Plumb: Efficient Processing of Multi-Users Pipelines (Extended)

USC/ISI Technical Report ISI-TR-727, Oct 2018

Abdul Qadeer and John Heidemann
University of Southern California, Information Sciences Institute
email:{*aqadeer, johnh*}*@isi.edu*

Abstract

Services such as DNS and websites often produce streams of data that are consumed by analytics pipelines operated by multiple teams. Often this data is processed in large chunks (megabytes) to allow analysis of a block of time or to amortize costs. Such pipelines pose two problems: first, duplication of computation and storage may occur when parts of the pipeline are operated by different groups. Second, processing can be *lumpy*, with *structural lumpiness* occurring when different stages need different amounts of resources, and *data lumpiness* occurring when a block of input requires increased resources. Duplication and structural lumpiness both can result in inefficient processing. Data lumpiness can cause pipeline failure or deadlock, for example if differences in DDoS traffic compared to normal can require $6\times$ CPU. We propose *Plumb*, a framework to abstract file processing for a multi-stage pipeline. Plumb integrates pipelines contributed by multiple users, detecting and eliminating duplication of computation and intermediate storage. It tracks and adjusts computation of each stage, accommodating both structural and data lumpiness. We exercise Plumb with the processing pipeline for B-Root DNS traffic, where it will replace a hand-tuned system to provide one third the original latency by utilizing 22% fewer CPU and will address limitations that occur as multiple users process data and when DDoS traffic causes huge shifts in performance.

1 Introduction

Services such as DNS and websites often produce streams of data that are consumed by analytics pipelines operated by multiple teams. Often this data is processed in large blocks or files that are megabytes in size, so that analytics can examine a range of time, amortizing processing costs, and chunk data for fault detection and recovery. These blocks of data are processed in pipelines of many steps, each transforming, indexing, or merging the data. For long-lived systems, the analysis grows over time, and different groups may be responsible for different portions of the pipeline.

Such pipelines pose two problems: *unnecessary duplication* of computation and storage, and *lumpiness* as different stages or files require different amounts of resources. Duplication and lumpiness reduce processing efficiency, increasing hardware costs and delaying results.

Unnecessary duplication of work can result as the pipeline is extended by multiple parties. In a large pipeline, data is often transformed into different formats, compressed, and decompressed. When different users are responsible for different portions of the pipeline, these stages may be duplicated if each user assumes they begin with raw input, or duplication may arise over time as portions of the pipeline evolve.

The second inefficiency is lumpiness, or uneven consumption of resources across the pipeline. We consider two kinds of lumpiness: *structural lumpiness*, when different stages need different amounts of resources; and *data lumpiness*, when a particular block of input requires increased resources (compared to typical input blocks).

Structural lumpiness is a problem in multi-stage pipelines. Consider an n-stage pipeline: it can either be scheduled to run concurrently using n cores, or each stage can run independently. With concurrent processing, any stage that consumes more or less than one core will unbalance the pipeline and leave other stages idling, wasting resources. If stages are scheduled independently, stages with different computation requirements may require more (or fewer) concurrently executing instances, and all stages must pay the overhead of buffering intermediate output, perhaps through the file system.

Data lumpiness is similar, but is triggered when a particular input increases the computation required at one or more stages. If computation increases somewhat, inefficiencies similar to structural lumpiness may arise. If computation increases significantly, processing timeouts may occur, making progress impossible. As an example of data lumpiness, consider a pipeline tuned for regular DNS processing that is then faced with a TCP flooding attack, or a spoof flooding attack that triggers use of TCP. In both cases, nearly all arriving flows are TCP, compared to a few percent in normal traffic, and each TCP flow requires reassembly, stressing memory and computation (search) in processing and pushing CPU to 6× normal or more.

In this paper we propose *Plumb*, a framework to abstract file processing for a multi-stage pipeline. Plumb integrates pipelines contributed by multiple users, detecting and eliminating duplication of computation and intermediate storage. It tracks and adjusts computation of each stage, creating more processing instances as required to accommodate both structural and data lumpiness.

We exercise Plumb with the processing pipeline for B-Root DNS traffic. Compared to the currently operational, hand tuned system, we expect Plumb to provide one-third the latency while utilizing 22% less CPU. Moreover, the Plumb abstractions enable multiple users to contribute to processing with minimal coordination, and it keep latency low during normal conditions, while adapting to cope with dramatic changes to traffic and processing requirements when handling denial-of-service attacks.

The contributions of this paper are to define the challenges of structural and data lumpiness in large-file, multi-user, streaming workloads. We then describe the Plumb architecture (§2), where each user specifies their computation with a pipeline graph (Figure 2). Plumb uses this description to integrate computation from multiple users while avoid duplicate computation (§2.3), and output (§2.4). To balance lumpiness, it supports identification of IO-bound stages (§2.5), and dynamically schedules stages to avoid structural (§2.6) and data lumpiness (§2.7). We evaluate each of these design choices in §3, using controlled experiments tested with components of the B-Root processing workload (§2.1).

2 System Design

We next describe Plumb, and its requirements, and specific optimizations to de-duplicate storage and processing, and to mitigate processing lumpiness.

2.1 Design Requirements and Case Study

Our system is designed to provided *large-file* streaming for *multiple users* while being *easy-to-use*.

Large-file streaming means data constantly arrives at the system, and it is delivered in relatively large blocks. Data is streaming because it is continuously collected. Unlike other streaming systems that emphasize small events (perhaps single records), we move data large blocks—from 512 MB to 2 GB in different deployments. Large files are important for some applications where computations frequently span multiple records. Large files also amortize processing costs and simplify detection of completeness (§2.2) and error recovery (§2.7).

For our sample application of DNS processing, large files are motivated by the need to do TCP reassembly, which benefits from having all packets for a TCP connection usually in the same file. Compression is much more efficient on bulk data (many KB). We find error handling (such as disk space exhaustion or data-specific bugs) and verification of completeness is easier when handling large, discrete chunks of data.

Streaming also implies that we must keep up with real-time data input over the long term. Fortunately, we can buffer and queue data on disk. At times (after a processing error) we have queued almost two weeks of data, requiring more than a week to drain.

Multiple users implies that different individuals or groups contribution components to the pipeline over time. This requirement affects our choice of processing definition in §2.4 and supports de-duplication of computation (§2.3).

Finally, *ease-of-use* is an explicit design goal. As with map-reduce [10], analysis are presented with a simple pipeline model and the framework handles parallelism. Inspired by Storm [40], we adapt parallelism of each pipeline stage to match the workload lumpiness (§2.7).

DNS Processing as a Case Study: These requirements are driven by our case-study: the B-Root DNS processing pipeline. Figure 1 shows the user-level view of this workflow: three different output files (the ovals at the bottom of the figure), include archival data (`pcap.xz`), statistics (`rssacint`), and processed data (`message_question.fsdb.sz`). Generating this output logically requires five steps (the squares), each of which have very different requirements for I/O and computation (shown later as Table 1). This pipeline has been in use for 2.5 years and takes as input 1.5 to 2 TB of data per day. We would like to extend it with additional steps, and migration of the the current hand-crafted code to Plumb is underway.

2.2 Plumb Overview

We next briefly describe workflow in Plumb to provide context for the optimizations described in the following sections. Figure 4 shows overall Plumb workflow.

Users provide (step 1 in Figure 4) Plumb their workloads as with a YAML-based pipeline specification (Figure 2). Plumb integrates workloads from multiple users and can provide both graphical (Figure 3) and textual (Figure 2) descriptions of the integrated graph. Plumb detects and optimizes away duplicate stages from multiple users (§2.3). Data access from each user is protected by proper authentication and authorization, mediated by a database of all available content (tag 5).

User's pipeline stage programs need to adhere to Plumb interaction rules. Programs get input and output via stdin and stdout respectively if there are just one input and one output. Otherwise, system provides inputs via multiple -0 and output via multiple -1 command line arguments. User stage program is expected to be a serial code, that streams

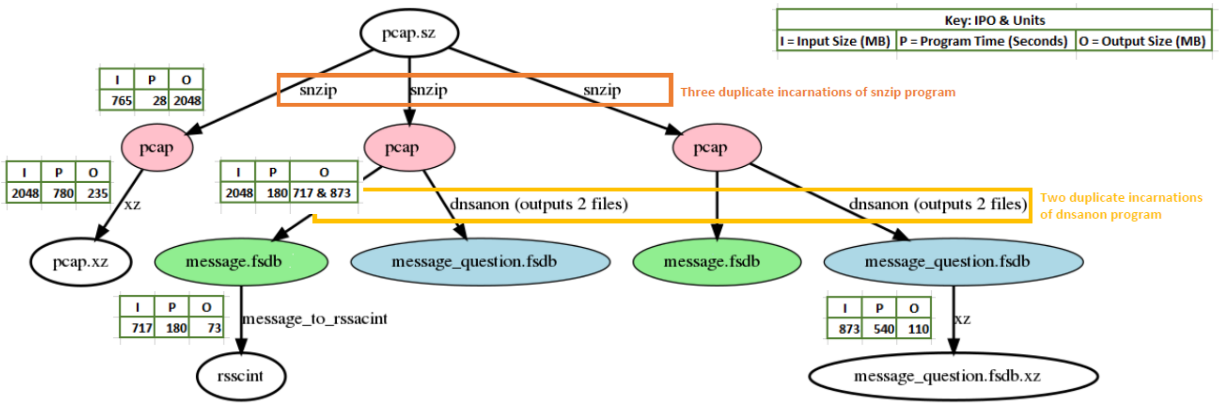

Figure 1: The DNS processing pipeline, our case study described in §2.1. Intermediate and final data are ovals, computation occurs on each arc, with duplicate computation across users in rectangles.

```
# Third user's pipeline
-
  input:   message_question.fsdb
  program: "/usr/bin/xz -c"
  output:  message_question.fsdb.xz

# Alternate representation of third user's pipeline
-
  input:   pcap.sz
  program: "/usr/bin/snzip -c -d"
  output:  pcap
-
  input:   pcap
  program: "/usr/bin/dnsanon -p mQ"
  output:  [ message_question.fsdb, message.fsdb ]
-
  input:   message_question.fsdb
  program: "/usr/bin/xz -c"
  output:  message_question.fsdb.xz
```

Figure 2: A Pipeline Graph for a portion of the DNS pipeline.

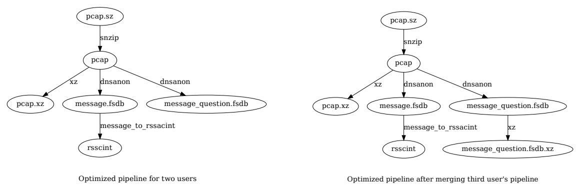

Figure 3: Optimized pipeline before and after merging a third user's pipeline.

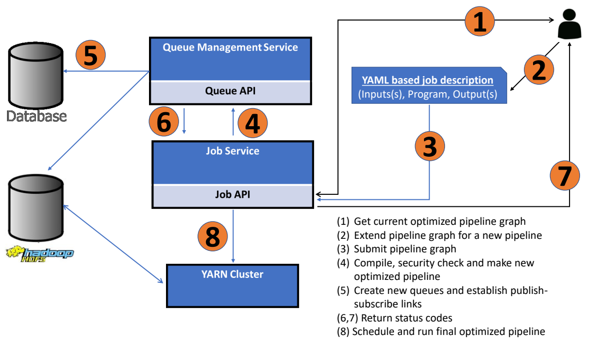
1
2
3
4
5
6
7
8
Database
Queue Management Service
Queue API
Job Service
Job API
YARN Cluster
YAML based job description
(Inputs(s), Program, Output(s)
(1) Get current optimized pipeline graph
(2) Extend pipeline graph for a new pipeline
(3) Submit pipeline graph
(4) Compile, security check and make new optimized pipeline
(5) Create new queues and establish publish-subscribe links
(6,7) Return status codes
(8) Schedule and run final optimized pipeline

Figure 4: A user submits their pipeline into Plumb.

input and output (without excessive buffering). Plumb enforces above restriction for efficient CPU utilization.

When a user submits their pipeline graph, Plumb evaluates it and integrates it with graphs from other users. Plumb first checks the user pipeline for any syntax errors, and checks access to input data. Then, system finds any processing or storage duplication across all users' jobs and removes it. Internally, our system abstracts storage as queues with data stored in a distributed file system, with input and output of each stage bound to specific queues.

If accepted, our system schedules each stage of the optimized pipeline to run in a YARN cluster. Stages typically require only small containers (1 core and 1 GB RAM), allowing many to run on each compute system, and avoiding external fragmentation that would occur scheduling larger jobs onto compute nodes. Also, system returns this container after processing one file for better and fair resource sharing on cluster. Our system assigns workers in proportion to current stage slowness due to lumpiness.

Finally, each large-file input is assigned a sequence number. Confirming all sequence numbers have been processed is a useful check on complete processing, and we can set aside files that trigger errors for manual analysis (§2.7).

2.3 The Pipeline Graph and De-duplication of Processing and Storage

Our first goal is to de-duplicate computation and storage across all users. To accomplish this goal, each user submits their computation as a pipeline graph and Plumb merges these graphs into a master pipeline graph after optimizing away duplication. Optimizations include combining stages that are identical computation, combining storage of intermediate and final output (§2.4) and identifying IO-bound stages where duplicating computation is more efficient than extra IO if it was eliminated (§2.5). To de-duplicate computation and data storage, we need to detect duplication, and eliminate the copies of computation and storage.

Our model is that each pipeline stage does one step of processing, consuming one or more inputs and producing data to be consumed by one or more downstream stages, or as an end-result. Data may stream directly between two stages that are executing concurrently, or it be buffered on stable storage when the computation cost of two stages are mismatched (§2.5), or if there is a backlog of processing.

We detect duplication by user-supplied pipeline graphs. Figure 2 is an example of a user's pipeline graph. In the pipeline graph, each stage defines its input sources and format, the computation to take place, and its outputs and their formats. Input formats are identified by global names such as pcap, DNS, anon-DNS, etc. By definition, any stages that identify input of the same format (that is, use the same format name) will share processed data of that type.

Plumb can now eliminate processing duplication by detecting stages submitted by multiple users that have the same input and output. We make this judgment based on textual equivalence of input and output of stages in the pipeline graph, since the general problem of algorithmically determining that two programs are equivalent is undecidable [38].

The outcome of merging users' pipeline graphs is a combined pipeline graph that efficiently executes computation for all users. We then schedule this computation into a Hadoop cluster with YARN [42].

Figure 3 shows optimized merged pipeline graph of first two users in Figure 1 while a third user is submitting his pipeline (Figure 2). Plumb will detect first two stages are same as in the current optimized pipeline and will remove them. Additionally, input of third stage is also already available and will be utilized.

2.4 Data Storage De-duplication

Although de-duplication *detects* identical computation and output, we still must efficiently and safely store a (single) copy of output.

We store exactly one logical copy of each unique data format and instance in a shared storage system. To store single copy of data, but with the user illusion of private individual data, we use a publish-subscribe system. This system emulates Linux hard-link like mechanism using database for meta-data and HDFS distributed file system to store actual data. Plumb on the behalf of a pipeline stage subscribes for its input to appropriate data store. Whenever an input data items appears, our system publishes it to all registered subscribers by putting an emulated hard-link per subscriber.

To uniquely identify a data item across system, we enforce a two-level naming scheme on all files names. Each file name has two parts: data store name based on user provided input or output names, and a time-stamp and a monotonically increasing number (for example: 20161108-235855-00484577. pcap.sz).

While there is one *logical* copy of each data block, when data is stored in HDFS, HDFS replicates the data multiple times. This replication provides reliability in the case of single machine failures, and locality when networks do not support full bisection bandwidth.

In our multi-user processing environment, security is very important and fully enforced. For all user interactions with the system, we use two-way strong authentication based on digital signatures. For data access authorization, we use HDFS group membership. Any user's pipeline is only accepted for execution if he has access to all the data formats mentioned in his pipeline graph.

2.5 Detecting I/O-Bound Stages

While we strive to elimination of de-duplicate computation and storage, in some cases, duplicate computation saves run-time by reducing data movement across stages via HDFS. Prior systems recognize this trade-off, recommending the use of lightweight compression between stages as a best-practice. We generalize this approach, by *detecting I/O-bound stages*; in §3.2 we show the importance of this optimization to good performance.

We automatically gather performance information about each stage during execution, measuring bytes in, out, and compute time. From this information we can compute the I/O-intensity of each stage as *intensity* $= (I + 3O)/P$ bytes per second. For our cluster's hardware, we consider stages with I/O-intensity more than 50 to suggest the computation should be duplicated to reduce I/O; clusters with different hardware and networks may choose different thresholds. Such threshold depends on cluster hardware and can be established empirically by running a stage known to have high I/O intensity. Table 1

stage	**I**	**O**	**P**	**IO-Intensity**
snzip	765	2048	28	246.75
dnsanon	2048	1590	180	37.87
rssac	717	73	180	5.2
xz1	2048	235	780	3.52
xz2	873	110	540	2.23

Table 1: Relative costs of Input and Output (I and O, measured in megabytes) processing (CPU seconds), and IO-intensity relating them.

shows an example of I/O intensity from our DNS pipeline. It correctly identifies snzip decompression stage as most IO bound among all.

Once I/O-bounds stages have been identified we allow the user to restructure the pipeline. We recommend that users duplicate lightweight computation to avoid I/O, by merging it with some other CPU-bound stage. Such merging should happen via pipes, so that inter-stage data flows via memory. Although in principle we could automate this step, placing the user in control of these kind of structural changes allows an expert to consider the consistency of the I/O data (does it vary across different inputs). It also allows an expert to make decisions such as how many stages to merge, and insures that they are aware of intermediate data formats when doing debugging.

2.6 Mitigating Structural Lumpiness

Structural lumpiness is when one stage of a pipeline is slower than others. We can define it as the ratio of the slowest stage to the fastest. As an example, our DNS pipeline (Figure 1) has a structural lumpiness ratio of 6.4, comparing the xz stage to the snzip stage.

We mitigate structural lumpiness by creating additional workers on slow stages and generally scheduling each stage independently. While this approach is taken in most big data computation systems (for example MapReduce [10], and even early systems like TransScend [17]), we considered alternative structures, and run I/O-bound stages concurrently. We describe these alternatives next and compare them in §3.6.

The challenge in scheduling is assigning cluster containers to pipeline stages. In principle, each stage can require one or more cores, and will have some typical runtime. One can run stages independently or together in parallel; when run in parallel they may choose to send data directly through inter-process communication, avoiding buffering I/O through a file system.

Table 2 shows the four options we consider. With *Linearized/multi-stage/1-core*, each input file is assigned a single container with one core, and it runs each stage sequentially inside that task. This scheme is efficient and flexible, providing excellent parallelism across input files. Though this scheme has high latency and yield no flexibility addressing lumpiness problem because container assignment is for full pipeline and not at stage level.

For *Parallel/multi-stage/multi-core with limits*, we assign multiple stages to a single YARN container, giving it as many cores as the number of stages in the pipeline. We then run all stages in parallel, with the output of one feeding directly into the other, and we strictly limit computation to the number of cores that are assigned (using an enforcing container in YARN). The advantage with this approach is that data can be shared directly between processes (the processes run in parallel), rather than through the file system. The difficulty is that it is hard to predict how many cores are required: structural lumpiness means some stages may underutilize their core (resulting in internal fragmentation), or stages with varying parallelism may overly stress what they have been allocated. Figure (a) and (b) in Figure 13 show, this configuration consumes substantially high container hours as compared to other choices and even latency benefits are mild on a heterogeneous cluster.

For *Parallel/multi-stage/multi-core without limiting*, we assign multiple stages to a single YARN container with as many cores as there are stages in the pipeline, running in parallel, with one core per stage (or perhaps fewer), but here we allow the container to consume cores beyond the container strictly allocates. The risk here is that resources are stressed—if we underprovision cores per container, we reduce internal fragmentation, but also stress the system as a whole when computation exceeds the allocated number of cores. Figure (c) in Figure 13 shows an 8 core server from a deployment, that started well with very little core waste, but became overloaded over time as workload characteristics changed. (This approach might benefit from approaches that adapt to system-wide overcommitment by adapting limits and throttling computation on the fly [54]; this approach is not yet widely available.)

With *Linearized/single-stage/1-core*, where we assign each stage (or two adjacent stages for I/O limited tasks (§2.5)), to its own YARN Container with a single core. In effect, the pipeline is *disaggregated* into many independent tasks. This approach minimizes both internal and external fragmentation: there is no internal fragmentation because each stage runs to completion on its own, and no external fragmentation because we can always allocate stages in single-core increments. It also solves structural lumpiness since we can schedule additional tasks for stages that are slower than others. The downside is data between stages must queue through the file system, but we minimize this cost with our I/O-based optimizations.

In §3.6 we compare these alternatives, showing that Linearized/single-stage/1-core is most efficient.

Run Configuration	Through-put	Latency	Cost Efficiency	Disk Use	Fragme-ntation	RAT	Cluster Sharing	Stage Scaling	LFD	Heterogeneous Cluster
Linearized/multi-stage/1-core	High	High	Good	High	No	Low	Good	Bad	Complex	Higher average latency
Parallel/multi-stage/multi-core with limits	Low	Low	Low	Low	Yes	High	Worse	Bad	Complex	Higher structural lumpiness
Parallel/multi-stage/multi-core without limiting	High	Low	Good	Low	No	High	Bad	Bad	Complex	Higher structural lumpiness
Linearized/single-stage/1-core	High	High	Good	Higher	No	Low	Good	Good	Simple	Lower latency

Table 2: Comparison of different configurations of stages into containers. RAT: Resource Allocation Time. LFD: Lumpiness management, Fault-tolerance, and Debugging.

2.7 Mitigating Data Lumpiness

Our approach to structural lumpiness (where each stage runs in its own container) can also address data lumpiness. The challenge of data lumpiness is that a shift in input data can suddenly change the computation required by a given stage.

To detect data lumpiness we monitor the amount of data queued at each stage over time (recall that each stage runs separately, with its on queue §2.4).

We can reduce the effects of data lumpiness by assigning computational resources in proportion to the queue lengths. Stage with the longest queue is assigned the most computational resources. We sample stages periodically (currently every 3 minutes) and insure that no stage is starved of processing.

Another risk of data lumpiness is that processing for a stage grows so much the stage times out. Our use of large-file processing helps here, since we can detect repeated failures on a given input and set those inputs aside for manual evaluation, applying the error-recovery processes from MapReduce [10] to our streaming workload.

3 Evaluation

We next evaluate the design choices to show their effects on efficiency. We evaluate efficiency by measuring cluster container hours spent, so lower numbers are better. For processing running in a commercial cloud, these hours translate directly to cost. A private cluster must be sized well above mean cluster hours per input file, so efficiency translates in to time to clear bursts, or availability of unused cluster hours for other projects.

3.1 Benefits of de-duplication

Our first optimization is to eliminate duplicate computation (§2.3) and data storage (§2.4) across multiple users. We show the effects of those optimizations here.

To measure the benefits of de-duplication, we use the DNS processing pipeline (Figure 1) with 8 stages, two of which (snzip and dnsanon) are duplicated across three users. We run our experiment on our development YARN cluster with configuration A from Table 3, scheduling stages as soon as inputs are available.

As input, we provide a fixed 8,16 or 24 files and measure total container hours consumed. In a real deployment, new data will always be arriving, and cycles will be shared across other applications.

	configuration	
resource	**A**	**B**
Servers	30	37
vCores	328	544
Memory (GB)	908	1853
HDFS Storage (TB)	139	224
Networking (Mbps)	1000	1000

Table 3: Cluster capabilities for experiments.

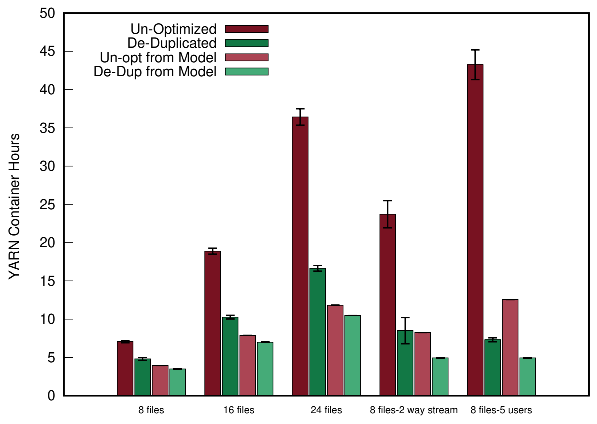

Figure 5: Comparing un-optimized (left bar) and de-duplicated (second bar) experimental pipelines and modeled (right two bars) for different workloads (bar groups). Mean of 10 runs with standard deviations as bars.

We compare measurements from our DNS pipeline running with related sample data in two configurations: un-optimized and de-duplicated. With the unoptimized configuration all stages run independently, while with de-duplication, identical stages are computed only once (see §2.3), producing an output file that is provided as input to each next stage (see §2.4).

We expect that removing redundant computation will reduce overall compute hours and HDFS storage, lowering time to finish a given input size and freeing resources for other concurrent jobs. These benefits are a function of how many duplicate stages can be combined, so an additional duplicated stage (for example, a decryption step) would provide even more savings relative to an unoptimized pipeline.

Processing: Figure 5 shows our results for three different numbers of inputs (each group of bars). The de-duplicated pipeline (the second bar from the left) is consistently faster than unoptimized (the leftmost bar).

We first compare increasing size workloads in the left three groups of bars. With 8 files as input, de-duplication uses 39% fewer container hours. These benefits increase with 16 and 24 input files, since the majority of compute time is in a stage (dnsanon) that can be de-duplicated.

We next vary the pipeline, adding an additional shared stage at the head of DNS pipeline for each user. This extra shared stage simulates a scenario when ingress and egress traffic is captured in separate large-files that must be merged (as can happen with optical fiber capture). This different configuration is shown in the fourth group of bars from the left of Figure 5, labeled "8 files-2 way stream". We observe 178% fewer container consumption by our de-duplicated configuration compared to unoptimized. This experiment shows how adding additional stages that can be de-duplicated has a greater benefit.

As a final variation of the pipeline, we add two users to the tail of the pipeline (consuming dnsanon output and emitting statistics data). In this case, the rightmost set of bars, we see that de-duplication shows a $6\times$ speedup relative to unoptimized, and a much greater savings compared to the either of the other two pipeline structures. Adding additional users late in the pipeline exposes significant potential savings.

To confirm our understanding, we defined a simple analytic model of speedup based on observed times and a linear increase of execution time for duplicated stages. The right two bars show predictions for optimized and unoptimized cases. The model significantly underestimates the savings we see (compare the change between the two right bars of each group to the change between the two left bars), but it captures the cost of additional input files and the benefits of additional stages that may be de-duplicated. This underestimate is due to I/O overhead due to synchronized file access (a "thundering herd problem") as we evaluate next.

Storage: Like processing, disk usage decreases with increasing de-duplication. Figure 6 shows total logical data that is stored over time (data is three-way replicated, so actual disk use is triple). Each data point is the average of ten runs.

In this for example, de-duplication reduces storage by 77% relative to un-optimized for the 8-file workload. Storage is particularly high in the middle of each run when the cluster is fully utilized. Finally, the larger amount of disk space not only consumes storage, but it can lead to disk contention, and even network contention in clusters that lack a Clos network.

This graph also highlights reduced latency that results from de-duplication. Since fewer resources are required,

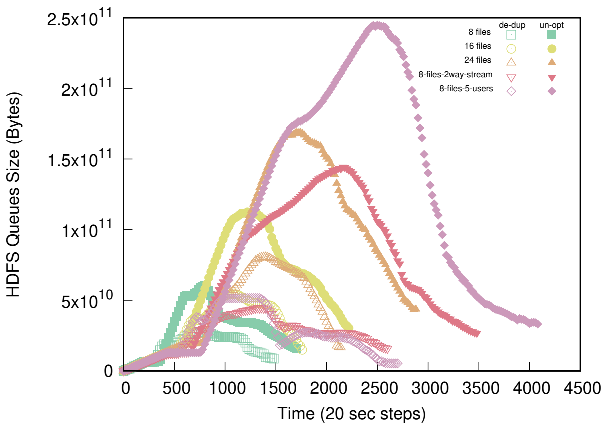

Figure 6: Amount of data stored over each processing run for different workloads (color and shape), unoptimized (filled shapes) and de-duplicated (empty shapes).

the backlog of input data is cleared more quickly, as shown by earlier termination of the de-duplicated cases (empty shapes) relative to their unoptimized counterparts (filled shapes).

Summary: Overall, these experiments demonstrate the significant potential efficiencies that are possible in a multi-user workload when users share common needs, and that those savings grow both with additional data and additional pipeline complexity. This savings can be used by other applications in shared clusters, or to reduce latency in a dedicated cluster.

3.2 I/O Costs and Merging

We next evaluate the improvements that are possible by reducing I/O. Reducing I/O is particularly important as the workload becomes more intense and contention over disk spindles grows. For example, when we increased the size of the initial workload from 8 to 16 and 24 (the left three groups in Figure 5), doubling the input size increases the container hours about $2.3\times$ for the un-optimized case.

We first establish that I/O contention can be a problem, then show that merging I/O-intensive stages (§2.5) can reduce this problem.

The problem of I/O contention: Figure 7 examines the container hours spent in each stage for our three workloads, both without optimization and with de-duplication.

We see that the compute hours of the snzip stage grow dramatically as the workload size increases. Even with de-duplication, the snzip stage consumes many container hours even though we know it actually requires little computation (Table 1).

This huge increase in cost for the snzip stage is because it is very I/O intensive (see Table 1), reading a file that is about 765 MB and creating at 2 GB output. Without

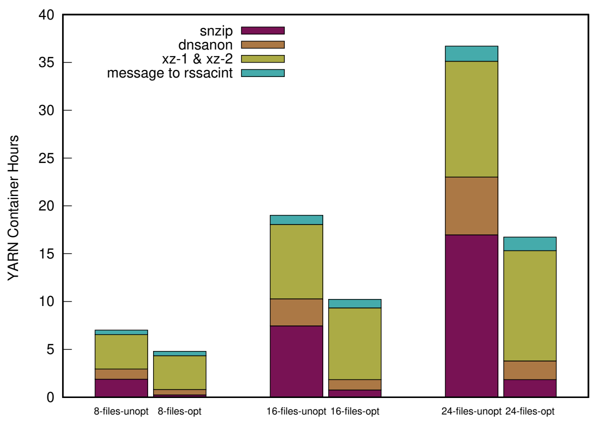

Figure 7: Computation spent in each stage (bar colors), unoptimized (left bar) and de-duplicated (right), for three size workloads (bar groups).

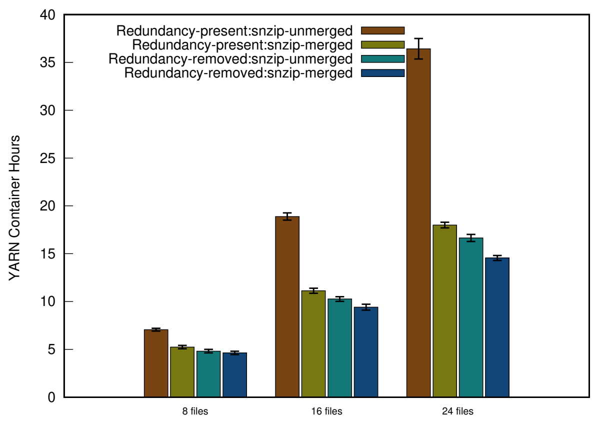

Figure 8: Computation with and without I/O merged with duplicated processing (left two bars) and de-duplicated (right two bars) for three size workloads (bar groups).

contention, this step takes 28 s, but when multiple concurrent instances are run, and with HDFS 3-way replication running underneath, we see a significant amount of disk contention.

Some of this cost is due to our hardware configuration, where our compute nodes have fewer disk spindles than cores. But even with a much more expensive SSD-based storage system, contention can occur over memory and I/O buses.

Benefits of Merging I/O-Bound Stages: Next we quantify the improvement in throughput that occurs when we allow duplicate I/O-intensive stages and merge them with a downstream, compute-intensive stage. This merger avoids storing I/O on stable storage (and it avoids file replication); the stages communicate directly through FIFO pipes. We expect that this reduction in I/O will make computation with merged stages more efficient.

We examine three different input sizes (8, 16, and 24 files) with the DNS pipeline with four different optimizations: first without computation de-duplication and with each stage in a separate process, then merging the snzip stage with the next downstream stage, then those two cases with compute de-duplication.

Figure 8 shows the results of this experiment, with different input sizes in a cluster of bars, and each optimization one of those bars in the cluster.

We expect merging the snzip stage with the next stage to reduce I/O, and to also reduce compute time by reducing disk contention. Comparing the left two bars (raw umber and olivetone) of each group shows that this optimization helps a great deal. We still see benefit after stage de-duplication (compare the right two bars of each group), but the relative savings is much less, because the amount of I/O contention is much, much lower.

Summary: We have shown that I/O contention can cause a super-linear increase in cost, balancing I/O across stages by running I/O intensive stages with the next stage can greatly improve efficiency, reducing container hours even if some lightweight stages duplicate computation. We saw up to 2× less container hours consumption and this benefit increases with higher I/O contention. That merging snzip helps should be expected—enabling compression for all stages of MapReduce output is a commonly used best practice, and the snzip protocol was designed to be computationally lightweight. However, Plumb generalizes this optimization to support merging *any* I/O-intensive stages, and we provide measurements to detect candidates stages to merge.

3.3 Pipeline Disaggregation Addresses Structural Lumpiness

Pipeline disaggregation (§2.6) addresses structural lumpiness by allowing additional copies of slower stages to run in parallel. Structural lumpiness occurs when two stages have unbalanced runtimes and they are forced to run together, allowing progress at only the rate of the slowest stage. Here we first demonstrate the problem, the show how pipeline disaggregation addresses it.

The problem of structural lumpiness: To demonstrate structural lumpiness we use a two-stage pipeline where first stage decompresses snzip-compressed input and recompresses it with xz, and the second stage does the opposite. Xz compression is quite slow, while xz decompression and snzip compression and decompression are quite fast. (The first stage has mean runtime of 19.65 minutes, standard deviation 5.8 minutes after 10 trials, while the second stage runs in just less than 1 minute.)

We compare two pipeline configurations: aggregated and disaggregated. With an aggregated pipeline, both

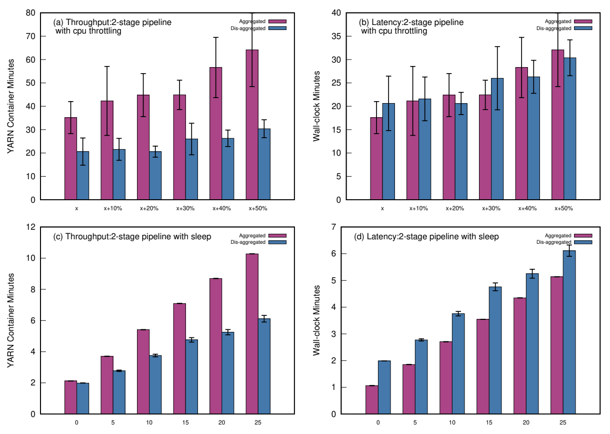

Figure 9: Comparing aggregated (left bar) and de-aggregated (right) processing, with delay added by cpulimit (top row) and sleep minutes (bottom row).

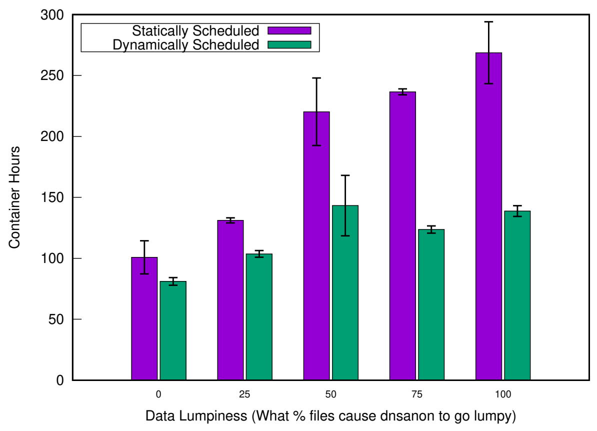

Figure 10: The throughput comparison of static (left bar) and dynamically scheduled workers (right) as lumpiness increases.

stages run in a single container with 2 cores and 2 GB of RAM, with each process communicating via pipes. For a disaggregated pipeline, each of the stages runs independently on a container with 1 core and 1 GB memory, with communication between stages through files. Thus the aggregated pipeline will be inefficient due to internal fragmentation since the first stage is 20× slower than the second, but the disaggregated pipeline will have somewhat greater I/O costs but computation will be more efficient.

During the experiment we vary lumpiness in two different ways. First, we can reduce the CPU speed available for use with cpulimit, and we can also lengthen computation by adding intentional sleep.

Figure 9 compares aggregated and disaggregated pipelines (the left and right of each pair of blocks), examining compute-time used (the left two graphs) and wall-clock latency (the right two graphs), with both methods of slow down (cpulimit in the top two graphs and sleep in the bottom two). We see that disaggregation is consistently *much* lower in compute minutes used (compare the left and right bars in the left two graphs). Aggregated is usually twice the number of compute minutes because one of its cores is often idling.

Disaggregation adds lightly to latency (compare the right bar to the left in the right two graphs), but only a fixed amount. This increase in latency represents the cost of queueing intermediate data on disk.

We see generally similar results for both methods of slowdown, except that CPU throttling shows much greater variance in the disaggregated case. This variance follows because our Hadoop cluster has nodes of very different speeds.

Summary: This experiment shows that disaggregation can greatly reduce the overhead of structural lumpiness, although at the cost of slightly higher latency.

3.4 Disaggregation Addresses of Data Lumpiness

Disaggregation is also important to address data lumpiness. With data lumpiness, changes in input temporarily alter the compute balance of stages of the pipeline. Disaggregation enables dynamic scheduling where Plumb adjusts the worker mix to accommodate the change in workload (§2.7). We next demonstrate the importance of this optimization by replaying a scenario drawn from our test application.

We encountered data lumpiness in our DNS Pipeline when data captured during a DDoS attack stressed the dnsanon stage of processing—TCP assembly increased stage runtime and memory usage six-fold. We reproduce this scenario here by replaying a input of 100 files (200 GB of data when uncompressed), while changing none, half, or all to data from a DDoS attack. (Both regular and attack traffic are sampled from real-world data.) We use our DNS processing pipeline and a workload of 100 files. We use YARN with 25 cores for this experiment. We then measure throughput (container hours) and time to process all input.

Figure 11 shows latency results from this experiment. (Throughput, measured by container hours, is similar in shape as shown in Figure 10.) The strong result is that dynamic scheduling greatly reduces latency as data lumpiness increases, as shown by the relative difference between each pair of bars. Without any DDoS traffic we can pick a good static configuration of workers, but dynamically adapting to the workload is important when data changes.

To show how the system adapts, each column of Figure 12 increases amount of data lumpiness (the fraction of DDoS input files), Each row of graphs shows one as-

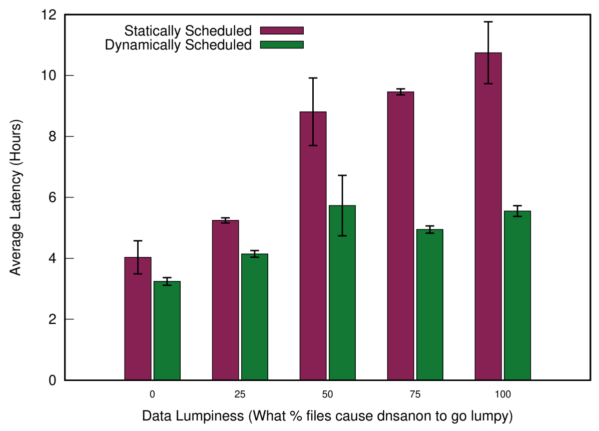

Figure 11: Comparing static (left bar) and dynamically scheduled workers (right) as lumpiness increases.

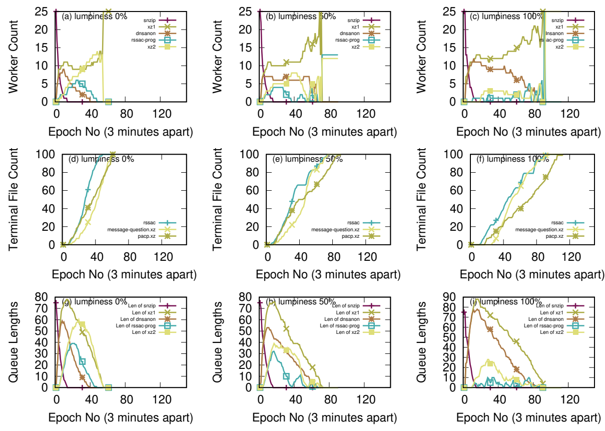

Figure 12: Comparing dynamic scheduling at 0%, 50% and 100% lumpiness: First row shows worker assignment to stages over time. Second row shows corresponding data production. Third row shows queue lengths for 5 stages.

pect of operation: the number of workers, the number of files output from each of the three final stages, and queue lengths at each stage. In the top row we see how the mix of workers changes with dynamic scheduling as the input changes, with more dnsanon processes (the brown line with the * symbol) scheduled as lumpiness increases.

3.5 Overall Evaluation on Real Inputs

To provide an overall view of the cumulative effect of these optimizations we next look at runtime to process two different days of real-world data, with a third day showing a synthetic attack to show differences in our system.

We draw on data from two full days of B-Root DNS: 2016-06-22, a typical day, with about 1.8 TB of input data in 896 files, and 2016-06-28, a day when there was a large DDoS attack [35]. Because of problems with rate-limiting associated with the DDoS attack, we actually capture much *less* data on that day, capturing about 1.4 TB of traffic in 711 files. To account for this input difference, we construct an synthetic attack day where we replicate one attack input 896 times, thus providing exactly the same volume as our normal day (1.8 TB in 896 files), but with data that is more expensive to process (the dnsanon stage is about 6× longer for attack data than regular data). This synthetic attack data depicts a worst case scenario where full day traffic is stressful. (Input data is compressed with xz rather than snzip as in our prior experiments, but xz decompression is quite fast so this does not change the workload much.)

scenario	**date**	**input**	**latency**
normal day	2016-06-22	1.8 TB / 896 files	8.3h
attack day	2016-06-25	1.4 TB / 711 files	6.20h
simulated attack	—	1.8 TB / 896 files	11.75h

Table 4: Latency with Plumb on full day worth of DNS data using 100 cores

We process this data on our compute cluster with a fixed 100 containers, while using all optimizations (de-duplication and lumpiness mitigation) and I/O-bound merging.

Table 4 shows the results of this experiment. The first observation is that, in all cases, plumb is able to keep up with the real-time data feed, processing a day of input data in less than half a day. Our operational system today processes data with a hand-coded system using a parallel/multi-stage/multi-core strategy (all stages run in a single large YARN container). Based on estimates from individual file processing times, we believe that Plumb requires about one-third the compute time of our hand-build system. Most of the savings results from elimination of internal fragmentation: we must over-size our hand-build system's containers to account for worst-case compute requirements (if we do not, tasks will terminate when they exceed container size), but that means that containers are frequently underutilized.

We were initially surprised that the processing day of DDoS attacks is complete faster than the typical day (6.2h vs. 8.3h), but that difference is due to there being less data on this day. This drop in traffic is due to a link-layer problem with B-Root's upstream provider where we were throttling traffic before collection due to its high rate, thus not receiving all traffic addressed to B-Root. Actual traffic sent to B-Root on that day was about 100× normal during the attack. We correct for this under-reporting with our synthetic attack data, which shows that a day-long attack requires about 40% more time to complete processing than our regular day.

Based on this success we are planning to transition Plumb into production use as testing is completed.

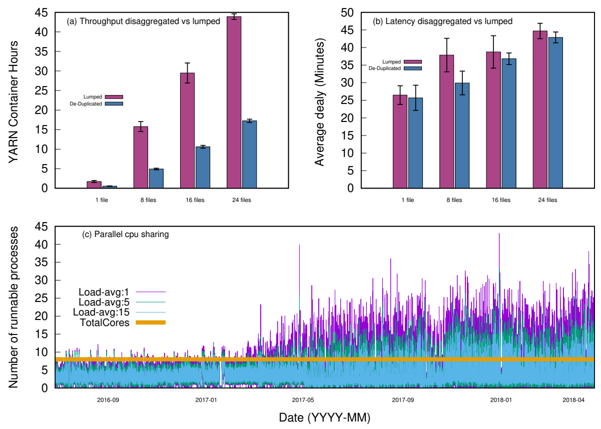

Figure 13: Comparing alternative configurations of stages per container. Parallel/multi-stage/multi-core with limits marked as lumped.

3.6 Comparing Design Alternatives for Stages per Container

In §2.6 we examined four alternatives (Table 2) for mapping pipeline stages into containers. We suggest that linearized/single-stage/1-core provides flexibility and efficiency. The alternative is to allow many stages run in one container, with or without resource over-commitment. Here we show that both of those alternatives have problems: without over-commitment is inefficient, and allowing over-commitment results in thrashing.

The top two graphs in Figure 13 compare parallel/multi-stage/multi-core with limits (left bar in each group) against linearized/single-stage/1-core ("disaggregated", the right bar in each group) for 4 sizes of input data. The left graph examines resource consumption, measured in container hours, and the right, latency. We see that disaggregation cuts resource consumption to less than half because it avoids internal fragmentation (idling cores). The effects on latency (right graph) is present but not as clear in this experiment; latency differences are difficult to see because CPU heterogeneity in our cluster results in high variance in latency.

The bottom graph in Figure 13 evaluates parallel/multi-stage/multi-core with and without limits by showing compute load over almost two years. Load is taken per minute by measurements of system load from one 8-core compute node of our current hand-built pipeline in operation today. There are at most 4 concurrent stages in our hand-built pipeline. Around Feb. 2017 (about one-third of the way across the graph) we changed this system from under-committed, with each container included 4 cores, to over-committed, with each container allocating only 2 cores. Under-committed resources ran well within machine capacities, but often left cores idle (as shown in the top two graphs). Over-commitment after Feb. 2017 shows that load average often peaks with the machine stressed with many more processes to run than it has cores. We conclude that it is very difficult to assign a static number of cores to a multi-process compute job while avoiding both underutilization and over-commitment. With dynamically changing workloads, one problem or the other will almost always appear. Disaggregation with the linearized/single-stage/1-core avoids this problem by better exposing application-level parallelism to the batch scheduler.

4 Related Work

We next briefly compare Plumb to representatives from several classes of big-data processing systems. Overall, our primary difference from prior work is our focus on integrating workflow from multiple users and coping with classes of performance problems (data and structural lumpiness) in a framework with minimal required semantics of the underlying data.

Workflow management: Workflow systems for scientific processing use explicit representations of their processing pipeline to capture dependencies and assign work to large, heterogeneous compute infrastructure. Unlike scientific workflow systems (for example [11, 9]) we only use workflow to capture data flow dependency to facilitate de-duplication. Big-data workflow systems [21, 45] bring together computation from different systems (perhaps MapReduce and Spark) into one workflow; we instead focus on optimizing inside a common, loosely coupled framework. Other systems place greater constraints on components (such as [26, 44, 25, 36]), allowing component-specific optimizations (for example, joins); our model does not strive for this level of integration. We consider data as opaque binary stream with unique naming scheme to find and unlock data and processing duplication. We currently use Apache YARN as a our executor [42].

Batch systems: Batch systems, such as MapReduce [10] and Dryad [20] focus on scheduling and fault tolerance, but do not directly consider streaming data, nor integration of multi-user workloads as we do. Google's pipelines provide meta-level scheduling, "filling holes" in a large cluster [13]. Like them, we are optimizing across multiple users, but unlike their work, we assume a single framework and leave cluster-wide sharing to YARN.

Streaming big-data systems: A number of recent systems focus on low-latency processing of small, streaming data, including Nephele [23], MillWheel [1], Storm [40], and at Facebook [8]. They often strive to provide simple semantics, such as transaction-like, exactly-once evaluation, and stream data in small pieces to minimize latency. We instead focus on large-file streaming, because our application needs a broader view than single records, and to provide easier fault-tolerance and completeness tests. Our focus on processing of larger blocks of data raises issues of structural and data lumpiness that differ from systems with fine-grain processing.

Streaming systems like Flink [5] optimize for throughput or latency by configuration; we focus on throughput while considering latency a secondary goal.

Programming-language integration: A number of programming languages provide abstractions that integrate with big-data processing and optimizations (for example [41, 47, 7, 37]). These systems often optimize only for a given job, while we integrate work from multiple users. Compilers can match an I/O pattern to a suitable optimization when framework enforces structure on I/O consumption [19] easily. SQL-compilers for declarative languages [6, 28, 39, 32, 4, 27, 3] and Parallel databases [31] match language abstractions to database access patterns and optimizations.

Most approaches integrated with programming languages focus on structured data. We instead target arbitrary programs processing data without a formal schema. In addition, they typically optimize each user's computation independently, while we consider integrating processing from multiple groups.

Big-data schedulers: Many systems consider different schedulers to optimize resource consumption or delay [15, 33, 46, 12], applications or to enhance data locality [48]. We work with existing schedulers, optimizing inside our framework to mitigate lumpiness.

Straggler handling: Lumpiness can be thought of a kind of performance problem like straggler handling. A number of systems cope with stragglers. Speculative execution [10] recovers from general faults. Static sampling of data [43, 16], predictive performance models [24, 55, 53], dynamic key-range adjustment [20], and aggressive data division [29] seek to detect or address data lumpiness. These systems are often optimized around specific data types or computation models, and assume structured data. Our system can be thought of as another approach to addressing this problem where there are few assumptions about the underlying data.

Resource optimization for high throughput: Several systems exploit close control and custom scheduling of cluster I/O [34] or memory [22, 49] to provide high-throughput or low latency operation. Such systems often require full control of the cluster; we instead assume a shared cluster and target good efficiency instead of absolute maximum performance.

One area of work emphasizes exploiting cluster memory for iterative workloads [52, 51, 2, 50, 49]. While we consider sharing across branches of a multi-user pipeline, our workloads are streaming and so are not amenable to caching.

Multi-user systems: Like our work, Task Fusion [14] merges jobs from multiple users. They show the potential of optimizing over multiple users, but their work is not automated and does not address performance problems such as structural and data lumpiness.

Several systems suggest frameworks or libraries to improve cluster sharing and utilization [30, 18]. Some of whose resemble our optimized pipeline, but we focus on a very simple streaming API and loosely coupled jobs.

5 Conclusions

We have described Plumb, a system for processing large-block, streaming data in a multi-user environment. Users specify their workflow, allowing Plumb to de-duplicate computation and storage. Plumb accommodates both structural and data lumpiness by dynamically scheduling workers for each processing data. We have exercised Plumb on workloads drawn from DNS processing, showing it shifts in traffic due to events such as DDoS attacks, and that it significantly more efficient than current inflexible, hand-built systems. We expect that Plumb will be of use to similar kinds of applications that need to analyze streaming data structured as large blocks.

6 Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-18-2-0280. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[3] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 3–14. ACM, 2013.

[4] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou. Recurring job optimization in scope. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*,

SIGMOD '12, pages 805–806, New York, NY, USA, 2012. ACM.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[6] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[7] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[8] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1087–1098. ACM, 2016.

[9] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. *Workflow Management in Condor*, pages 357–375. Springer London, London, 2007.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[11] E. Deelman, G. Singh, M. hui Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *SCIENTIFIC PROGRAMMING JOURNAL*, 13:219–237, 2005.

[12] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference*, number EPFL-CONF-208856, pages 499–510. USENIX Association, 2015.

[13] D. Dennison. Continuous pipelines at google. In *SRECon Europe 2015*, page 12, Dublin, Ireland, 2015.

[14] R. Dyer. Task fusion: Improving utilization of multi-user clusters. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, &#38; Applications: Software for Humanity*, SPLASH '13, pages 117–118, New York, NY, USA, 2013. ACM.

[15] M. Ehsan and R. Sion. Lips: A cost-efficient data and task co-scheduler for mapreduce. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 2230–2233, May 2013.

[16] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[17] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. of 16th*, pages 78–91, St. Malo, France, Oct. 1997. ACM.

[18] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A common substrate for cluster computing. In *HotCloud*, 2009.

[19] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[21] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: Towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 4:1–4:10, New York, NY, USA, 2012. ACM.

[22] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[23] B. Lohrmann, D. Warneke, and O. Kao. Massively-parallel stream processing under qos constraints with nephele. In *Proceedings of the 21st international symposium on High-Performance Parallel*

and Distributed Computing, pages 271–282. ACM, 2012.

[24] K. Morton, M. Balazinska, and D. Grossman. Paratimer: A progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.

[25] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[26] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2011.

[27] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[29] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *HotOS*, volume 13, pages 14–14, 2013.

[30] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[31] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[32] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[33] J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley. Performance-driven task co-scheduling for mapreduce environments. In *2010 IEEE Network Operations and Management Symposium - NOMS 2010*, pages 373–380, April 2010.

[34] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, New York, NY, USA, 2012. ACM.

[35] Root Server Operators. Events of 2016-06-25. Technical report, Root Server Operators, June 29 2016.

[36] H. Saputra and T. Yim. Build large scale applications in yarn with apache twill. *Apache Big-Data North America*, May 9 2016.

[37] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2012.

[38] M. Sipser. *Introduction to the Theory of Computation, Theorem 5.4 EQ_{TM} is undecidable.* Course Technology, Boston, MA, third edition, 2013.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[40] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[41] N.-L. Tran, S. Skhiri, A. Lesuisse, and E. Zimányi. Arom: Processing big data with data flow graphs and functional programming. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 875–882. IEEE, 2012.

[42] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[43] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[44] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, R. Sears, B. Sezgin, and J. Wang. Reef: Retainable evaluator execution framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1343–1355, New York, NY, USA, 2015. ACM.

[45] D. Wu, L. Zhu, X. Xu, S. Sakr, D. Sun, and Q. Lu. Building pipelines for heterogeneous execution environments for big data processing. *IEEE Software*, 33(2):60–67, 2016.

[46] Y. Yao, J. Wang, B. Sheng, J. Lin, and N. Mi. Haste: Hadoop yarn scheduling based on task-dependency and resource-demand. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 184–191. IEEE, 2014.

[47] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[48] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[50] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[51] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[52] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.

[53] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[54] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.

[55] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th international conference on Autonomic computing*, pages 53–62. ACM, 2012.