a). Since aggregation at different levels does not run concurrently, workers can host nodes along a single branch without resource contention. Nodes of the same color can be harmlessly colocated on the same worker machine. In addition to reducing the number of workers required for an overlay, using RAM for intra-worker communication (dashed lines) instead of the network as for inter-worker communications (solid lines) reduces communication latency. Figure 6(b) shows an optimized physical overlay where boxes represent physical machines. Such optimization reduces the communication time of each parent by a factor of $\frac{F-1}{F}$.

3.4.3 Root Node Bypass. When ROME is integrated into a third-party system, we can improve latency by running the final aggregation in that system instead of in ROME. The aggregation is the same, in fan-in and result, yet we save one overlay level and thus networking.

Figure 7 shows an aggregation performed in this manner. The ROME nodes run most of the aggregation. The results from the level beneath the root are sent to a component of the third-party system, which acts as the root of the overlay and completes the aggregation. Note that in the case of $R \geq n$, which means a fan-in of n, the ROME workers are not used at all. In this case, the overlay contains no ROME nodes, and the entire aggregation is completed in the third-party system.

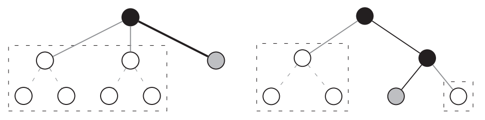
3.4.4 Incremental Computations. Many big data applications deal with dynamic datasets, meaning that new partitions may be added, or existing partitions may have their data changed. Aggregating from scratch upon changes in these cases is not necessary, and highly inefficient. ROME thus allows users to specify two ways for efficiently handling incremental data:

Append-only updates: Append-only data is the norm for some applications and filesystems [21, 46]. In this case, new data is aggregated directly with the results cached at the root of the original overlay. Figure 8(a) illustrates it. The new data in the gray node is aggregated with the result from the unshaded nodes, stored in the black node. This is no more work than the black node would have done during reaggregation anyway, and the rest of the overlay is not needed, freeing up resources and reducing latency. This refined approach can be naturally extended to adding multiple new nodes at once, creating a new overlay where one leaf is the old root node and other leaves compute the new incoming data.



(a) Traditional overlay (b) Optimized overlay

Fig. 7. Final aggregation in ROME vs. a third-party system. Black nodes produce data. Uncolored nodes are ROME workers. Gray nodes are third-party system workers storing results.



(a) Append-only update (b) Infrequent updates

Fig. 8. Minimizing reaggregation. Gray nodes have new or updated data. Black nodes perform (re-)aggregations. Results from boxed nodes are obtained from caches.

Infrequent updates: In this scenario, some existing partitions of the dataset change over time. In this case, the overlay will be kept alive by ROME after the final aggregation, and partial results remain in an in-memory cache managed by each worker. When a new version of a partition reaches a worker, only the aggregation along the path to the root is rerun, since other results are unaffected. Figure 8(b) shows an example where only a single partition (gray node) is changed: only black nodes perform reaggregation.

Table 5 summarizes the parameters of the aggregation overlay ROME uses when running with autodetection. Next, we discuss optimality of these parameters.

4 THEORETICAL OPTIMALITY RESULTS

This section proves optimality of overlay trees for R in idealized scenarios as used for ROME's Overlay phase (cf. Section 3.3).

4.1 Synopsis

We focus on the case where input processed in a single block, and each aggregation results in a single block of its own. This means all of the data is available and computed at a single step at the leaves, then the aggregation tree reads the entire results and outputs a single output without natural breaks, which can be used to distribute the work.

In this case, we are concerned with the optimal latency of processing the block. The number of inputs—including the direct outputs from the computation phase and aggregations of them—that can be aggregated at each node in the tree is variable, as long as each result from the computation phase is included exactly once. Reducing the fan-in means increasing the number of siblings in the tree, which in turn means that more aggregation is done in parallel at each level. However, subsequent levels have to repeat some of the work their children did. Therefore, the optimal fan-in depends on the ratio of work that is repeated to the amount of parallelism gained.

4.2 Notation and Assumptions

We rely on several assumptions to calculate optimality. These do *not* affect the correctness of the output. Table 2 shows the notation used to rigorously define our assumptions and prove optimal fan-ins for the trees.

We assume time complexities of aggregation functions depend solely on input size. Some aggregation algorithms have time complexities expressed in terms of the fan-in and the size of individual inputs, e.g., $O\,(\text{fan-in} \times \log\,(\text{average input size}))$. Our experiments show this deviation is a small factor.

We model communication time as linear on the amount of data transferred. Contemporary streams to a node share bandwidth, so the communication time at a node is the same as long as all children start sending data before the node finishes receiving the data from any set of other nodes. Communication at a node can be affected by other nodes on the network, especially when TCP incast is present [61]. We assume that TCP incast is resolved on the TCP level (cf. Reference [56]) or communication time is relatively inconsequential. We wait for all streams when aggregating at a node to avoid higher programming complexity to account for locking and race conditions. These assumptions require, performance-wise, a pseudo-synchrony that requires a degree of homogeneity among nodes of a level. Heterogeneity may exist across levels.

We do ignore some of the other complexities that can be associated with communication, especially with regard to node location on the network and shared resources or conflicts. We are considering mostly the case of using third-party cloud offerings, which often hide such things from the user. As such, we consider the network architecture a black box best approximated in the average case as mostly uniform.

Homogeneous hardware is a fair premise when datacenters mass order standard hardware, and cloud services provide tiers of service based on performance. The minor service variations can be unpredictable. Homogeneous input follows from our model for data distribution. If aggregation time depends on the data traits other than size (e.g., element order), then those traits must be distributed. We find the leeway in synchrony and the impact of tree customization mask enough heterogeneity in practice.

Aggregation costs are modeled as monotonically increasing on input size and are zero for no input. Modeling non-linear setup overheads complicates analysis, and the practical impact of these overheads is very small. When aggregation changes the size of the data, we model ratio of output to input at each level to be the same. This is or can be made to be true for many applications. In many other cases the ratio stays within a range, e.g., below 1, and the optimal fan-in is unaffected.

We assume the same ideal fan-in is used at all levels of the tree (unlike, say, in Figure 2(d)). Without a rigorous proof of this, we note that once R_1 is applied to the first level, the optimal remaining fan-in is essentially being applied to $\frac{1}{F}$ nodes with input different by a factor of R_1, which is essentially the same problem.

We model full and balanced trees. Means of working around this in practice have been discussed in Section 3.3 and Section 3.4.

4.3 Optimality Proofs

Here, we prove theoretical (near-)optimality of the fan-in F for minimizing latency given n input partitions and $g^t\left(\overline{x}\right)$ for several cases of R when there is a single block of input at each tree leaf.

4.3.1 *Ratio $R < 1$ and $g^t\left(\right) \geq$ Linear.*

Lemma 1. *The total aggregation time with linear $g^t\left(\overline{x}\right)$ and $R \neq 1$ is $a\left(F, n, R\right) = \frac{t\left|x_0\right|F(R-1)}{\sqrt[\log_F n]{R}-1}$, and*

$$\frac{\partial}{\partial F} a\left(F, n, R\right) = \frac{t\left|x_0\right|\left(R-1\right)R^{\log_n d} - \left(1 + \frac{\log R_1}{\log n} R_1^{\log_n F}\right)}{\left(R^{\log_n F} - 1\right)^2}.$$

Proof. For aggregation linear on input size, the time at a level is constant $t \times$ (the input size at the level). Initial input size is $\left|x_0\right|$, and size changes by a factor of R_1 at each of the subsequent $\log_F n$ levels. Thus, total aggregation time is $\sum_{m=1}^{\log_F n} tFR_1^{m-1}x_0$. Pulling out the constants gives $tF\left|x_0\right| \sum_{m=1}^{\log_F n} R_1^{m-1} = \frac{tx_0FR_1^{\log_F n}-1}{R_1-1}$. Recalling $R_1 = \sqrt[\log_F n]{R}$ gives $\frac{t\left|x_0\right|F(R-1)}{\sqrt[\log_F n]{R}-1}$. □

Lemma 2. *The total aggregation time with $R = 1$ is $g^t\left(F\left|x_0\right|\right)\log_F n$.*

Proof. Reusing the logic from Lemma 1 without introducing R gives us $\sum_{m=1}^{\log_F n} g^t\left(F\left|x_0\right|\right)$. This simplifies to $g^t\left(F\left|x_0\right|\right)\sum_{m=1}^{\log_F n} 1$, and then to $g^t\left(F\left|x_0\right|\right)\log_F n$. □

Theorem 1. *The optimal fan-in is* 2 *when* $R < 1$ *and* $g^t\left(\overline{x}\right)$ *is linear or superlinear.*

Proof. By Lemma 1, the total aggregation time taken is $a\left(F,n,R\right) = \frac{tF|x_0|(R-1)}{\sqrt[\log_F n]{R}-1}$, and $\frac{\partial}{\partial F}a(F,n,R)$ is $t\left|x_0\right|\left(R-1\right)\frac{R^{\log_n F}-(1+R^{\log_n F}\log_n R)}{(R^{\log_n F}-1)^2}$. $0 < \log_n F \leq 1$ for $2 \leq F \leq n$, so $\frac{\partial}{\partial F}a(F,n,R) > 0$ for $0 < R < 1$.
∴ The optimal F is 2.

We assumed $g^t\left(\overline{x}_1\right) + \cdots + g^t\left(\overline{x}_m\right) = g^t\left(\overline{x}_1 + \cdots + \overline{x}_m\right)$. As F grows there are more inputs and input size is reduced less, and $g^t\left(\overline{x}_1\right) + \cdots + g^t\left(\overline{x}_m\right) < g^t\left(\overline{x}_1 + \cdots + \overline{x}_m\right)$ for superlinear $g^t\left(\overline{x}\right)$, so superlinear $g^t\left(\overline{x}\right)$ is more sensitive to F than linear $g^t\left(\overline{x}\right)$.
∴ This result holds for superlinear $g^t\left(\overline{x}\right)$. □

Intuition. *Consider the case where you are aggregating two inputs at one node. If the aggregation time is linear or super-linear, then adding two more inputs would at least double the aggregation time at that node. Alternatively the other two inputs could be aggregated in parallel, and the two results could be aggregated at the next level. Because each output is smaller than the original input, the second layer of aggregation is faster than the first layer, meaning the total time is less than twice that of the first layer. As long as the first layer reduces the size of the input, it more than offsets the extra layers of aggregation.*

4.3.2 *Ratio* $R = 1$ *and Linear* $g^t\left(\right)$.

Theorem 2. *The optimal fan-in is* e *when* $R = 1$ *and* $g^t\left(\overline{x}\right)$ *is linear.*

Proof. With $g^t\left(F\left|x_0\right|\right)\log_F m$ from Lemma 2 and linear $g^t\left(x\right)$,

$$a\left(F,n,R\right) = F\left|x_0\right| t\log_F n.$$

$$\frac{\partial}{\partial F}a\left(F,n,R\right) = \frac{\left|x_0\right| t\left(\log F - 1\right)\log n}{\log^2 F}, \text{ which is 0 iff } F = e.$$

$$\frac{\partial^2}{\partial F^2}a\left(F,n,R\right) = -\frac{\left|x_0\right| t\left(\log F - 2\right)\log n}{F\log^3 F}.$$

At $F = e$, $\frac{\partial^2}{\partial F^2}a\left(F,n,R\right) > 0$, so this is a minimum.
∴ The optimal F is e. □

Intuition. *In this case increasing the number of inputs increases the aggregation time at a layer, but it does not change the aggregation time at the subsequent layer. However, if enough inputs are aggregated at a single layer, then it can reduce the height of the tree. If we start with a binary tree, then it does not take much to reduce the height by one layer. However, if we already have a wider fan-in, then it takes progressively higher increases in fan-in to reduce the height further due to the nature of the* log *function. Therefore, it makes sense that we have a fan-in that is close to, but not quite, 2.*

4.3.3 *Ratio* $R = 1$ *and Superlinear* $g^t\left(\right)$.

Theorem 3. *The optimal fan-in is* $[2, e)$ *when* $R = 1$ *and* $g^t\left(\overline{x}\right)$ *is superlinear.*

Proof. By Lemma 2 the total aggregation time is $a_{linear}\left(F,n,R\right) = g^t\left(F\left|x_0\right|\right)\log_F n$. As shown in Theorem 2, $\lim_{g^t(\overline{x})\to linear}$ is e. We can assume $\frac{\partial^2}{\partial F^2}a_{superlinear}\left(F,n,R\right) > \frac{\partial^2}{\partial F^2}a_{linear}\left(F,n,R\right)$. Thus, $\frac{\partial^2}{\partial F^2}a_{linear}\left(F,n,R\right) > 0$ for $F \geq e \implies \frac{\partial^2}{\partial F^2}a_{superlinear}\left(F,n,R\right) > 0$. Thus, any minimum occurs at $F < e$.

$\therefore$ The optimal value of F is in the range $[2, e)$ □

Intuition. *Compared to the case where aggregation time is linear, this scenario has a greater increase in the aggregation time when the fan-in increases. There is still a benefit to decreasing the height of the tree, but that offset needs to be more significant, especially for very superlinear aggregations. Therefore, the optimal fan-in is somewhere between 2 (the quickest aggregation at each level) and the optimal time calculated for linear aggregation.*

4.3.4 *Ratio $1 < R < n$ and Linear g^t ().*

Theorem 4. *The optimal fan-in is $(1 - \log_n R)^{-\log_R n}$ when $1 < R < n$ and $g^t(\overline{x})$ is linear.*

Proof. From Lemma 1, the amount of time taken to aggregate is $a(F, n, R) = \frac{tF|x_0|(R-1)}{\sqrt[\log_F n]{R}-1}$, and $\frac{\partial^2}{\partial F^2} a(F, n, R) = \frac{t|x_0|(R-1)}{(R^{\log_n F}-1)^2}(R^{\log_n F} - (1 + R^{\log_n F}\log_n R))$.
For $R > 1$, $\frac{t|x_0|(R-1)}{(R^{\log_n F}-1)^2} > 0$, so the expression is 0 iff $R^{\log_n F} = (1 + R^{\log_n F}\log_n R)$, which happens at $F = (1 - \log_n R)^{-\log_R n}$.

$\frac{\partial^2}{\partial F^2} a(F, n, R) = \frac{t|x_0|(R-1)R^{\log_n F}\log R(\log n+\log R-(\log n-\log R)R^{\log_n F})}{F\log^2 n(R^{\log_n F}-1)^3}$.
Because $\frac{t|x_0|(R-1)R^{\log_n F}\log R}{F\log^2 n(R^{\log_n F}-1)^3} > 0$, $\frac{\partial^2}{\partial F^2} a(F, n, R) > 0$ iff $\log n + \log R - (\log n - \log R)R^{\log_n F} > 0$. Substituting the extrema value for F gives $\frac{R^{-\log_R(1-\log_n R)}-1}{R^{-\log_R(1-\log_n R)}+1} - \log_n R < 0$. To prove this, we fix n and find R to maximize $b(F, R) = \frac{R^{-\log_R(1-\log_n R)}-1}{R^{-\log_R(1-\log_n R)}+1} - \log_n R$. $\frac{\partial}{\partial R} b(F, R) = \frac{2\log^2 n+\log^2 R-4\log n\log R}{R\log n(\log R-2\log n)^2}$. $\frac{\partial}{\partial R} b(F, R) > 0$ for $1 < R < n$. Thus, $\max(b(n, R))$ occurs at $\lim_{R\to n}$, and $\lim_{R\to n} b(n, R) < 0$. The extrema of $a(F, n, R)$ is a minimum.

$(1-\log_n R)^{-\log_R n} > n \implies d > n$, which is impossible. Since the only local extrema is a minimum, $\frac{\partial}{\partial F} a(F, n, R) < 0$ for $F = [2, (1 - \log_n R)^{-\log_R n}]$, so the optimal fan-in is the largest possible value, i.e., n, in this case.

$\therefore$ The optimal F is $\min(n, (1 - \log_n R)^{-\log_R n})$. □

Intuition. *As R grows, there is less incentive to have trees with more height, because each level ends up redoing more and more work of the levels below it. So, it makes sense that the optimal fan-in (and height) fall between the values calculated for $R = 1$ and $R > n$. That $\lim_{R\to e^+}(1 - \log_n R)^{-\log_R n} = e$ is a nice sanity check that the equation is continuous with the previous proof.*

4.3.5 *Ratio $R > n$.*

Theorem 5. *The optimal fan-in is n when $R \geq n$.*

Proof. The time taken by the root node is $g^t(R_1^{\log_F n}|x_0|)$. $R \geq n \implies R_1 \geq F$ and $\log_F n \geq 1$, so this is minimal at $\log_F n = 1$. In addition, for $F < n$ the rest of the tree takes non-zero time.
$\therefore$ The optimal F is n. □

Intuition. *In this formulation, the aggregation at each subsequent layer is slower than the sum of the layer below it. Therefore, it makes sense to have the shortest tree possible.*

Table 6 summarizes the results from our proofs. There are still unproven cells where the degree of sub- or superlinearity of the aggregation is required to find the optimal value. There is always an aspect of linearity to $g^t(\overline{x})$ due to communication time, so it makes sense to use the results from the linear cases on the sublinear cases, which would complete most of the table.

Table 6. The Optimal Value for F to Minimize the Latency of a Single Input Block and the Applicable Proof When Available

R	Optimal fan-in	Model runtime	Sublin. g^t ()	Linear g^t ()	Superlinear g^t ()
$R < 1$	2	$\frac{t\lvert x_0\rvert F(R-1)}{\log_F \sqrt[n]{R}-1}$	*unproven*	Theorem 1	Theorem 1
$R = 1$	e	$tF\lvert x_0\rvert \log_F n$	*unproven*	Theorem 2	Theorem 3 (near opt.)
$1 < R < n$	$\min\left(n, (1 - \log_n R)^{-\log_R n}\right)$	$\frac{t\lvert x_0\rvert F(R-1)}{\log_F \sqrt[n]{R}-1}$	*unproven*	Theorem 4	*unproven*
$R \geq n$	n	$\frac{t\lvert x_0\rvert F(R-1)}{\log_F \sqrt[n]{R}-1}$	Theorem 5	Theorem 5	Theorem 5

As announced in Section 3.3.3 our analysis indicates that time complexity of aggregation functions has little to no impact on distributed aggregation time (e.g., the same fan-in of n is optimal for $R \geq n$ regardless of sublinear, linear, or superlinear time complexity for g^t ()), leading to simpler heuristics.

5 ROME SYSTEM

We present our ROME system for optimized compute-aggregate task processing leveraging the heuristics presented in the previous sections. We focus on its architecture, API, fault tolerance support, and integration into general-purpose data analytics frameworks.

5.1 System Architecture

Figure 9 shows the architecture of ROME, which is implemented in Java (6,000 LoC). There are two core components inside ROME: *(i) workers* that are deployed on all nodes and *(ii)* a *controller* that coordinates workers. A full processing environment also requires an *invoker*, a set of *producers*, and a *consumer*. These can be implemented inside ROME, but will typically reside in a third-party framework. More precisely, the complete set of components/component types in ROME with their respective duties is as follows:

Workers: aggregate data received from other nodes and send the results to their parent in the aggregation tree. If requested, then results are stored inside an in-memory cache to avoid repetitive aggregations.

Controller: directs the workers in creating and maintaining the overlay, tracks the status of each worker, and repairs any active aggregation overlay and restarts aggregation as needed if a worker leaves the system (or fails; see Section 5.3).

Invoker: a client that interacts with the ROME controller to initialize the overlay and relays relevant data between a third-party analytics system and ROME. This role is typically played by the "job manager" of the former system.

Producers: feed the data into the leaf nodes of the aggregation overlay. The data may be read directly from a local or distributed file system, but typically producers are worker components of a third-party system.

Consumer: receives the final aggregation result. This role is typically played by a worker of the third-party system.

```java
interface Accumulator<T> extends Externalizable {
   T get();
   void add(List<Accumulator<T>> list);
}
```

Listing 1. `Accumulator` interface (`public` visibility).

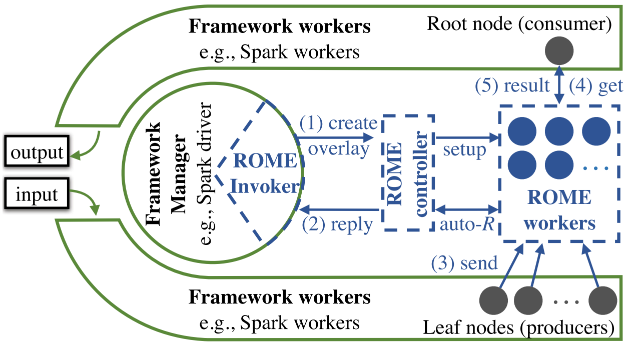


Fig. 9. Architecture of ROME.

Table 7. Examples of `Accumulator` Implementations

Accumulator	**Description**	**Lines of code**	R
`MergeLists`	Merge sorted lists into a list of all elements in sorted order.	79	n
`TopKSort`	Output a sorted list of the k highest scoring elements from sorted input.	71	1 on pre-partitioned, ≥1 on arbitrary data
`LCS`	Output longest substring, which is contained in all sequences.	62 (without generalized suffix tree)	<1
`WordCount`	Take mappings of `key`↦**`int`** and combine the counts of same keys.	114	≥1
`SVM + SGD`	Iterative gradient descent on mini-batches.	118	1

5.2 API

Because we assume aggregation is associative, we need a standard interface to link up the aggregating nodes. The `Accumulator` interface is shown in Listing 1. There are only two necessary functions.

(1) `add()` simply allows an input to be added or replaced. When a producer or worker lower in the overlay calls `add()` the data is placed in the worker. If data already exists for that child, then the existing data is replaced. If after the call there is data from every child available, then aggregation at that worker begins.
(2) `get()` retrieves the result. A communication manager on each worker thus fetches ("`get()`s") results from its node and sends them to its parent.

Each aggregation requires a separate `Accumulator` implementation. We implemented several types of `Accumulators` during our experimentation. Table 7 shows how little implementation complexity they require. The entire implementation of an `Accumulator` is often less than 100 **lines of code (LoC)**.

Table 8 outlines our simple API, and Figure 9 shows where in the workflow the API calls occur. An invoker calls `initialize()` with a list of nodes ROME can use for the workers and controller. To start a new aggregation, the invoker calls the `createOverlay()` procedure (step 1 in Figure 9)

Table 8. Core ROME API

Signature	Action	Return value
`initialize(nodes)`	Setup	–
`createOverlay(id, R, n, flags)`	Overlay configured	List of nodes to send data *and* root node
`send(id, Accumulator, node)`	`Accumulator` sent to `node`	–
`get(id, root)`	Get result from `root`	Aggregation result
`releaseOverlay(id)`	Overlay dissolved	–

with an ID to uniquely identify the overlay. This function sends *R* and a number of leaf nodes to the controller. A flag specifies whether to enable incremental computations (cf. Section 3.4.4). If there are not enough available nodes, then the controller returns an error; otherwise, it sets up an overlay and replies (step 2) to the invoker with a list of ROME workers where producers should send their data and the worker that is the overlay root. The overlay is maintained until the invoker calls `releaseOverlay()`. At that point, workers drop their connections and release any cached partial results.

A leaf node forwards its local compute results to the assigned ROME worker by calling the `send()` procedure (step 3). Each ROME worker independently merges the received `Accumulators` from its children and then sends the result to its parent in the aggregation tree.

The node that receives the final result of the aggregation (the consumer) invokes `get()` (step 4). When the aggregation is completed by the root, the result is returned (step 5).

Note that to request autodetection of *R*, when calling `createOverlay()`, an invoker simply specifies a negative *R*. Since ROME will not build the full aggregation tree, the reply received by the invoker from the controller will contain an invalid root node. When the consumer invokes `get()`, our framework transparently retrieves the actual root from the controller to obtain the aggregation result.

5.3 Fault Tolerance

Our fault recovery strategy [30] is similar to that of Spark [60]. We maintain data from unaffected portions of the overlay and only recompute what was lost. For some component recoveries, we rely on the fault tolerance mechanism of the third-party system without hampering safety or liveness. Below, we discuss the process for each component:

Controller: Apache ZooKeeper [24] provides services to build reliable distributed coordination including a hierarchical key-value store. ZooKeeper can be utilized to maintain the controller's state and make it fault tolerant. If ZooKeeper's scalability is an issue for large deployments with several hundreds of workers, then heartbeats from workers can be rerouted with techniques similar as in Heron [31] (using special "heartbeat" daemons on separate machines that consume all the keep-alive heartbeat traffic) to further reduce load on ZooKeeper. In our deployments, the controller load was low enough that this was not necessary.

Invoker: If a failure happens during `createOverlay()`, then the controller aborts the overlay. Otherwise, the new instance of the invoker can recover details about a previously created overlay using the unique overlay identifier used in creation.

Producer: The responsibility for producers remains with the invoking system. If a producer fails after calling `send()`, then it is presumably restarted and calls `send()` again. The second call is ignored unless the overlay is configured to accept incremental changes. If so, then the new `send()` is treated like new data and data is reaggregated along that branch.

Consumer: Likewise, the third-party system is responsible for recovering consumers. Any consumer wishing to receive the aggregation result can invoke `get()`. This can be done multiple times (it is idempotent) by different consumers as long as `releaseOverlay()` has not been invoked.

Worker: Workers send heartbeat messages to the controller. They also notify the controller if another worker is not responding to attempts to send data along the overlay. Upon suspecting a worker failure, the controller creates a new worker at an unused node. It also notifies the failed worker's parent and children and sends them the address of the new worker. If a worker fails after it sends all its results to its parent, then the new worker (and its children) do not need to perform any additional task for recovery. Otherwise, the children need to (re)send their results to the new worker. Observe that if a child of a failed worker misses a portion of the result in its memory, then it needs to recursively ask for the corresponding portions from its own children. ROME has no direct hooks into third-party systems. Thus, in the case of producers part of such a failure, ROME kills the processes at the end of the associated branches. This forces them to restart and re-`send()` their data. If no unused nodes are available, then ROME returns an error to the invoker.

5.4 Integrating ROME

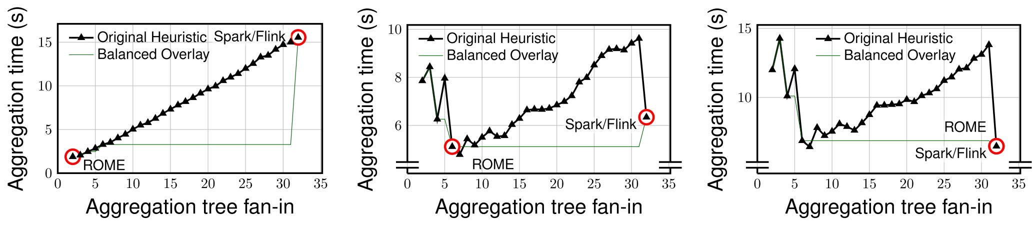
We designed ROME to be easy to integrate with popular more generic data analytics systems. Developers should be able to accelerate their system when performing aggregations with minimal implementation effort. In this section, we describe how we have integrated ROME in Apache Spark [4] 2.4 and Apache Flink [3] 1.1. We chose a more stable version of Flink in favor of some benchmarks, however, we adapted a very recent version of Spark to benefit from recent updates (see Section 2.3) on `treeReduce`. For both systems, less than 350 LoC were needed for successful integration.

5.4.1 Spark. We extend Spark's API with a `reduceWithROME()` operator. Besides the aggregation function, the user can also provide the output to input ratio R. If R is unknown to the user, then a negative value can be passed in its stead.

As discussed in Section 5.2, aggregators in ROME implement the `Accumulator` interface. Our class `SparkAccumulator` wraps the aggregation function provided by a user and transparently reuses several utilities supplied by Spark (e.g., serializers). To execute `reduceWithROME()`, the following changes to Spark were also required: *(i)* during the submission of a job using `reduceWithROME()`, the Spark driver invokes `createOverlay()`; *(ii)* upon executing the job, each Spark worker wraps the local data and the aggregation function into a `SparkAccumulator` and sends it to ROME; *(iii)* the Spark driver invokes `get()` as many times as needed to get the partial results from ROME and then does the final aggregation.

5.4.2 Flink. We implement a `reduceWithROME()` method to extend the `ReduceFunction` interface. The method is parameterized by the ratio R. Again, a negative value can be passed to indicate an unknown R. We also support a variant of `ReduceFunction` called `GroupReduceFunction`, which runs an aggregation over an object *list* instead of a single pair.

As with Spark, integrating ROME with Flink requires little effort. We provide a `FlinkAccumulator` to wrap the `ReduceFunction` provided by the user and some Flink-specific utilities. The following changes are also made to Flink: *(i)* the Flink job manager calls the `createOverlay()` procedure if `reduceWithROME()` is used; *(ii)* each Flink worker forwards a `FlinkAccumulator` object with the results of local computation on its data partition to ROME using the `send()` procedure; *(iii)* the Flink worker elected by the job manager for performing the final aggregation retrieves the partial results from ROME via `get()`.



(a) Varying fan-ins, $R = \frac{1}{n}$ (b) Varying fan-ins, $R = n^{0.8}$ (c) Varying fan-ins, $R = n$

Fig. 10. Overlay comparison for various values of R.

6 EVALUATION

In this section, we evaluate our heuristics and ROME integrated into Flink and Spark in Amazon AWS.

6.1 Overlay Evaluation

First, we evaluated the accuracy of our heuristics against one-size-fits-all overlays with simulated workloads on m3.medium nodes started from a single image. All nodes were provided on demand, so no network location or locality information was available. For the overlay evaluation, we measured aggregation latency in isolation from computation. To that end, aggregation was delayed until all leaves completed computation. The controller then initiated aggregation and timed from that point until the root node reported completion.

6.1.1 Varying Fan-in. Our first experiments verified our heuristics and balancing mechanism on a system with 32 leaf nodes. The compute phase generated a set number of random integers. Aggregators read the size of their input and use a given R to calculate their output size before generating another list of random numbers to fill that output.

Figure 10 shows the average time to aggregate across 25 runs of each possible fan-in tree for each of 3 R values both with and without the balancing mechanism. Overlays using the original heuristic assigned children to an aggregator until the prescribed fan-in was met, then continued with the next node in a left-to-right fashion. Balanced overlays applied the mechanism on the fan-in before constructing the trees. This means there were often large jumps in performance when the chosen fan-in crosses a threshold changing the height of the overlay.

We circle the performance of two overlays in each graph to highlight the comparison of our performance to the overlay of fan-in n commonly seen in practice. The second overlay is unavoidable for total aggregation problems when aggregating by partition, such as with Spark without `treeReduce` or Flink as described in Section 2.3.

For both $R = \frac{1}{n}$ (Figure 10(a)) and $R = n$ (Figure 10(c)) ROME's heuristics correctly chose the fastest aggregation overlay. This is especially visible with $R = \frac{1}{n}$, as the fastest overlay outperformed the slowest overlay of a single aggregation, used in frameworks aggregating by partition, by 86%.

When $R = n^{0.8}$ (Figure 10(b)), there was an overlay outperforming our chosen one by 6%. This very small difference happened in one overlay adjacent to the chosen one, suggesting the heuristics are successful at finding a nearly optimal overlay.

The difference between the balanced and unbalanced overlays is most obvious in the range of [7, 31], where the unbalanced overlays become significantly more skewed. In this range all the unbalanced overlays diverged from the balanced ones. This is most notable in the case of $R = n$, when the idealized model [18] monotonically decreases, because it allows fractional overlay height. Given that the trend increases in that range in practice and that the balanced overlays

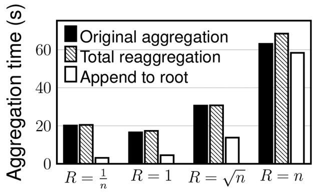


Fig. 11. Strategies for new data.

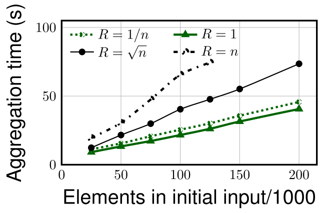


Fig. 12. Varying input size.

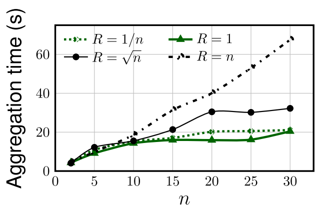


Fig. 13. Varying leaf node #.

thus outperformed the unbalanced ones, we can assert that our balancing mechanism effectively improves performance.

6.1.2 Append-only Updates. Next, we tested our append-only update technique by appending a single node to an existing 25-node overlay. Figure 11 shows the difference between creating a new overlay with 26 leaves vs. aggregating the prior output with the new data.

Complete reaggregation took slightly longer than the original aggregation. Aggregating the previous results with the new data was faster in all cases, although the amount of savings depended on R, i.e., on the size of the output of the previous data. With $R = \frac{1}{n}$, aggregating the prior output with the new data directly was 85% faster than using a new overlay. With $R = n$ the benefit was a 15% speedup.

If the overlay was reused each time, then the second approach would require time equal to the original aggregation time plus the time to aggregate those new results with the data from the new node. In all cases that combined time was greater than the cost of using a new overlay. Thus, it makes sense to use this approach only for append-only data.

6.1.3 Input Size and Distribution. To fully understand the performance of ROME, we next considered the effects of the number of leaf nodes and the input data size. These are environmental parameters imposed by the user and problem and are thus not configurable by ROME. Nonetheless, both of course significantly affect performance. For each of the chosen values of R, we varied the number of leaves n from 3 to 30 and, independently, the input size from 25,000 elements per leaf to 200,000.

Figure 12 shows the effect of changing the input size across 25 nodes using the same R values as the earlier experiments. The final two points for $R = n$ are omitted, as memory requirements at the root exceeded the available RAM, and latency increases dramatically with disk accesses. With the exception of the omitted points, latency is very predictably correlated with the input size.

Figure 13 shows how the number of leaves affects the latency, with the input size per leaf remaining the same. Thus, a cluster with 20 leaves is processing twice the total amount of input data as a cluster of 10 leaves. Observe that smaller values of R, which have smaller fan-ins and thus higher parallelism, are less sensitive to changes in the number of nodes. The latency increased in a sublinear fashion relative to the number of leaves, suggesting that the job should be parallelized to a larger degree when possible.

For $R = n$, the relationship is predictably linear. This is not surprising, given that all aggregation takes place at a single node, which means that we were increasing the workload of a single machine by a factor equal to the change in fan-in.

6.2 Integrated Evaluation

We compared performance of unmodified versions of Spark and Flink to versions integrated with ROME. In the unmodified systems, each worker node was placed on a separate AWS instance and used the whole available main memory.

In the case of Spark+ROME (respectively, Flink+ROME), a ROME node is colocated with a Spark (or Flink) worker node and uses half of the instance's memory. We differentiate between ROME-manual and ROME-auto. The first represents a user providing the correct value of R, while the second represents a user providing a negative value, meaning our system autodetects R with a single layer of aggregation (cf. Section 3.2). Additional details on how ROME is integrated into Spark and Flink can be found in Section 5.4.

Flink and Spark use a fixed fan-in of the number of worker nodes. We also compared to Spark using `treeReduce` to specify tree depth (see Section 2.3 for a detailed discussion of this functionality). For Spark using `treeReduce`, the user sets only an upper bound of the total tree height (depth) of the aggregation tree. Hence, Spark is forced to fit the aggregation overlay to the number of partitions, workers, and given (max) height, which ignores the actual aggregation function. Since `treeReduce` uses a depth of 2 as default, but takes a user-provided depth as an upper bound, we show results for both `treeReduce` with default depth and `treeReduce`(d) with d equivalent to the optimal (but unbalanced) overlay of ROME.

We chose three experiments exploring behavior in different ranges of R. The first experiment considers a longest common substring problem where $R < 1$. The second experiment considers a top-k sort on not pre-partitioned data comprising two aggregation phases, with $n \geq R \geq 1$ and $R = 1$, respectively. The third experiment runs an iterative algorithm—a gradient descent with mini-batches with the goal of learning a classification problem. Each iteration runs an aggregation phase with $R = 1$.

We run each experiment 10 times and report the average and standard deviations as error bars.

6.2.1 Longest Common Substring. We ran the **longest common substring (LCS)** problem on a DNA dataset [49]. Worker nodes were assigned unique DNA sequences from a section of the genome. Each worker built an suffix tree containing all the substrings contained within its sequence. This data structure was then compared to those from other workers to remove substrings that are not contained in all sequences. Since the output size was smaller than the input, the fan-in for ROME is 2.

Because of the amount of computing power and the memory requirement to aggregate at a single node with Flink and Spark, we chose m3.2xlarge nodes on AWS, which have 8 virtual CPUs, 30 GB of RAM, and 2 SSDs for permanent storage. Since the suffix trees necessary for aggregation are about 700 MB per sequence, Flink is unable to process the results from 32 sequences, even with 30 GB of RAM. We thus run Flink with 16 sequences, but Spark with 16 and 32 sequences.

We run three variants of this test. The first, labeled "16 Seq. (Agg)," reads the precomputed data structures from disk. The time to read 700 MB from an SSD is minimal, so this essentially isolates the aggregation phase for sequences without needing to modify Flink or Spark with a stopping mechanism between compute and aggregate to synchronize a timer. The second test, "16 Seq. (Comp-Agg)," represents the entire compute-aggregate workload for 16 sequences including building the LCS structures online from the DNA sequence. Similarly, we have a "32 Seq. (Comp-Agg)" test, which runs the entire compute-aggregate job with 32 sequences on 32 nodes. The Comp-Agg variants show how much effect the improvement on aggregation time has on the total job latency. The times in Figures 14 and 15 are all normalized to the time that ROME-manual (fastest system) takes.

Figure 14 shows the results when running Flink. Flink+ROME improves latency by a factor of 3.68 over Flink when reading the precomputed data structures ("16 Seq. Agg"). Computing the 16 LCS structures online increases job latency, which lowers the benefit of running Flink+ROME—a speedup of 2.42 over Flink when considering the entire compute-aggregate workload for 16 sequences ("16 Seq. Comp-Agg"). We see that if a user provides R to Flink+ROME then that brings

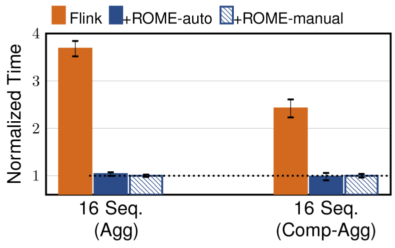


Fig. 14. LCS with Flink.

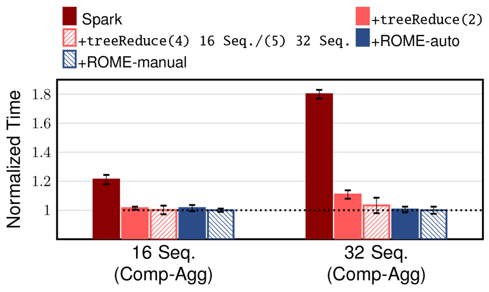


Fig. 15. LCS with Spark.

almost no benefit over ROME to autodetect R, since autodetection uses the optimal R anyway during the probing phase.

Figure 15 shows the LCS problem using Spark with 16 and 32 DNA sequences. The 32 Seq. (Comp-Agg) test taxes the standard Spark system. In part because the memory requirement at the aggregating node is very close to the total allotment, the execution takes 260% of the time of the comp-agg job on 16 sequences. The fastest configuration is Spark+ROME when R is provided, reducing total time for the entire compute-workload by a speedup of 21% and 80% over vanilla Spark when working on 16 and 32 sequences, respectively. The overhead of automatically detecting R is less than 1%, simply because autodetection runs the initial aggregation phase with the optimal R.

Using `treeReduce` for the 16 sequences with the manually calculated ideal height (4) shows same performance as ROME when running with autodetection. However, `treeReduce` becomes less efficient when running the problem of 32 sequences (using optimal height 5), adding an overhead of almost 5% compared to ROME with autodetection. Spark `treeReduce` with the default configuration (height 2) shows similar performance for the small problem of 16 sequences, but adds a penalty of 11% over ROME with autodetection when running the problem of 32 sequences.

The sublinear relationship between latency and the number of machines matches the earlier results and shows ideal overlays are more important as data sizes grow.

6.2.2 Top-k Sort. Next, we analyzed Wikipedia page accesses [55] to find the top-k most visited pages. We distributed 35.6 GB of relevant data (with a total of 706, 639, 151 words), part of a much larger dataset available for more expressive querying, across 32 i3.large workers. The benchmark calculated each page's score in a first aggregation phase (R-ratio s.t. $n \geq R \geq 1$). Then, in a second aggregation phase each compute task found the k most visited pages in its partition. Aggregation involved finding the k highest scoring pages from all those, so the R-ratio of the second phase is 1. This benchmark stresses the R autodetection of ROME, since ROME uses a non-optimal temporary fan-in while running the first intermediate aggregation.

Figure 16 shows the completion time for 3 different values of k. We normalize the values in this graph to the time Flink+ROME takes when R is provided to determine the pages with the top k% most accessed pages. When k is 0.1%, autodetecting R adds less than 3% overhead. Running vanilla Flink requires almost 2 times as long as running Flink+ROME.

Increasing k to 0.2% minimally impacts the compute phase but doubles the aggregation load. As a result, the Flink vanilla deployment adds a slowdown of factor 2.3 compared to Flink+ROME with manual configured R. Flink+ROME autodetection overhead grows to 7%.

When we increase k to 0.5%, garbage collection comprises a third of the resulting runtime for vanilla Flink. In contrast, garbage collection does not affect either Flink+ROME setup despite the reduced RAM allocation to each worker. As a result, ROME shows a speedup of almost 2.7 over vanilla Flink, with only 9% overhead to autodetect R.

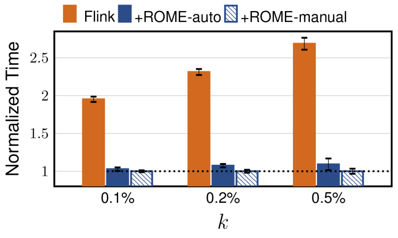


Fig. 16. Top-k with Flink.

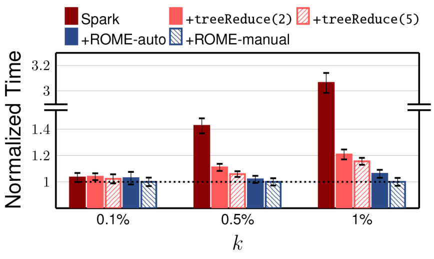


Fig. 17. Top-k with Spark.

Figure 17 shows the Top-K sort problem when using Spark with different configurations for k. Spark shows a more efficient resource usage compared to Flink, so we run the problem with larger values of k. We normalized the values in this graph to the time ROME takes with manual configured R.

For the smallest k (0.1) all systems perform similarly (vanilla Spark adds a penalty of 4% over ROME), since the aggregation load is very small. Increasing k to 0.5% leads to five times higher aggregation load, which unveils differences in performance. ROME with autodetection adds 1.9% overhead compared to manual configuration of ROME, but vanilla Spark needs 43% more time. With Spark's default depth of 2, `treeReduce` is 11% slower than ROME; using `treeReduce` with the same depth as ROME is 6% slower.

When k was increased to 1% the aggregation overlay became even more important. Autodetection of ROME adds 6% overhead compared to a manually configured ROME. `treeReduce` with manually set optimal tree depth (same as ROME) adds 16%, while default `treeReduce` adds 21%. vanilla Spark took 3× more than ROME.

This experiment also showed a worst-case scenario for ROME-auto during the first experiment phase with $n \geq R \geq 1$. When R is autodetected there must be a level of aggregation to learn R, adding overhead in this case.

6.2.3 Gradient Descent. **Gradient descent (GD)** is an iterative method for optimizing a differentiable objective function by updating the parameters of the function in the opposite direction of the gradient of the function. GDs are widely used by many data-intensive machine learning tasks including training of neural networks [43]. Mini-batch GD is a variant that uses a small (randomly sampled) subset of the data to perform the GD update (instead of the complete dataset in each iteration). Spark MLlib[1] provides a GD implementation supporting mini-batching. In each iteration, a mini-batch uses an aggregation task with a `treeAggregate` overlay (used internally by `treeReduce`) for computing and summing up the subgradients, hence an aggregation with $R = 1$.

In this benchmark, we trained a **Support Vector Machine (SVM)** using Spark's MLlib (`SVMWithSGD`) to perform binary classification. We used a 21.4 GB large dataset from KDD Cup 2012 [47], which holds feature vectors and corresponding classification; in total, 55 million binary features [27].

We compare Spark using the default implementation of MLlib's `SVMWithSGD` (using `treeAggregate`) and a modified version using ROME as an aggregation overlay using 32 i3.xlarge workers. The ROME aggregation function uses the same logic as the default `SVMWithSGD` function, hence only 118 LoC were changed for integrating the ROME overlay (mostly accounting for wrapping/boilerplate code). We set the number of iterations (from 1 to 100) used by SVM to learn and

[1]https://spark.apache.org/mllib/.

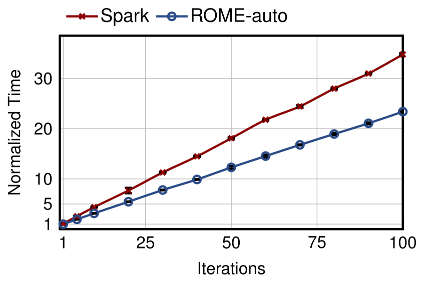


Fig. 18. Gradient descent with Spark.

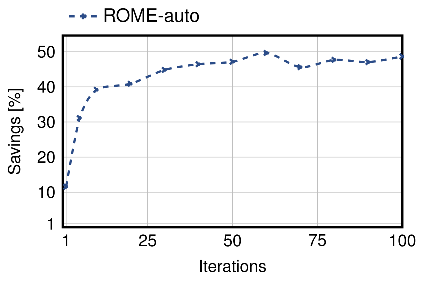


Fig. 19. Gradient descent time savings.

Table 9. Network Optimizations

	No colocation	**Colocation**
No root bypass	166.4 ± 5.1	158.3 ± 3.5
Root bypass	160.5 ± 4.1	154.2 ± 2.6

Average and standard deviation in seconds.

ignore earlier convergence, so all iterations run. Using ROME for aggregation does not change PR/ROC (quality) so we focus again on total running time.

Figure 18 shows the time for varying number of iterations and Figure 19 the savings of ROME over Spark. We normalized the values in this graph to the time ROME needs for 1 iteration (2.33 seconds). When running a single iteration, Spark is 11% slower than ROME, even though ROME needs to set up a larger aggregation overlay. With increasing number of iterations, the advantage of ROME over Spark increases. At 10 iterations, ROME runs 28% faster, at 20 iterations 40% faster, and at 100 iterations 50% faster than Spark. The trend indicates higher savings with more iterations.

6.2.4 Parent-child Colocation and Root Node Bypass. Our next tests were for the two location-based optimizations. Since both the root node bypass and node colocation optimizations were targeted at reducing network latency by decreasing the amount of traffic, we reran the top-k experiment with $k = 1\%$. With its slightly higher network component, this test was best suited to see small but significant network latency savings. We reran the test using four configurations of Spark+ROME—with and without root node bypass, with and without parent-child colocation (see Table 9). The time taken with both mechanisms applied was 7.3% faster than when neither was applied. Applying only colocation or root node bypass only saved 4.9% or 3.5%, respectively. The combined savings was less than the sum of the two, because we could not colocate the final parent in the case of root bypass.

6.3 Autodetection

Next, we investigate the R ratio autodetection of ROME when running the top-k experiment with Spark (Section 6.2.2). We ran the experiment on i3.large instances with a group of 16 workers and 32 workers and vary k from 0.1% to 1%. We report the measured R ratio during the autodetection phase when running the aggregation phase to find the k most visited pages in each task's partition. In theory, if all partitions have at least k unique (no overlaps across partitions) pages, and if all partitions are of same size, then the R ratio should be 1.

Figure 20 shows the measured R ratio when running ROME with Spark on the same dataset, but using 16 and 32 workers, with varying k. Hence, a bar for 16 workers summarizes all 8 reported R ratio values of the partial aggregation. When running with 16 workers, the R ratio starts at .96

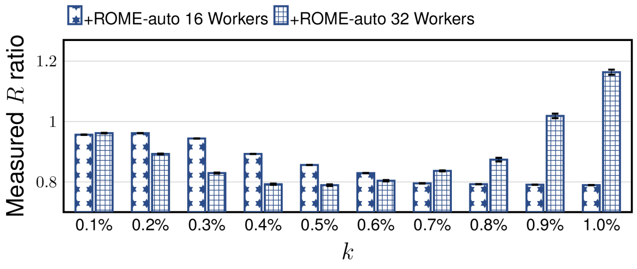


Fig. 20. ROME R ratio autodetection running top-k with Spark.

and goes down to .78 at $k = 1\%$. When running the same experiment using 32 workers, the R ratio starts at .96 but goes down to .79 at $k = 0.4\%$ and grows to 1.16 at $k = 1\%$. Furthermore, standard deviation was less than 0.01 for all configurations. Across all configurations of this experiment, the average R ratio is 0.88 with standard deviation of 0.10.

This experiment shows the advantage of ROME's autodetection over a manually configured system, simply because the R ratio depends highly on the actual setup of an experiment. Not only the input data can cause a change in the R ratio, but also a change in the number of workers or a parameter update of the aggregation function. This is one of the key benefits of using ROME over other tree overlays like Spark's manually configured `treeAggregate`—ROME automatically adapts to the actual situation.

The very low standard deviation for each of the configurations highlights that within a single setup, R ratio does not vary much. This allows ROME to run autodetection only once and use the detected value for the remaining aggregation without any drawback.

6.4 Fault Tolerance Overhead

This experiment used the top-k most visited pages of Wikipedia and was run 31 times [30]. ROME used a fan-in of 3 (see Figure 21). We ran the experiment on i3.large spot instances. Each of the 32 nodes contained a Spark worker and a ROME worker. The controller node ran Spark's cluster manager and ROME controller. We also deployed a Spark driver on a separate instance.

We first evaluated the overhead of providing fault tolerance in ROME, as shown in Figure 22. This was the only experiment that ran ROME without fault tolerance. Because this experiment was solely testing ROME internal functions, we show only the results with Spark. The mean execution time was around 44 seconds with and without fault tolerance. The standard deviation was around 2 seconds for both cases. This shows that ROME's fault tolerance has negligible impact.

To evaluate the recovery time under failure, we put a tier two ROME worker node along with its parent and one of its children on the same worker node and crashed that worker node at different stages of computation and before the second tier node finished its computation (black node in Figure 21). We observed that upon failure the latency increased by 50%.

7 RELATED WORK

The data analytics systems discussed in Section 2.3 are popular in part because of their power to adapt to current processing needs. Data analytics has quickly outgrown many early attempts at aggregation, which required special architecture, such as processors in tree networks and fully parallelizable functions [11, 29]. Aggregation is so fundamental to data analytics that the MapReduce framework has been modified to include aggregation more than once. Incoop [6] exploits `Combiners` to aggregate between the map and reduce phases, and Map-Reduce-Merge [58]

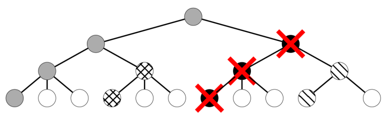


Fig. 21. Failure affecting multiple tree levels.

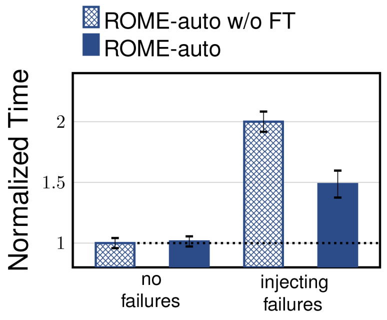


Fig. 22. ROME fault tolerance.

implements aggregation after the reduce phase. Yu et al. [59] also attempt to optimize aggregation between phases. All these approaches attempt to add the necessary aggregation functionality to MapReduce but do not consider the effect of topology.

Kumar et al. [32] consider the need for an aggregation overlay to respond with the most accurate result available within a deadline. The proposed solution determines the probability of a child node providing additional results within a given timeframe, then decides to wait or forward the current known results for the parent node to meet its deadline. This work is complementary to our focus on overlays.

Astrolabe [52] and STAR [26] use a holistic data management approach and optimize aggregation within their given hierarchies. Astrolabe uses gossip protocols for large-scale data propagation. STAR, extending the work of SDIMS [57] to build on top of distributed hash tables, has more flexible and configurable aggregation options but tries to use its topology rather than a prescribed fan-in. SDIMS allows the user to specify an aggregation precision with the understanding that very low latency is sometimes a priority over perfect answers. Kumar et al. [32] also consider the ability to selectively drop data during processing to minimize latency for applications that need very fast aggregation.

PIER [23] is another system built on DHTs to distribute workload, this time for database use. The overlays are once again restricted by the underlying framework, but the system itself efficiently aggregates results to respond to queries.

While not specific to aggregation, Kim et al. [29] extend the work by Cheng and Robertazzi [11] to optimize load distribution on processors connected by a tree network. The newer work maximizes parallelization for fastest completion, because there is no computation to aggregate results from each processor. Work with sensors optimizes overlays for power consumption while conforming to the routing restrictions imposed by the location and communication capabilities of sensors [9, 48]. TinyDB [35] furthers this in determining when to sample, which is equivalent to local computation.

Valerio et al. [51] consider aggregation when servers span multiple administration domains that do not trust each other. Their auditing system detects if a server may bias the aggregation result when manipulating the aggregation overlay. We assume all nodes are within the same administration domain.

Morozov and Weber [40] consider *merge trees* for distributed computations. Their approach monitors data traits in different branches and recomputes a better tree, *sparsifying* the trees on different computations nodes to scale the computation. It does not optimize for aggregation functions or extrapolate to find optima.

Naiad [41] considers aggregation as part of iterative or cyclic computations. It allows distributed data access and updates to be interleaved and is not aggregation-specific. Like resilient distributed datasets put forth by Zaharia et al. for Spark [60], Naiad relies on data retained in memory. This offers latencies that can be orders of magnitude better than with disk accesses. This addresses the problem raised by Venkataraman et al. [53] that data access is a significant bottleneck for iterative calculations on many distributed frameworks.

While users may be able to use our heuristics to manually configure aggregation in in-memory distributed data analytics systems like Spark [60] and Presto [53], as mentioned, this approach may fail when datasets are yet unknown, which is commonly the case with non-initial computation phases.

One work that does explicitly consider topology is CamCube [1], which allows users with full environmental control to define neighbor nodes that bypass traditional network routing to minimize latency. Each machine is limited to six such neighbors, but the configuration has been shown to decrease latency in some jobs using a MapReduce style framework with built-in aggregation called Camdoop [14].

LightSaber [50] is a recent stream processing engine, optimized for window aggregation queries. Lightsaber focuses on single server setups with a large number of cores and a large amount of shared memory. In this context, several performance gains result from optimizations, such as thread pinning, NUMA-aware scheduling, static memory allocations, SIMD instructions, and multicore computations. In contrast, ROME is designed for large-scale distributed systems, where the network communication cost must be taken into account and where some low-level optimizations may not be easily exploitable.

Distributed machine learning systems need to move model parameters across different nodes to avoid divergence during the training phase, possibly incurring high communication costs. Blink [54] focuses on multi-GPU servers with high-bandwidth NVLink/NVSwitch inter-GPU communication: Given a topology of the allocated GPUs, it aims at the optimal communication rate by packing spanning trees and by optimizing the amount of data transferred across the GPU links. MLNet [36] tackles the parameter synchronization problem by proposing instead a customized communication layer, which can optimize the communication through an aggregation overlay; however, this is built using a fixed and user-defined fan-in. SwitchML [45] and ATP [33] take a step further by carrying out the aggregation phase directly, as an in-network service, on top of the programmable switches available in the network. Differently from SwitchML, ATP also considers the multi-tenant scenarios where multiple jobs run at the same time in the network. Since programmable switches come with hardware restrictions, PANAMA [20] proposes to run the in-network service on top of FPGA-based devices. Finally, KungFu [37] proposes to perform real-time monitoring of the distributed ML system to detect when the communication functions interfere with the training process: When the training throughput drops due to network contention, KungFu automatically adjusts the topology to reduce the use of contended network links. While these works involve data aggregation, they focus on specific application goals and target specific hardware setups.

Our initial intuition of using the R ratio for optimizing the execution of basic compute-aggregate tasks was presented at a workshop [17], however, without thorough consideration of dealing with real-life (discrete) settings or implementation experience. The formal aspects of the problem were studied subsequently [18], again in an idealized setting without considering implementation, further adding focus on the case of data streaming.

Finally, Chuprikov et al. [12, 13] investigate compute-aggregate problems in setups where network links have different capacities/costs and "hard" topological constraints (e.g., compute nodes may not aggregate) focusing on hardness of optimal distribution and lower bounds.

8 CONCLUSIONS

We present ROME to construct and maintain low-latency aggregation overlays. ROME chooses an overlay based on the ratio of aggregation output to one input—a ratio highly characteristic of performance of distributed overlay-based aggregation—which we call R. Our targeted reuse of nodes further decreases latency and resource requirements. We empirically show our overlays are nearly, if not actually, optimal.

We propose that R is easy to find in most practical scenarios, at least to the granularity of the ranges $R < 1$, $R = 1$, and $R \geq n$ identified by our heuristics. If such estimation is not possible, then the user can inform our system to compute R on-the-fly via partial aggregation with low overhead—9% in the worst case in our experiments.

We integrate ROME into Flink and Spark and validate the effects of the overlay in end-to-end distributed data analytics with real-world data and problems. ROME decreases latency by a factor of up to 3 over unmodified systems that do not use tree overlays for aggregation, and up to 21% and 16% over a Spark deployment using `treeReduce` with default and *manually tuned* configurations, respectively. When running iterative algorithms with many aggregation phases, ROME decreases total runtime by up to 50% over the fastest Spark configuration. These improvements are despite sharing the RAM allocation between two systems. Even with the reduced RAM available, ROME completes some applications Flink and Spark are unable to handle due to memory constraints.

In ongoing work, we are considering optimizing the computation and aggregation in tandem instead of considering aggregation in isolation (e.g., calculating an optimal number of leaf nodes and specific optimizations for iterative computation), supporting streaming-type aggregation (cf. Reference [18]), and the addition of an additional phase exploiting the ability to perform basic computations such as aggregation directly on switches [22, 38, 44], and (smart) network interface controllers.

Another interesting aspect is how ROME could cope with stragglers, i.e., tasks with extremely large running times that may slow down the entire job computation when their results are needed for computing the final job output. With ROME, stragglers may emerge within the producers or the workers. In both cases, ROME could monitor the running times of the producers or the workers and, in case of significant time skewness, it may choose to dynamically reconfigure the overlay: Non-straggler workers should continue to aggregate using a near-optimal overlay, while straggler nodes could send their output directly to the root node of the aggregation overlay. In other words, ROME could use a strategy that is similar to the one it adopts in case of incremental updates (Section 3.4.4). This strategy should minimize the aggregation latency, assuming that the stragglers are restricted to a limited number of tasks.

REFERENCES

[1] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. 2010. Symbiotic routing in future data centers. In *SIGCOMM*.

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The stratosphere platform for big data analytics. *VLDB J.* 23, 6 (2014), 939–964.

[3] Apache Software Foundation. 2021. Flink. Retrieved from http://flink.apache.org.

[4] Apache Software Foundation. 2021. Spark. Retrieved from http://spark.apache.org.

[5] Francine Berman, Rob A. Rutenbar, Brent Hailpern, Henrik Christensen, Susan Davidson, Deborah Estrin, Michael J. Franklin, Margaret Martonosi, Padma Raghavan, Victoria Stodden, and Alexander S. Szalay. 2018. Realizing the potential of data science. *Commun. ACM* 61, 4 (2018), 67–72. DOI:https://doi.org/10.1145/3188721

[6] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for incremental computations. In *SOCC*.

[7] S. Borzsony, D. Kossmann, and K. Stocker. 2001. The Skyline operator. In *ICDE*.
[8] P. Cao and Z. Wang. 2004. Efficient Top-K query calculation in distributed networks. In *PODC*.
[9] Jae-Hwan Chang and L. Tassiulas. 2000. Energy conserving routing in wireless ad-hoc networks. In *INFOCOM*. DOI:https://doi.org/10.1109/INFCOM.2000.832170
[10] Yanpei Chen, A. Ganapathi, R. Griffith, and R. Katz. 2011. The case for evaluating MapReduce performance using workload suites. In *MASCOTS*.
[11] Y. C. Cheng and T. G. Robertazzi. 1990. Distributed computation for a tree network with communication delays. *IEEE Trans. Aerosp. Electron. Syst.* 26, 3 (1990), 511–516. DOI:https://doi.org/10.1109/7.106129
[12] Pavel Chuprikov, Alex Davydow, Kirill Kogan, Sergey I. Nikolenko, and Alexander Sirotkin. 2018. Formalizing compute-aggregate problems in cloud computing. In *Structural Information and Communication Complexity - 25th International Colloquium, SIROCCO 2018, Revised Selected Papers*. Springer, 377–391.
[13] Pavel Chuprikov, Alex Davydow, Kirill Kogan, Sergey I. Nikolenko, and Alexander V. Sirotkin. 2017. Planning in compute-aggregate problems as optimization problems on graphs. In *ICNP*. 1–2.
[14] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. 2012. Camdoop: Exploiting in-network aggregation for big data applications. In *NSDI*.
[15] National Research Council et al. 2013. *Frontiers in Massive Data Analysis*. National Academies Press.
[16] W. Culhane. 2015. *Optimal "Big Data" Aggregation Systems—From Theory to Practical Application*. Ph. D. Dissertation. Purdue University. Retrieved from https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1218&context=open_access_dissertations.
[17] William Culhane, Kirill Kogan, Chamikara Jayalath, and Patrick Eugster. 2014. LOOM: Optimal aggregation overlays for in-memory big data processing. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. USENIX Association, Philadelphia, PA.
[18] William Culhane, Kirill Kogan, Chamikara Jayalath, and Patrick Eugster. 2015. Optimal communication structures for big data aggregation. In *INFOCOM*.
[19] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. DOI:https://doi.org/10.1145/1327452.1327492
[20] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-network aggregation for shared machine learning clusters. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 829–844.
[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP*.
[22] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. 2016. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 1–10.
[23] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. 2003. Querying the Internet with PIER. In *29th International Conference on Very Large Data Bases - Volume 29 (VLDB'03)*. VLDB Endowment, 321–332. Retrieved from http://dl.acm.org/citation.cfm?id=1315451.1315480.
[24] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *ATC*.
[25] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *Comput. Surv.* 40, 4 (2008).
[26] Navendu Jain, Dmitry Kit, Prince Mahajan, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. 2007. STAR: Self-tuning aggregation for scalable monitoring. In *VLDB*.
[27] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-aware factorization machines for CTR prediction. In *RecSys*. ACM, 43–50.
[28] Huan Ke, Peng Li, Song Guo, and I. Stojmenovic. 2015. Aggregation on the fly: Reducing traffic for big data in the cloud. *IEEE Netw.* 29, 5 (Sept. 2015), 17–23. DOI:https://doi.org/10.1109/MNET.2015.7293300
[29] Hyoung-Joong Kim, Gyu-In Jee, and Jang-Gyu Lee. 1996. Optimal load distribution for tree network processors. *IEEE Trans. Aerosp. Electron. Syst.* 32, 2 (1996), 607–612. DOI:https://doi.org/10.1109/7.489505
[30] Pascal Kleber. 2017. *Fault Tolerance in Optimal Aggregation Overlays for Big Data Applications*. Master-Thesis. TU Darmstadt.
[31] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream processing at scale. In *SIGMOD*.
[32] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. 2016. Hold'em or fold'Em? Aggregation queries under performance variations. In *EuroSys*. Association for Computing Machinery, New York, NY. DOI:https://doi.org/10.1145/2901318.2901351

[33] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*. USENIX Association, 741–761.

[34] Y. Liu, Z. Hu, and K. Matsuzaki. 2011. Towards systematic parallel programming over MapReduce. In *Euro-Par*.

[35] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2005. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Datab. Syst.* 30, 1 (Mar. 2005), 122–173. DOI:https://doi.org/10.1145/1061318.1061322

[36] Luo Mai, Chuntao Hong, and Paolo Costa. 2015. Optimizing network performance in distributed machine learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, Santa Clara, CA.

[37] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, 937–954.

[38] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. 2014. NetAgg: Using middleboxes for application-specific on-path aggregation in data centres. In *CoNext*.

[39] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. 2009. The third homomorphism theorem on trees: Downward & upward lead to divide-and-conquer. In *POPL*.

[40] Dmitriy Morozov and Gunther Weber. 2013. Distributed merge trees. In *PPoPP*.

[41] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 439–455. DOI:https://doi.org/10.1145/2517349.2522738

[42] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. 2018. Big data technologies: A survey. *J. King Saud Univ.-Comput. Inf. Sci.* 30, 4 (2018), 431–448.

[43] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Conference on Advances in Neural Information Processing Systems*. 693–701.

[44] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *ACM HotNets*.

[45] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*. USENIX Association, 785–808.

[46] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. 2010. The Hadoop distributed file system. In *MSST*.

[47] SIGKDD. 2012. KDD CUP 2012 Data Set. Retrieved from https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#kdd2012.

[48] Hüseyin Özgür Tan and Ibrahim Körpeoğlu. 2003. Power efficient data gathering and aggregation in wireless sensor networks. *SIGMOD Rec.* 32, 4 (Dec. 2003), 66–71. DOI:https://doi.org/10.1145/959060.959072

[49] The Computational Biology and Functional Genomics Laboratory at DFCI/Harward. 2015. TGI Database: DNA Sequences. Retrieved from http://compbio.dfci.harvard.edu/tgi/.

[50] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient window aggregation on multi-core processors. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. Association for Computing Machinery, New York, NY, 2505–2521. DOI:https://doi.org/10.1145/3318464.3389753

[51] José Valerio, Pascal Felber, Martin Rajman, and Etienne Riviere. 2012. CADA: Collaborative auditing for distributed aggregation. In *9th European Dependable Computing Conference*. 1–12.

[52] Robbert Van Renesse, Kenneth Birman, Dan Dumitriu, and Werner Vogels. 2002. Scalable management and data mining using Astrolabe. In *Peer-to-Peer Systems*. Springer, 280–294.

[53] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. 2013. Presto: Distributed machine learning and graph processing with sparse matrices. In *EuroSys*.

[54] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ML. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 172–186.

[55] Wikipedia. 2016. Pageviews Hourly Statistics Dumps. Retrieved from https://wikitech.wikimedia.org/wiki/Analytics/Data/Pagecounts-raw.

[56] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. 2013. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM Trans. Netw.* 21, 2 (Apr. 2013), 345–358.

[57] Praveen Yalagandula and Mike Dahlin. 2004. SDIMS: A scalable distributed information management system. In *SIGCOMM*.

[58] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 2007. Map-reduce-merge: Simplified relational data processing on large clusters. In *SIGMOD*.

[59] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP*.
[60] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.
[61] Yan Zhang and Nirwan Ansari. 2013. On architecture design, congestion notification, TCP incast and power consumption in data centers. *IEEE Commun. Surv. Tutor.* 15, 1 (2013), 39–64.

Received March 2020; revised October 2021; accepted January 2022