LOOM: Optimal Aggregation Overlays for In-Memory Big Data Processing*

William Culhane Kirill Kogan Chamikara Jayalath Patrick Eugster
Department of Computer Science, Purdue University

Abstract

Aggregation underlies the distillation of information from big data. Many well-known basic operations including top-*k* matching and word count hinge on fast aggregation across large data-sets. Common frameworks including MapReduce support aggregation, but do not explicitly consider or optimize it. Optimizing aggregation however becomes yet more relevant in recent "online" approaches to expressive big data analysis which store data in main memory across nodes. This shifts the bottlenecks from disk I/O to distributed computation and network communication and significantly increases the impact of aggregation time on total job completion time.

This paper presents LOOM, a (sub)system for efficient big data aggregation for use within big data analysis frameworks. LOOM efficiently supports two-phased (sub)computations consisting in a first phase performed on individual data sub-sets (e.g., word count, top-*k* matching) followed by a second aggregation phase which consolidates individual results of the first phase (e.g., count sum, top-*k*). Using characteristics of an aggregation function, LOOM constructs a specifically configured aggregation overlay to minimize aggregation costs. We present optimality heuristics and experimentally demonstrate the benefits of thus optimizing aggregation overlays using microbenchmarks and real world examples.

1 Introduction

Processing big data is a core challenge of our era. Extracting and distilling more concise information from large datasets is vital to scientific research (e.g., astrophysics, chemistry), the commercial sector (e.g., advertising, recommendation), and governmental institutions (e.g., intelligence services, population demographics).

Aggregation of information — broadly construed — inherently underlies the distillation process. In commonly employed toolkits for parallelized big data batch processing including MapReduce [5], aggregation happens straightforwardly. In the popular word count example, mappers produce intermediate key-value pairs where keys represent words and values their number of occurrences in a slice of input data – in a common simple approach mappers are executed for individual words, and this value is trivially 1. Aggregation happens when reducers sum all values with the same key, i.e., word.

As demonstrated by Yu, Gunda, and Isard [15], performance can be significantly improved by considering partial word counts computed for larger subsets of data, and subsequently aggregating these. The intuition is that some "functions" such as word-count can be performed in a *distributed associative* [15] manner by expressing them as two parts – a first one which can be performed on data subsets individually, and a second one aggregating the results of the first stage, possibly in several steps.

The relevance of *how exactly* to perform such aggregation is addressed by recent work which stores datasets in main memory to increase expressiveness of computations for instance by supporting iterative or incremental computations, somewhat departing from pipelined batch processing towards more online processing (e.g., distributed arrays in Presto [11, 10], resilient distributed datasets in Shark [16, 12]). By striping a dataset across the main memory of a large number of *n* nodes, heavy disk I/O is avoided between subsequent computation phases, within iterations of a phase, upon incremental computations triggered by updates, or upon continuous, interactive computations. By bypassing disk I/O, performance can be improved by an order of magnitude, yet the bottlenck is shifted to communication and aggregation.

If the function used for aggregation is cumulative (so results are the combination of results on subsets), commutative (so subsets can be aggregated in any order), and associative (so subsets can be aggregated in any grouping), there is some configurability in the structure of an overlay network along which such aggregation can take place: aggregation can happen in one step aggregating all sub-results on one node, between two sub-results at a time over $log_2 n$ steps, or something in between.

This paper thus presents Lightweight Optimal Overlay Mechanism (LOOM), a system to efficiently determine and implement provably optimal aggregation overlays to weave together results in *compute-aggregate* operations. Based on traits of the aggregation function and the *n* nodes taking part, LOOM *automatically* heuristically determines the provably (near-)ideal fanout for an aggregation tree consolidating sub-results from these nodes after they performed the initial computation phase individu-

*Financially supported by DARPA grant # N11AP20014, Purdue Research Foundation grant # 205434, and Google Award "Geo-Distributed Big Data Processing".

ally. Depending on the parameters, performance between the overlay identified by LOOM and a naïve one can vary by more than 600% in our experiments, and we expect greater variation in more extreme environments.

Our contributions are as follows. After introducing the model of compute-aggregate tasks considered we

1. present provably optimal heuristics for determining the fanout of an aggregation tree given the knowledge of the aggregation method (Section 2).
2. discuss the architecture of LOOM, a system that uses the heuristics to create optimal aggregation trees in the case of well-defined or sampled aggregation functions used in compute-aggregate problems (Section 3).
3. empirically show via microbenchmarks and some typical compute-aggregate tasks that the overlay determined by LOOM matches the ideal case in practice and leads to significant time savings (Section 4).

Section 5 discusses prior art. We draw conclusions in Section 6.

2 Model

Compute-aggregate is a two-phase class of problems (see Figure 1). The first computation phase happens only at the leaf nodes. At each such node, data z is contained in memory prior to computation, and computation applies some function f such that $x = f(z)$, where x is data ready for aggregation. Once the computation phase is completed the aggregation overlay is responsible for aggregating the partial results from all leaves into a single result. This is done by applying a function g which takes $x_1 \ldots x_i$ and outputs the aggregate $x^{1..i}$, i.e., $x^{1..i} = g(\bar{x})$. Table 1 defines more precisely the required traits of an aggregation function which allows for configurable overlays. Note that we say results are equivalent ($\equiv$) rather than equal. For example, if you want the top-k results and there are $k+1$ results with the same score, different aggregation orders may result in different results, but each is correct.

Each node applies the function to all of its inputs and forwards its results. As long as the function is cumulative commutative, and associative, any overlay network that has exactly one path from each leaf to the root of the aggregation tree gives a correct output.

With everything in memory the time to aggregate at each node is mostly dependent upon the size of its combined input. The most efficient system is one with no

Table 1: Mathematical definitions of requirements for $g(\bar{x})$.

Property	Definition
Cumulative	$g(g(\bar{x}), g(\bar{x}')) \equiv g(\bar{x}, \bar{x}')$
Commutative	$g(\bar{x}', \bar{x}) \equiv g(\bar{x}, \bar{x}')$
Associative	$g(g(\bar{x}), \bar{x}') \equiv g(\bar{x}, \bar{x}')$

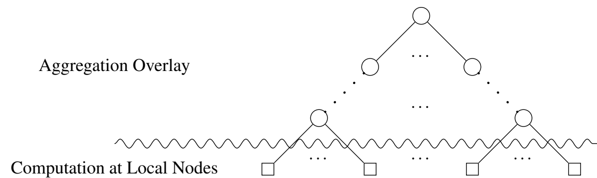


Figure 1: Computation and aggregation phases.

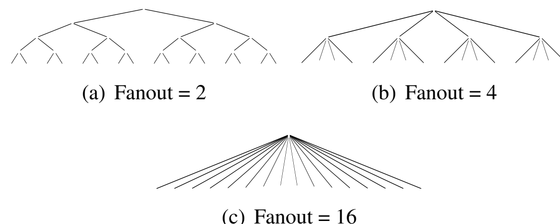


Figure 2: Three aggregation overlays with 16 leaves.

waiting at any nodes that have work to do, so we assume a balanced and full tree; sibling aggregation nodes begin and finish at roughly the same time, with a bit of a buffer provided by the nature of network communication. This does require some level of homogeneity, which is not unreasonable to expect in cloud services. If any assumptions are broken, the optimality guarantee breaks, but the correctness of the output remains.

Our objective is to minimize the total latency of the system. Since the aggregation phase is separate from the local computation time, we can optimize them separately. The aggregation overlay simply aggregates the outputs from all the computation nodes, so the number of leaf nodes for the tree defining the aggregation overlay is given by the number of local nodes involved in the computation phase. The functions to aggregate multiple inputs into a single input are also given by the problem, so the only variable left to change is the *fanout*. Figure 2 shows three equivalent trees with different fanouts.

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 $\bar{x}$. We use $g^c(\bar{x})$ to denote the *time required* by $g(\bar{x})$ to aggregate input of size $|\bar{x}|$, including communication time. Aggregation on the same level of the overlay happens in parallel, so only the time of a single branch is modeled.

Fanout and tree height are inversely related. Increased height increases amount of work done in parallel at the lower levels, but it also might increase the total amount of work to be done when results have to filter up through more levels. We analyze traits of the aggregation function to determine if the time saved by the parallelism offsets the time required by extra levels.

Table 2 summarizes our optimal heuristics for choosing the fanout of the aggregation overlay tree and notes

Table 2: Heuristic values for fanout. ✓ means provably optimal, * indicates provably near-optimal result.

y_0	Optimal fanout	Sublin. $g^c(x)$	Linear $g^c(x)$	Superlin. $g^c(x)$
$y_0 < 1$	2		✓	✓
$y_0 = 1$	e		✓	*
$1 < y_0 < n$	$min\left(n, (1-\log_n y_0)^{-\log_{y_0} n}\right)$		✓	
$n \leq y_0$	n	✓	✓	✓

the cases for which they are proved. Two variables in the table represent characteristics of the aggregation. $g^c(\overline{x})$ is one. The other is the ratio of the size of the final aggregate output to the size of the output of a single computation node, denoted y_0. n is simply the number of leaves.

Optimal fanouts remain unproven when the degree of sublinearity or superlinearity is necessary for meaningful analysis. Communication time is linear with respect to the input size, so there is always linear component to $g^c(\overline{x})$. Thus it makes sense to use the heuristics from the linear cases on the sublinear cases. In practice we also use the heuristics from the linear cases on the superlinear cases as they are the best available analyses.

y_0 is a very relevant variable to aggregation. As seen in Table 2, it is the primary factor for deciding the optimal fanout of the aggregation overlay. To give an idea what types of problems fall into each category we present Table 3. It shows some aggregation functions which fall into some of the different regions for y_0.

Table 3: Typical aggregation functions g grouped by y_0.

y_0	Common problems
$y_0 < 1$	The average MapReduce job at Google [5], the average "aggregate" jobs at Facebook and Yahoo [3]
$y_0 = 1$	Min, max, average, top-k match, word count with a fixed dictionary, multiplication of square matrices
$y_0 > 1$	sort, concatenate, word count with disjoint dictionaries

3 Implementation

Architecture. LOOM uses the heuristics defined previously to form optimal hierarchies for compute-aggregate problems. Figure 3 overviews its architecture. Computation nodes are managed by a big data analysis system (e.g., Apache Crunch [1] or Spark [16]) which uses LOOM. When a compute-aggregate problem is initialized, computation nodes start the first phase. A list of the computation nodes and the traits of the aggregation function are sent to the Controller within LOOM. After this LOOM determines the optimal fanout and determines the appropriate overlay on an ordered set of vertices. The controller tells each vertex its children and parent. All nodes create the relevant TCP connections while waiting for the computation phase to complete.

When a computation node finishes, it sends its results to the appropriate vertex. Each aggregator waits until it has results from all its children, then aggregates and pushes the results to its parent. When the root finishes it can hand the result over to the big data analysis system or write it out to a distributed file system. If the overlay is created for ongoing computation then the vertices maintain their connections and do work when new input is pushed in. Otherwise a node drops connections after communicating its portion to its parent.

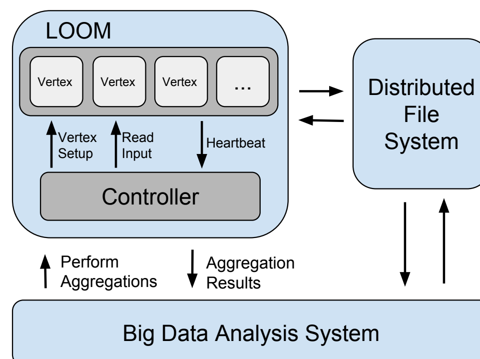


Figure 3: Architecture

Programming abstractions. Next we outline a minor API extension to FlumeJava [2] implemented in Apache Crunch [1] for users to directly, more easily, define compute-aggregate operations. FlumeJava is a Java library for dealing with collections that represent possibly large amounts of data, and expressing operations that should be performed on these collections. The two main data structures of FlumeJava are `PCollection` and `PTable`, representing a collection of elements and a collection of *key-value* pairs respectively. FlumeJava proposes four main types of operations that can be performed on its data structures: (1) `parallelDo()` performs a general operation defined by a function object (`DoFn`) on each element; (2) `groupByKey()` groups elements of a `PTable` based on keys; (3) `combineVales()` combines elements of a grouped `PTable` that has the same key based on a function object of type `CombineFn`; (4) `flatten()` flattens elements of two or more `PCollection`s resulting in a single `PCollection`.

We extend FlumeJava with a fifth main operation `parallelAggregate()`. The syntax of the operation performed on a `PCollection<A>` is given below.

```
PCollection<B> parallelAggregate(DoFn<A,B>,
        AggregateFn<B>)
```

`parallelAggregate()` takes two parameters: (1) a compute function of type `DoFn<A,B>` and (2) an aggregation function of type `AggregateFn<B>`. The result of the operation is a `PCollection<B>` that represents the aggregated data. The following example illustrates how

`parallelAggregate()` can be used to concisely determine the top-k elements of a dataset.

```
PCollection<String> originalElements = ...;
PCollection<Integer> topK =
  originalElements.parallelAggregate(
    computeFn, aggregateFn);
```

For brevity we elide the computation function `computeFn` used to generate `Integer` elements from a `String` dataset and the aggregation function `aggregateFn` used to determine the top-k elements of the set of `Integers`. The operation returns a `PCollection` that contains the top-k elements.

4 Experiments

Setup. We ran our experiments on m1.small nodes in an Amazon EC2 datacenter. Each leaf and vertex is on a distinct node. We show fanouts resulting in full trees to avoid the noise of straying from the model, but our system handles all fanouts. Averages of 5 runs are shown.

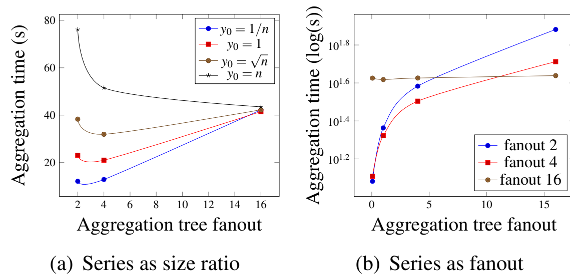
We first test the system with microbenchmarks varying y_0 with linear aggregation methods. Leaf nodes generate a random list of integers. Aggregators generate a series of random numbers proportional to the size of their inputs, then prune down the list to the size dictated by y_0. All microbenchmarks are performed on 16 leaf nodes.

Then we evaluate the system on two common aggregation tasks using a dataset that consists of log files of Yahoo's Hadoop clusters. Each leaf node contains 830MB of input data, which is in the range of what is studied in RDDs [16]. We perform on both 16 and 64 leaves:

- Top-k match – a simple equation "scores" how well each line of the log matches a filter. The k lines with the highest score are returned in sorted order from each leaf. Aggregator nodes forward the k highest scores across their inputs. We use $k = 100000$.
- Word count – counts the occurrences of each word in the log files at the leaves and sums the results. The final aggregate result is a map of each word in the logs and the number of times it appeared across all logs. Log entries are not very disparate, so any word appearing in the log at one leaf has a high probability of appearing in the logs at every other leaf, so if y_0 is greater than 1, it is negligibly so.

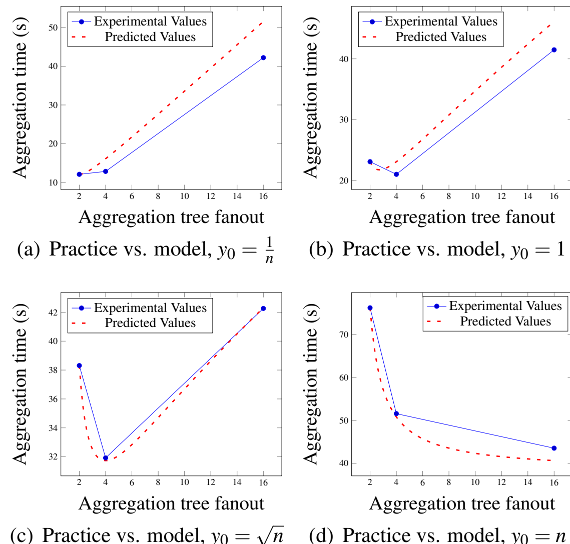
For all tests the leaves complete their computation and inform the controller. The controller then starts a timer, sends a "go" signal to begin aggregation and stops the timer when it receives the final aggregate result from the root of the overlay. We use the value from $d = 2$ from most experiments and the model for the associated value of y_0 to create the "Predicted Values" lines.

Results. Figure 4 shows the results of the microbenchmarks. In Figure 4(a) each line represents the data grouped by y_0. Figure 4(b) draws the same data, but each line represents an aggregation tree with a given fanout.When y_0 is small, the smaller fanouts outperform. As y_0 grows, the performance of those overlays degrades until a larger fanout is faster.



(a) Series as size ratio (b) Series as fanout

Figure 4: Microbenchmark results



(a) Practice vs. model, $y_0 = \frac{1}{n}$ (b) Practice vs. model, $y_0 = 1$

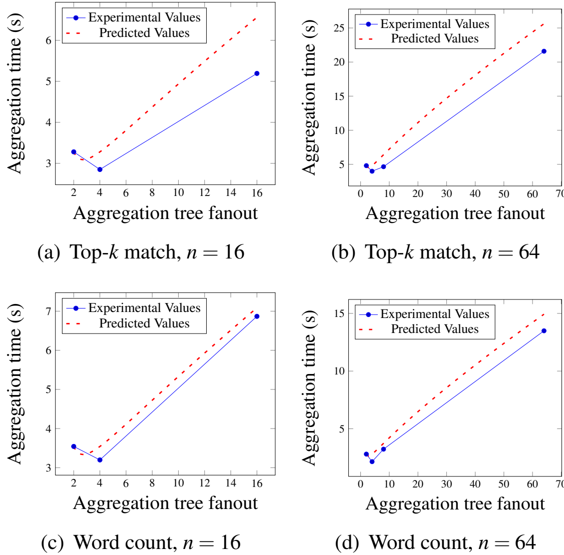
(c) Practice vs. model, $y_0 = \sqrt{n}$ (d) Practice vs. model, $y_0 = n$

Figure 5: Microbenchmarks versus model

Figure 5 show the performance of each y_0 for varying fanouts versus that predicted by the model. In all four cases the trends match. The only minimum not at the predicted place is for $y_0 = 1$. Fanout 4 slightly outperforms fanout 2 when the model predicts identical performance.

For $y_0 = \frac{1}{n}$, the time taken for $d = 16$ is 350% what it is for $d = 2$. For $y_0 = n$, the time for $d = 2$ is 175% what it is for $d = 16$. That is a very significant penalty for choosing the wrong fanout, and the right fanouts are opposite in these two examples. Even choosing between 2 and n is not a good heuristic for all cases, as the faster of the two still takes 132% of the time as $d = 4$ for $y = \sqrt{n}$.

Figure 6 shows the results from our common aggregation problems on real world data. For the 16 leaf top-k matching experiment in Figure 6(a) the values seem to diverge from the predicted values as d grows, but the trend matches the model and the microbenchmarks. The deviation may be because the aggregation function de-



(a) Top-k match, $n = 16$ (b) Top-k match, $n = 64$

(c) Word count, $n = 16$ (d) Word count, $n = 64$

Figure 6: Common problem results

pends on d and input size ($O(d\,|x_0|\log d)$) and not just the input size. At increased scale with 64 leaves, as seen in Figure 6(b), the divergence becomes overshadowed.

Figures 6(c) and 6(d) show the results from the word count test for 16 and 64 leaves respectively. They show the same trends as the microbenchmarks and top-k matching experiments and deviate even less from the predicted values. This is likely because the sets of words at each node were the same, so summing occurrences is an aggregation function truly linear on the size of the input.

Predictably, the relative impact of choosing the right fanout increases as more leaves are added. For the top-k matching and word count applications with 16 nodes the worst case fanouts take 182% and 215% as much time as their best case fanouts respectively. When n increases to 64 those numbers jump to 542% and 629% respectively.

5 Related Work

Big data aggregation. The idea of distinct phases in problems is not new. The popular MapReduce [5] framework requires a problem to be broken into two phases as suggested by its name. The application of MapReduce is so enticing that its Hadoop implementation has been modified to fit some aggregation problems which cannot be recast efficiently as strict MapReduce problems. MapReduceMerge [14] extends the framework with a step called merge – a synonym for aggregation. MapReduceMerge is motivated by many operations relevant to applications using MapReduce, e.g. full sorts, joins, set unions, and cartesian products, which are not being supported efficiently. Yu et al. [15] extend MapReduce to efficiently aggregate data in between phases of jobs.

Optimizing aggregation. Astrolabe [9] is introduced as a summarizing mechanism of bounded size using hierarchical overlays. Gossip protocols use aggregation to monitor system state and promote dynamic scalability. The overlay's impact on the aggregation is considered, but minimal aggregation time is not the primary concern.

STAR [6] extends a line of research started by SDIMS [13] creating information management systems on top of distributed hash tables. STAR adaptively sets the precision constraints for processing aggregation.

Cheng and Robertazzi [4] tackle the problem of optimal load distribution on processors connected by a tree network. Their work and the extension by Kim et al [7] consider balancing computation for problems in which the greatest parallelization leads to the fastest completion because there is no computation to aggregate results from each processor. Morozov and Weber [8] consider distributed computations resulting in merge trees, an abstraction for combining subsets of large structured datasets. Their system monitors data attributes in different branches and recomputes a more optimal tree.

These works do not optimize for aggregation or use extrapolation to find (a potentially local) optimum.

6 Conclusions and Future Work

We present a system constructing optimal aggregation overlays for a relevant class of problems which includes a time-sensitive aggregation phase. We deduce that improving the time for aggregation significantly improves the total system latency, and this is doable by adjusting the fanout of the aggregation overlay.

We experimentally show our optima to be very close to the values seen in practice using targeted microbenchmarks and common aggregation functions on real world data. On the real world data we saw a performance difference in excess of 600% between optimal and non-optimal fanouts, and we expect the difference to grow as more leaves are added. Especially since the aggregation phase takes a greater portion of the total system time as work is parallelized across a greater number of leaves, this represents a significant potential for optimization.

Our heuristics rely only upon the number of leaves and the ratio of the output size of the aggregation to the input size, which are provided upfront. There are still a handful of cases for which the optima remain unproven based on the execution time of the aggregation relative to input size. Proving these cases will result in two relevant variables which could be determined through experimentation and interpolation without the need for human intervention. This will be harder for cases when the growth factor changes between levels, but that may provide more interesting results including a dynamic fanout.

In addition, we are currently investigating optimizations for incremental and iterative computation.

References

[1] Apache Software Foundation. Incubator Crunch. `http://crunch.apache.org/`.

[2] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *PLDI* (2010).

[3] Chen, Y., Ganapathi, A., Griffith, R., and Katz, R. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS* (2011).

[4] Cheng, Y., and Robertazzi, T. Distributed Computation for a Tree Network with Communication Delays. *IEEE Transactions on Aerospace and Electronic Systems 26*, 3 (1990), 511–516.

[5] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[6] Jain, N., Kit, D., Mahajan, P., Yalagandula, P., Dahlin, M., and Zhang, Y. STAR: Self-tuning Aggregation for Scalable Monitoring. In *VLDB* (2007).

[7] Kim, H.-J., Jee, G.-I., and Lee, J.-G. Optimal Load Distribution for Tree Network Processors. *IEEE Transactions on Aerospace and Electronic Systems 32*, 2 (1996), 607–612.

[8] Morozov, D., and Weber, G. Distributed Merge Trees. In *PPoPP* (2013).

[9] Van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Trans. Comput. Syst. 21*, 2 (May 2003), 164–206.

[10] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A., and Schreiber, R. S. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. In *EuroSys* (2013).

[11] Venkataraman, S., Roy, I., AuYoung, A., and Schreiber, R. Using R for Iterative and Incremental Processing. In *HotClouds* (2012).

[12] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. Shark: SQL and Rich Analytics at Scale. In *SIGMOD* (2013).

[13] Yalagandula, P., and Dahlin, M. SDIMS: A Scalable Distributed Information Management System. In *SIGCOMM* (2004).

[14] Yang, H.-c., Dasdan, A., Hsiao, R.-L., and Parker, D. S. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD* (2007).

[15] Yu, Y., Gunda, P. K., and Isard, M. Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations. In *SOSP* (2009).

[16] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing. In *NSDI* (2012).