Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

RESEARCH ARTICLE

Plumb: Efficient Stream Processing of Multi-User Pipelines[†]

Abdul Qadeer*[1] | John Heidemann[2]

[1]Information Sciences Institute, University of Southern California, CA, USA
[2]Information Sciences Institute, University of Southern California, CA, USA

Correspondence
*Abdul Qadeer, 4676 Admiralty Way, Suite 1001, Marina Del Rey, CA 90292 Email: aqadeer@isi.edu

Present Address
4676 Admiralty Way, Suite 1001, Marina Del Rey, CA 90292

Abstract

Operational services run 24x7 and require analytics pipelines to evaluate performance. In mature services such as DNS, these pipelines often grow to many stages developed by multiple, loosely-coupled teams. Such pipelines pose two problems: first, computation and data storage may be *duplicated across components* developed by different groups, wasting resources. Second, processing can be *skewed*, with *structural skew* occurring when different pipeline stages need different amounts of resources, and *computational skew* occurring when a block of input data requires increased resources. Duplication and structural skew both decrease efficiency, increasing cost, latency, or both. Computational skew can cause pipeline failure or deadlock when resource consumption balloons; we have seen cases where pessimal traffic increases CPU requirements 6-fold. Detecting duplication is challenging when components from multiple teams evolve independently and require fault isolation. Skew management is hard due to dynamic workloads coupled with the conflicting goals of both minimizing latency and maximizing utilization. We propose *Plumb*, a framework to abstract stream processing as large-block streaming (LBS) for a multi-stage, multi-user workflow. Plumb users express analytics as a DAG of processing modules, allowing Plumb to integrate and optimize workflows from multiple users. Many real-world applications map to the LBS abstraction. Plumb detects and eliminates duplicate computation and storage, and it detects and addresses both structural and computational skew by tracking computation across the pipeline. We exercise Plumb using the analytics pipeline for B-Root DNS. We compare Plumb to a hand-tuned system, cutting latency to one-third the original, and requiring 39% fewer container hours, while supporting more flexible, multi-user analytics and providing greater robustness to DDoS-driven demands.

KEYWORDS:
multi-user pipelines,data and processing de-duplication,unstructured data,binary UDF,arbitrary operators

1 | INTRODUCTION

As the field of big data analytics matures, workflows are increasingly complex and often include components that are shared by different users and built by different teams. With multiple groups contributing to different stages of a complex workflow, it is easy to lose track of computation that may be shared across different groups.

[†]An early report of this work was presented at a poster session at ACM's SoCC 2018. doi:10.1145/3267809.3275461

Domain Name System (DNS, providing a mapping from human-readable names to computer-friendly addresses) analytics is an example of a complex workflow with across-team duplicate processing. A typical DNS service is geographically distributed and needs monitoring to estimate service quality[1], while analysis of the same data can detect malicious activity[2]. With geographically distributed sites and limited local processing capability, data from multiple sites may be back-hauled to a central site for analysis. High traffic volumes and bandwidth limitations dictate that data be collected in large blocks, compressed, and shipped to the processing site. In our example, two different teams (service monitoring, malicious activity detection) need the same data. Their initial steps are the same, where they decrypt, uncompress, and clean the data. Later processing diverges into specialized workflow. Considerable efficiency can be gained by detecting and removing duplication in the common initial stages of the pipeline. When future needs arise (perhaps an anti-DDoS analysis team), additional duplication of and similar wasted resources will occur if steps are not taken to avoid it.

In this paper we propose a new framework, *Plumb*, a workflow system for multi-stage pipelines, where parts of computation and data are shared across different groups. Plumb focuses on streaming workflows where data is processed in blocks, a middle ground between large-size batch processing and small-record streaming. A particular challenge in this problem domain is *structural and computational skew* since the computation of different stages and different data blocks can vary by a factor of ten due to differences in the work or data.

Continuing our DNS example, both teams need to consume a stream of DNS data packaged as large-blocks. Many analytics programs for this domain can emit output after consuming a small part of input data. An example is the TCP reassembly. The DNS captured data have TCP flows, where each flow can have multiple DNS records. Processing needs to stitch together all such flows so that it can extract DNS data out of it. Our captured large blocks (each with about 2 GB of DNS data before compression) have many such flows in a single block. DNS over TCP has the property that flows are small in size and nearby in time. Due to these properties, the TCP reassembly program can start emitting output data without consuming all the input. Such streaming operators make it possible to run adjacent pipeline stages concurrently (like Unix pipelines). The output of one stage is going into the input of the next workflow stage. Let us say the input to TCP reassembly was from a fast decompression (like snzip), and both stages run concurrently. TCP reassembly is slower than the snzip stage, and snzip waits for slower TCP stage, an example of structural skew in the pipeline. Additionally, it is hard to accommodate such skew statically. Often, under a malicious attack, TCP reassembly stage can take substantially longer than the typical case, hence throwing any static adjustments into the disarray (an example of computational skew).

Plumb is designed for *large-block, streaming* (LBS) workloads. Traditional map-reduce has focused on batch processing, and systems such as Storm[3] consider streams of small records. We have identified a class of applications that involve long-term streams of data, but where the processing requires examination of large blocks of data (say, 10 to 1000 megabytes) at a time. These applications need to capture temporal or spatial locality, integration with existing serial tools, and to support fault tolerance and human-guided recovery in the long-running data processing. Applications that require large-block data preclude the use of adaptive sharding schemes to present skew. Plumb exploits this “middle ground” where per-block scheduling is possible.

One block is the smallest unit of consumption by an instance of the analytics in the LBS domain. Skew management schemes that rely on sharding data and utilizing many workers (one per shard) are not suitable for LBS. For example, arbitrarily breaking data of a TCP flow makes it harder to stitch it together. One needs a stage where we need to gather everything again based on the key before stitching could be applied. Similarly, schemes that place data in memory suffer due to a rapidly changing working set. It implies that either data spills to the disk or we discard it altogether. Skew can make one branch of a workflow slower, and when it needs some data (and it is not there), it will cost penalties in terms of disk reads (or worse full recomputation using data provenance). In both cases, one challenge is that for how long to keep state before moving on. LBS processing has the temporal and spatial context of data; hence its state management is mostly confined within a block and is much simpler.

Plumb’s first goal is to identify duplication of computation and storage that can occur when different groups share components of a pipeline. When workflows are shared across users, work done in common stages will be duplicated if each user assumes they begin with raw input, particularly as the workflow evolves throughout development. Previous work has made strides for processing de-duplication where operator semantics are well defined (for example[4,5]), or where multiple users use the same programming language and run-time[6]. Databases and other systems[7] sometimes save and share intermediate or final results because workload access patterns are amenable to caching. Finding duplication in arbitrary user-defined programs remains challenging, and LBS workloads are single-pass and hence not a good fit for caching. Our *novel* equivalence definition exploits LBS constraint where we define that for any two pipeline stages of all users, if the set of inputs and outputs blocks are the same, then the processing is the same.

The second problem we address is *skew*. Data skew is a known problem, where many data items fall into one processing bin, slowing the overall workflow[8]. We address computational and structural skew under the new context of LBS, where adaptive sharding is not possible. *Computational skew* occurs when a bin of data takes extra long to process, not necessarily because there is more data, but because the data interacts with the processing algorithms to take extra time. *Structural skew* occurs when one stage of the processing pipeline is noticeably slower than other stages.

We address structural skew in Plumb by scheduling additional processing elements when one data block or one stage falls behind. Plumb decouples processing for each stage of the workflow, buffering output when required and supporting independent stage execution. However, to avoid overhead from data buffering, Plumb can run stages concurrently when they are well matched. This decoupling also addresses computational skew, since additional computation can be brought to bear when specific data inputs take extra time.

Plumb provides a *new abstraction* to its users to solve the challenges of duplication (of code and data) and skew (structural and computational) in a multi-user, shared workflow. The strength of this abstraction is that it is simple to use for the end-user, where they specify their workflow using domain-specific names of inputs and outputs. Users do not need to remember or find the same programs because Plumb infers code similarity from input/output names. Plumb uses naming in a novel way to detect and remove duplication and manage skew automatically. Our new abstraction lets individual users/teams remain decoupled from each other, yet Plumb tackles sharing and skew aspects.

We compare Plumb to a hand-tuned system, resulting in one-third the original latency (§4.7) and 39% less container hours (§4.1). Plumb's abstractions promise to support a much more flexible, multi-user analysis (§5) while being robust to DDoS-driven changes in processing needs.

The *contribution* of this paper is to show that Plumb provides a new abstraction that makes it easy for analysts to express complex, multi-stage, multi-user workflows of loosely-coupled processing elements, and while still allowing optimization to detect and eliminate duplicate processing and storage. In addition to eliminating duplication in a multi-user workflow, Plumb automatically detects and eliminates structural and computational skew. Our abstraction also facilitates easy debugging due to per-block processing, as it maps directly to user mental context of serial processing. We evaluate these properties with an operational workflow with components contributed by multiple people over about four years.

In summary, we show that real-world analytics often have substantial similarity across components developed by different teams. We identify Large-Block Steaming as a new domain for big data processing, and identify computational skew as a new challenge (in addition to existing data skew) for such workflows. Current solutions are deficient in solving these problems (see §6 for detailed comparison with the current systems). We show that workflow component naming, coupled with new strategies to manage both computational and structural skew can reduce duplication of computation to provide efficient processing. Finally, Large-Block Streaming provides a simple abstraction while addressing these challenges.

2 | SYSTEM DESIGN

We next describe Plumb's requirements, and how it reduces duplicate storage and processing and addresses skew.

2.1 | Definitions, Goals, Assumptions, and Scope of Work

In this section, we define different terms as they apply to our work. We also list the goals, assumptions, and scope of our work.

Definitions: Plumb supports streaming, large-block data that is opaque to the system. By *streaming* we mean that data is unbounded—new data continues to arrive at all times. For our work, we assume long-term, soft-real-time processing requirements, but our primary goal is to guarantee processing 100% of provided data. Thus we support buffering to handle temporary bursts or maintenance, and do not target hard-real-time guarantees as some other stream processing systems. (Typically we see end-to-end processing delays of minutes, if necessary backlogs can grow to fill disks.) Plumb is not a batch processing system; it processes an unbounded stream of large-block data. *Batch processing* systems, instead consume a fixed amount of data for a one-time task. Our system is therefore a middle ground between pure-batch systems like map-reduce, and fine-grain streaming systems that process individual (small) records.

By *large block*, we mean a collection of data into the smallest unit of consumption in any processing. The exact size of the block is domain-specific and depends on factors like data collection and relay and spatial and temporal efficacy of the operators.

Plumb can work with both structured and unstructured data because we see data and user programs as opaque, and operators are complex. By *unstructured data* we mean that data does not necessarily conform to a specific schema, as one would see in a relational database or a streaming system following that model. By *data opaqueness*, we mean that framework does not understand the structure of the underlying data. By *complex code*, we mean an arbitrary binary whose semantics are not known. The *operator semantics* precisely tell what code does on data. While many relational operators have well-known semantics, we assume no knowledge of our user code.

Plumb uses three-way data replication to guard against data loss. *Three-way data replication* means that we make three discreet copies of the same data for data reliability. These copies use in-lined replication where all three copies are in progress in parallel. Processing with streaming operators and in-lined replication, input reading, processing, and replicated output writing overlap in time. By *streaming operators*, we mean code can start emitting output without first consuming full input. Most of our user code in this work are streaming operators.

Goals: Our system provides *data completeness* as an SLA (Service Level Agreement). We strive hard not to lose any data, provide high system utilization and throughput, and best-effort low latency. We assume that most of our pipeline has low change velocity, and processing is for the long-term. Our system is not suitable for ad-hoc, exploratory analyses where the pipeline structure is changing rapidly (hundreds of times per hour).

Assumptions: We assume that our system users are *non-malicious*, and that processing is *idempotent*. We assume lack of malice because the cost of protecting against an adversarial user is quite high and the users are part of the same organization. However, we take typical precautions against bugs, for example protecting data from processing stages that fail. We also use group membership and authentication to control data access. Idempotent processing implies user code generates the same output when provided the same input, even if run multiple times or other things change.

Scope: Currently Plumb provides *per-block* processing. Per-block processing implies the system does not manage any state across the blocks. State within a large-blocks can be maintained by the applications (and outputted in the outgoing data). For state maintenance across blocks, applications need windowing abstraction. That windowing is currently work in progress. However, applications can do stateful computation if they work at it: each app can write state to a new place, then another thread can watch for the state to appear and replay it (in order) when it is all there. However, an application would do that, not plumb. Our next work on windowing will fix this. While our system takes special care to run each block once at a time, under some scenarios (for example, network partitioning), processing for a block can run more than once concurrently. Our system ensures that output from only one of these concurrent instances goes to the output. Currently, our system does not support shuffling or any other reductions on multiple blocks. Users needing such facilities use Plumb API to take data out of the system, process it using a specific abstraction, and bring them back inside Plumb. Our next new abstraction based on windowing is currently under development to enable multiple abstractions out-of-the-box (§7).

2.2 | Design Requirements and Case Study

Our system is designed to solve *multi-user* processing problems while being *easy-to-use*. Each user expresses their workflow through a framework that abstracts inputs and outputs, allowing Plumb to detect and eliminate duplicate computation and storage. The framework is flexible and we have evolved this framework over time, adding additional optimizations.

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

Large-block streaming means data constantly arrives at the system, and it is delivered in relatively large blocks. Many applications involve continuous data collection with analytics. Unlike other streaming systems that emphasize small events (perhaps single records), we process data in large blocks—from 512 MB to 2 GB in different deployments. Large blocks are important for applications where actions frequently span multiple records that are nearby in space or time, since those records can often be processed together. Large blocks also amortize processing costs and simplify detection of completeness (§2.3) and error recovery (§2.8).

For our sample application of DNS processing, large blocks are motivated by the need to do TCP reassembly, since all packets for a TCP connection are usually in the same block. Compression is much more efficient on bulk data (many MB). 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 (instead of millions of small records). Debugability is increasingly crucial in today's complex systems, and large-blocks based processing makes manual inspection not only viable but also most of debugging expertise from the serial world is useful in this context as well.

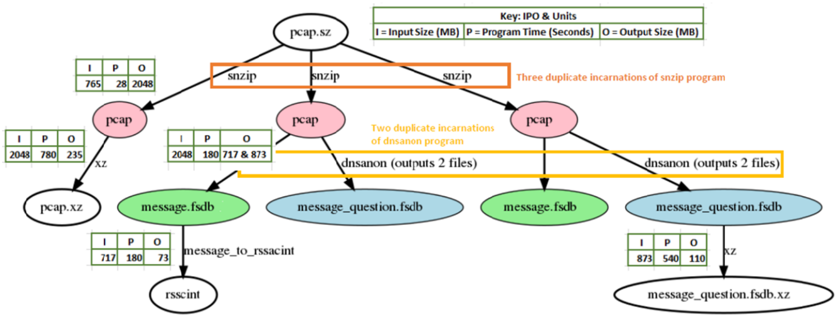

FIGURE 1 The DNS processing pipeline, our case study described in §2.2. Intermediate and final data are ovals, computation occurs on each arc.

```
# 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.

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 in the current hand-coded system) we have queued almost two weeks of data, requiring more than a week to drain.

Ease-of-use for the programmer is an explicit design goal in Plumb. As with map-reduce[9], individual processing modules focus on discrete inputs and outputs and the framework handles parallelism. Inspired by Storm[3] queue management, we adapt parallelism of each pipeline stage to match the workload skew (§2.8).

Pipeline-graph Abstraction and Programming Model: Users express their workflow by defining an input-processing-output (IPO) graph, where each processing element has one or more inputs and outputs. The format of any input or output is defined by a name, and all users share a common namespace. (We use conventions analogous to the naming of Java or Go libraries to deconflict namespace management.) The full pipeline is the concatenation of such named stages in the IPO workflow. Two users utilizing the same I or O name implies referring to same data, and processing stages are considered identical when they share the same inputs and output types.

User semantics for a processing stage is to read a large-block, process it, and write to a new large-block. Each large block is identified by its type and a unique sequence number. In this framework, operators and data are opaque to the system and processing elements are programs supplied by the user and are idempotent. User programs can be other framework drivers for specific operator needs. Many of networking pipelines use well-established tools that are serial and can relatively easily adapt to cluster environment using our large-block based abstraction. User programs can either tolerate possible analytic errors due to data cut-off at block boundaries, or using more sophisticated techniques to handle such cases if needed. Figure 2 is an example use of pipeline graph abstraction.

Case Study - DNS: 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 (`rsscint`)[1] , and processed data (`message_question.fsdb.xz`). Generating this output logically requires five steps (the IPO squares), each of which has very different requirements for I/O and computation (shown later as Table 1). This pipeline has been in use for 3 years and takes as input 1.5 to 2 TB of data per day. We are extending it with additional steps, and migration of the the current hand-crafted code to Plumb is already complete and in production use (§5).

2.3 | Plumb Overview

We next briefly describe workflow in Plumb to provide context for the optimizations described in the following sections. Figure 3 shows overall Plumb workflow.

Users provide (step 2 in Figure 3) Plumb their workloads as with a YAML-based pipeline specification (Figure 2). Plumb integrates workloads from multiple users and can provide both graphical (similar to Figure 1 but with all the redundancies removed) and textual (Figure 2) descriptions of the integrated graph. Plumb detects and optimizes away duplicate stages from multiple users (§2.4). Data access from each user is protected by proper authentication and authorization, mediated by a database of all available content (tag 5).

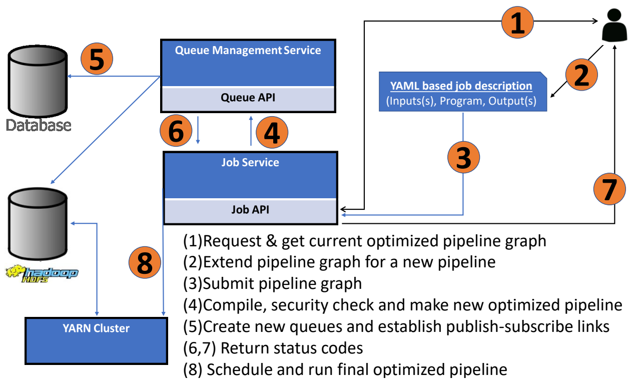
Queue Management Service
Queue API
Database
Job Service
Job API
YAML based job description
(Inputs(s), Program, Output(s)
YARN Cluster
1
2
3
4
5
6
7
8
(1)Request & get current optimized pipeline graph
(2)Extend pipeline graph for a new pipeline
(3)Submit pipeline graph
(4)Compile, security check and 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 3 A user submits his pipeline into Plumb.

Each pipeline stage has a single-user supplied program. Programs with only one input read from standard input, and with one output write to standard output. For programs with multiple inputs or outputs, they are specified as command-line FIFO stream arguments. Each stage is allocated a single core. These resource limits allow Plumb to densely allocate processing over many cores in multiple computers. Allocation is done within Hadoop YARN, so stages are isolated from each other.

When a user submits his or her pipeline graph, Plumb evaluates it and integrates it with graphs from other users. Plumb verifies the pipeline syntax. Then the 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 YARN containers (1 core and 1 GB RAM), allowing many to run on each multi-core computer in the cluster, and avoiding external fragmentation (when multiple cores are needed for a task but are not available on any particular computer) due to scheduling larger jobs onto compute nodes. Also, the system returns this container after processing one instance of a user program for better and fair resource sharing on the cluster. Our system assigns workers in proportion to the current stage slowness due to skew.

Finally, each input block is stored as a file with a unique sequence number. Confirming all sequence numbers have been processed is a useful check of completion, and we can set aside files that trigger errors for manual analysis (§2.8).

To provide de-duplication, our optimizations combine stages that are identical computation (§2.4), combine storage of intermediate and final output (§2.5) and identify IO-bound stages that are better to merge with up or downstream computation (§2.6). We also explain solutions to structural skew (§2.7) and computational skew (§2.8) in large-block, multi-user, streaming workloads.

2.4 | De-duplicating Processing and Data via the Pipeline Graph

We detect duplication by identifying identical data when merging user-supplied workflows. Figure 2 is an example of a user's pipeline graph. In the pipeline graph, each stage defines its input, the computation to take place, and its outputs. Input and output are identified by global names such as pcap, DNS, anon-DNS, etc. Data is opaque for Plumb and any format or structural information about inputs and outputs is left for user applications. By definition, any stages that use the same named inputs/outputs, refer to the same backing data.

Plumb can now eliminate processing duplication by detecting stages submitted by multiple users that have the same set of input and output. We make this judgment based on textual equivalence of input and output names of a program in the pipeline graph, since the general problem of algorithmically determining that two programs are equivalent is undecidable[10]. Plumb considers all user programs as black boxes and assumes no knowledge of operators. Many operators in networking domain does not lend itself for SQL like representation and do arbitrary computation.

The outcome of merged users' pipelines is a combined pipeline that has all users' computation. We then schedule this computation in a cluster with YARN[11].

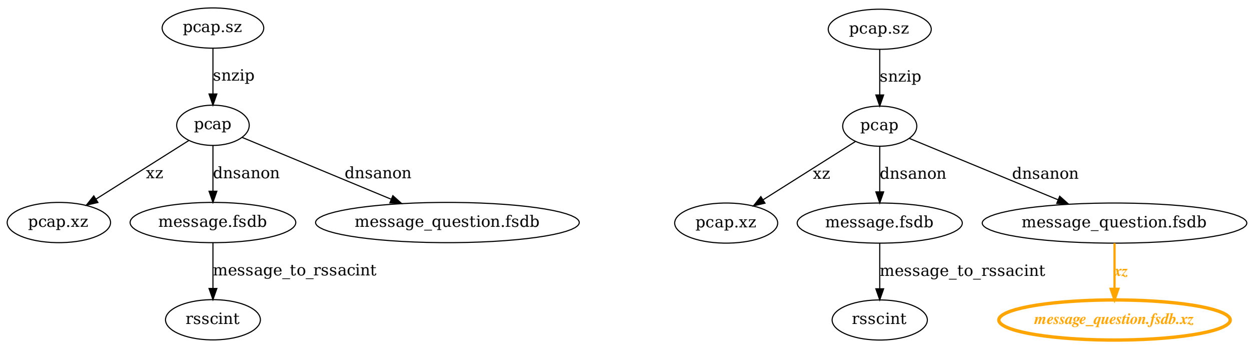

FIGURE 4 Optimized pipeline before (left picture with two users) and after (right picture) merging a third user's pipeline.

As an example, when a new user (right-most branch in Figure 1) submits his pipeline (Figure 2), Plumb detects that first two stages are identical to some other users' stages and de-duplicates them. Consequently, input to the third stage of new user was already available and hence reused (Figure 4).

2.5 | Data Storage De-duplication

De-duplication *detects* identical computation and data requires safe storage of a single copy of each input and output.

We store exactly one copy of each large-block (that is in a single file) 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-links using a database for meta-data and the HDFS distributed file system to store actual data. Plumb for each pipeline stage subscribes it to its input. Whenever an input data item appears, our system publishes it to all registered subscribers by putting an emulated hard-link per subscriber.

The storage efficiency and at-least once processing semantics rely on the unique identification of data blocks. 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 (in pipeline graph), and a time-stamp and a monotonically increasing number (for example: 20161108-235855-00484577. pcap.sz).

Although there is one logical copy of each output, HDFS replicates the data multiple times (default: 3). HDFS replication provides reliability in the case of machine failures.

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 that user has access to all the data formats mentioned in his pipeline graph.

2.6 | Detecting I/O-Bound Stages

While we strive for computation and storage de-duplication, 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 §4.2 we show the importance of this optimization to good performance.

We automatically gather performance information about each stage during execution, measuring bytes in (I), out (O), and compute time (P that includes time for data read and write along with CPU time). From this information we can compute the I/O-intensity of each stage as follows:

$$IO\ intensity\ of\ a\ pipeline\ stage = \frac{(I + (HDFSReplicationFactor \times O))}{P} \quad (bytes\ per\ second) \tag{1}$$

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 (I read time + CPU time + O write time in seconds), and IO-intensity relating them.

Run Configuration	Through-put	Latency	Cost Efficiency	Disk Use	Fragme-ntation	RAT	Cluster Sharing	Stage Scaling	SFD	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 skew
Parallel/multi-stage/multi-core without limiting	High	Low	Good	Low	No	High	Bad	Bad	Complex	Higher structural skew
Linearized/single-stage/1-core	High	High	Good	Higher	No	Low	Good	Good	Simple	Lower latency

TABLE 2 Comparison of different configurations for stages to containers. RAT: Resource Allocation Time. SFD: Skew management, Fault-tolerance, and Debugging. Fragmentation refers to unused CPU inside an allocated execution container.

Here the value of *HDFSReplicationFactor* is thrice due to our use of 3-way HDFS data replication. For our cluster's hardware, we consider stages with I/O-intensity more than 50 (bytes/second) 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 shows an example of I/O intensity from our DNS pipeline. It correctly identifies snzip decompression stage as most IO bound among all.

Identification of I/O-bound stages allows the user to restructure the pipeline. We recommend that users duplicate lightweight computation to avoid I/O by connecting it through a pipe with another CPU-bound stage. In principle, this step could be automated, but we encourage user control of structural changes to support informed decisions of what is merged and ensures that users are aware of intermediate data for debugging.

2.7 | Mitigating Structural Skew

Structural skew is when one stage of a pipeline is slower than others in a pipeline. 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 skew ratio of 27.9, comparing the xz stage to the snzip stage.

One way the structural skew can materialize is when unbalanced pipeline stages concurrently run. Individual users often use such run configuration, assuming it will minimize their wait time. In reality, doing so structural skew wastes computational resources because faster stages need to wait on slower stages, and during this time, resources are held but idle.

We mitigate structural skew by creating additional workers on slow stages and scheduling stages independently. We share this approach with other big-data systems (for example MapReduce[9], and even early systems like TransScend[12]), but we go beyond strictly independent stages by supporting stage execution both independent (here) and concurrent (using pipelines for reduced IO contention). Such concurrent execution (as compared to traditional operators where next stage can only start when previous one completes) provides many pipeline execution alternatives. We describe these alternatives next and compare them in §4.6.

When mapping pipeline stages to YARN containers, we can merge stages or split them across containers, so one stage my require one or more cores, with parallelism either between containers or inside each container. In addition, containers can buffer output in the file system, or stream it through pipes. 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. This scheme has high latency (because pipeline stages sequentially run one after the other) and is unable to mitigate the computational skew problem (because we can not increase container assignment for an affected stage).

For *Parallel/multi-stage/multi-core with limits*, multiple stages map to a single YARN container with 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 running in parallel, rather than through the file system. The difficulty is that it is hard to predict how many cores are required: structural skew means some stages may under-utilize their core (resulting in internal core-fragmentation), or stages with varying parallelism may overly stress what they have been allocated. Figures (a) and (b) in Figure 13 show that this configuration requires more container-hours compared to other choices, while providing only modest reduction in latency.

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, but here we allow the container to consume cores beyond the container strictly allocates. The challenge is to come up with a right degree of multi-programming for the duration of the pipeline execution. The risk here is that resources are stressed—if we under-provision cores per container, we reduce internal core-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 over-commitment by adapting limits and throttling computation on the fly[13]; this approach is not yet widely available and needs complex feedback loops.)

With *Linearized/single-stage/1-core*, we assign each stage (or two adjacent stages for I/O limited tasks (§2.6)), 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 core-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 skew 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 §4.6 we compare these alternatives, showing that Linearized/single-stage/1-core is the most efficient.

2.8 | Mitigating Computational Skew

Our solution to structural skew (running each stage independently) also addresses computational skew. The challenge of computational skew is that a shift in input data can suddenly change the computation required by a given stage.

To detect computational skew we monitor the amount of data queued at each stage over time (recall that each stage runs separately, with its own queue §2.5). We then reduce the effects of computational skew 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 ensure that no stage is starved of processing.

Another risk of computational skew is that processing for a stage grows so much that stage program times out. Our use of named, 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[9] to our streaming workload.

Plumb's pipeline graph addresses both structural and computational skew with no additional user effort. As new, improved skew solutions evolve, the system can transparently switch and deploy them.

3 | ADDITIONAL APPLICATIONS

In addition to our operational DNS pipeline, Plumb is ideal for other applications. The following applications benefit from our per-block processing abstraction. When they need any data shuffling or data reductions, they use Plumb's API to take data out of a queue, process it, and bring it back in the system. We are extending Plumb with further new abstractions so that users can do powerful reductions in place (see §7 for more details).

Early Detection of Malicious Activity: DNS backscatter can detect scanning and other malicious activity[2]. The input to this workflow is DNS data, and the final output is a list of IPs labeled as malicious (spam, scan, etc.) or benign (CDN, research scan, etc.) activity. The longitudinal studies in that paper employed largely ad-hoc processing with GNU parallel on specific computers, a process that was difficult to implement, slow to run, and often interfered with other users. We are currently adapting this analysis to use Plumb, which will be easier, faster, and automatically share resources over a full cluster.

resource	config. 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. One vCore uses one physical CPU exclusively.

Flow analysis: Several sites at Colorado State U. capture packets and process them to anonymized flows. Input to this workflow is Argus flows, while the final output is monitoring alerts. Converting this post-processing to Plumb will support higher traffic rates. The current system is hand-tuned (hence challenging to evolve) and suffers from throughput issues.

Merging Bidirectional Traffic: Network data capturing on high-speed optical networks often must be done each direction (incoming and outgoing) separately. We want to knit independent captures together after capture to provide a unified view at very high bitrates. The input to this workflow are two TCP flows, one for incoming while the other for outgoing traffic. The output is a single bi-directional TCP flow. Note that, after this stage, our current pipeline workflow can be used as usual (without this step, each subsequent code would need to change to accommodate two separate traffic flows).

Annual Day-In-The-Life of the Internet (DITL) Collection: DITL[14] is an annual data collection event where many service operators come together to collect their service data and share it publicly for facilitating research and development, and understanding evolving trends. B-Root participates in this event regularly. Initial workflow stages of DITL processing work well with our per-block processing abstraction, where we decompress data, consistently anonymize it for privacy preservation, and re-compress for a relay. (We anticipate that future work will add support to identify and support processing over just the DITL time window.)

Shared Datasets inside a Cloud: Plumb can also apply to data processing in the cloud, particularly when multiple groups share parts of the same workflow. These days all major cloud providers share massive datasets for the public use, and many internal and external projects utilize this data. A large cloud provider has internal dataflow between different projects (loosely coupled groups), and they, too, end up shuffling big files around every 5 minutes like we do in B-root. They are fixed time, not fixed duration, but they share all the issues like duplication, skew, etc. Our system applies to such use cases as well.

4 | EVALUATION

We next evaluate how our design choices improve efficiency. We measure efficiency as cluster-container hours, so lower numbers are better. When using 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 to availability of unused cluster hours for other projects.

4.1 | Benefits of de-duplication

We first examine optimizations to eliminate duplicate computation (§2.4) and data storage (§2.5) across multiple users.

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 8, 16 or 24 files, (each 765 MB) and measure total container hours consumed. In a real deployment, new data will always be arriving, and compute cycles will be shared across other applications.

We compare measurements from our DNS pipeline running with related sample data in two configurations: unoptimized and de-duplicated. With the unoptimized configuration all stages run independently, while with de-duplication, identical stages are computed only once (§2.4), with data between each stage buffered in a file (§2.5).

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

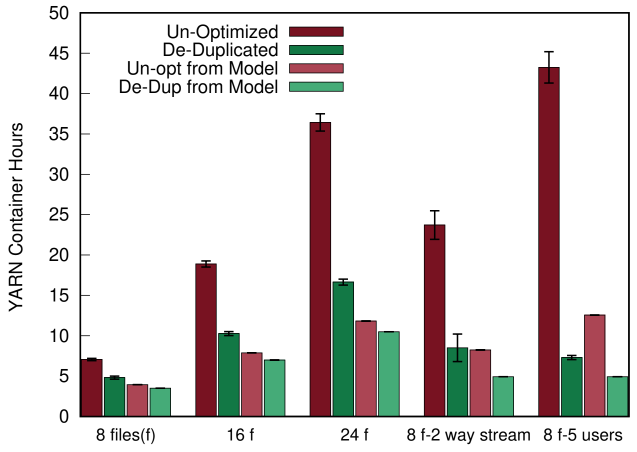

FIGURE 5 Comparing un-optimized (left bar) and de-duplicated (second bar) experimental pipelines and modeled (right two bars) for different workloads (bar groups). Mean of 10 runs with standard deviations as error bars.

can be combined, so an additional duplicated stage (for example, a decryption step) would provide even more savings relative to an unoptimized pipeline.

Experimental Evaluation:

Figure 5 shows our results for three different size inputs (the left three groups of bars), and then for two different pipelines with 8 inputs (the right two groups). The de-duplicated pipeline (the second bar from the left) is always faster than unoptimized (the leftmost bar).

To evaluate the effect of input size we first compare inside each of the left three groups of bars. With 8 files as input, de-duplication uses 39% fewer container hours (compare the first and second bar on the left in each group). 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, prepending an additional shared stage 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 explained in §3). This different configuration is shown in the fourth group of bars from the left of Figure 5, labeled “8 files-2 way stream”. De-duplicated uses about one-third the container-hours 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). In this case, the rightmost set of bars, we see that de-duplication shows a 6× speedup relative to unoptimized, and a much greater savings compared to either of the other two pipeline structures. Adding additional users late in the pipeline exposes significant potential savings.

Modeling De-Duplication: 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. In each group, the right two bars show our model’s prediction for unoptimized and de-duplicated 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 of synchronized file access (a “thundering herd problem”) as we evaluate next. This model confirms our understanding of the underlying speedup.

De-Duplication and Storage:

Like processing, disk usage decreases as we de-duplicate. Figure 6 shows the amount of unique data that is stored over time (data is three-way replicated, so actual disk use is triple) from the experiments of Figure 5. Each point is the mean of ten runs.

In this experiment, de-duplication reduces storage by 55% relative to un-optimized for the 8-file workload (compare the peak of the green squares at time 600 s). Other workloads show similar or greater savings. Storage is particularly high in the middle of each run when the cluster is fully utilized. Greater storage not only consumes storage space, but can result in disk arm contention with disks, and even network contention in clusters that lack a Clos network.

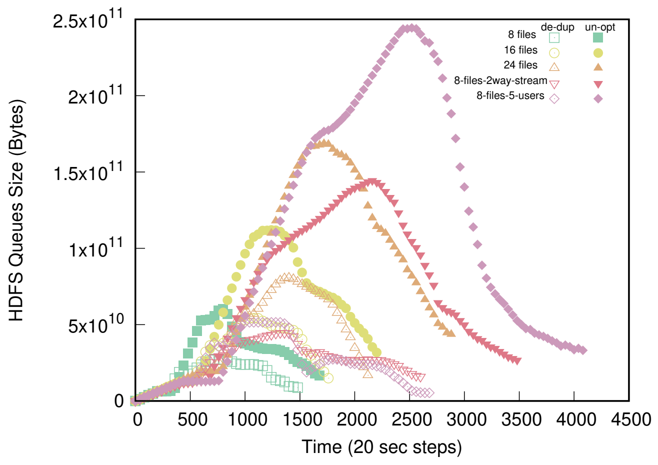

FIGURE 6 Amount of data stored over each processing run for different workloads (color and shape), unoptimized (filled shapes) and de-duplicated (empty shapes).

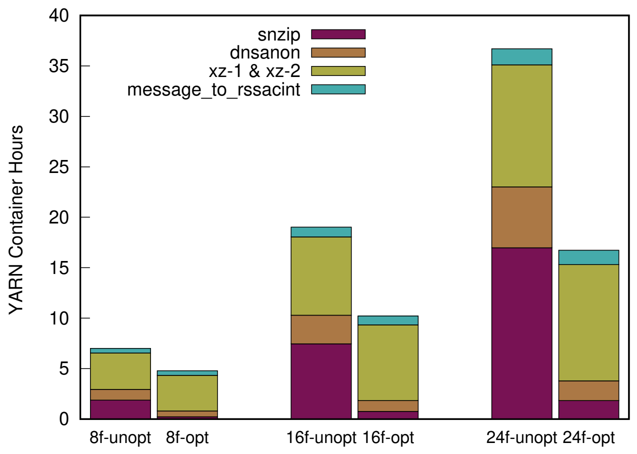

FIGURE 7 Computation spent in each stage (bar colors), unoptimized (left bar) and de-duplicated (right), for three size workloads (bar groups).

This experiment shows latency reduction as well. With fewer required resources, 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). For example, the 8-file-5-user case terminates around 2700 s, with de-duplication, while unoptimized takes about 4100 s, 50% longer (compare the rightmost filled curve against its unfilled pair).

Summary: These experiments show significant duplicate computation can be eliminated in multi-user workloads, and that these savings grow both with additional input data and additional pipeline complexity. These savings reduce costs in the cloud, or latency in a dedicated cluster.

4.2 | 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 disk contention occurs. 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× for the un-optimized case. (SSDs avoid spindle contention, but hard disks are still often used in big data projects where capacity cost can be critical.)

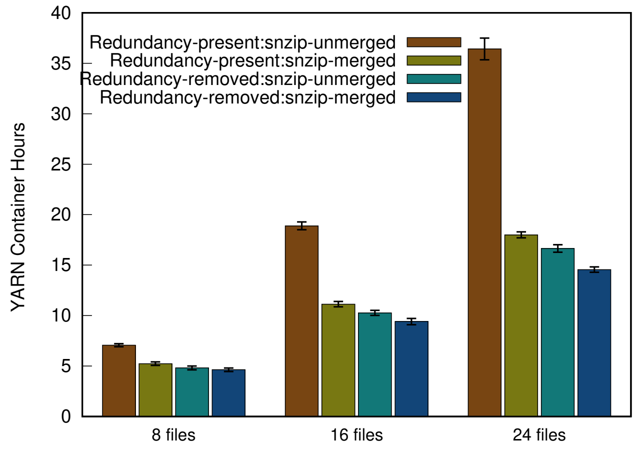

FIGURE 8 Computation with and without I/O merged with duplicated processing (left two bars) and de-duplicated (right two bars) for three size workloads (bar groups).

We first establish that I/O contention can be a problem, then show that merging I/O-intensive stages with CPU-intensive stages (§2.6) 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, snzip consumes many container hours even though it actually requires little computation (Table 1, where it is 6× faster than the next slower stage).

This huge increase in cost for the snzip stage is because it is very I/O intensive (in Table 1, its I/O-intensity is 6 to 100× other stages), reading a file that is about 765 MB and creating a 2 GB output in short amount of time. Without contention, this step takes 28 s, but when multiple concurrent instances are run with 3-way replication 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 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 how throughput improves 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 (including replication); the stages communicate directly through FIFO pipes. We expect that this reduction in I/O will make computation with merged stages more efficient because merged stages will read and write data slower per unit time.

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

Figure 8 shows 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 both reduce I/O, and to lower compute time with less disk contention. Comparing the left two bars (brown and green) of each group shows that this optimization helps a great deal. Stage de-duplication still helps (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 candidate stages to merge.

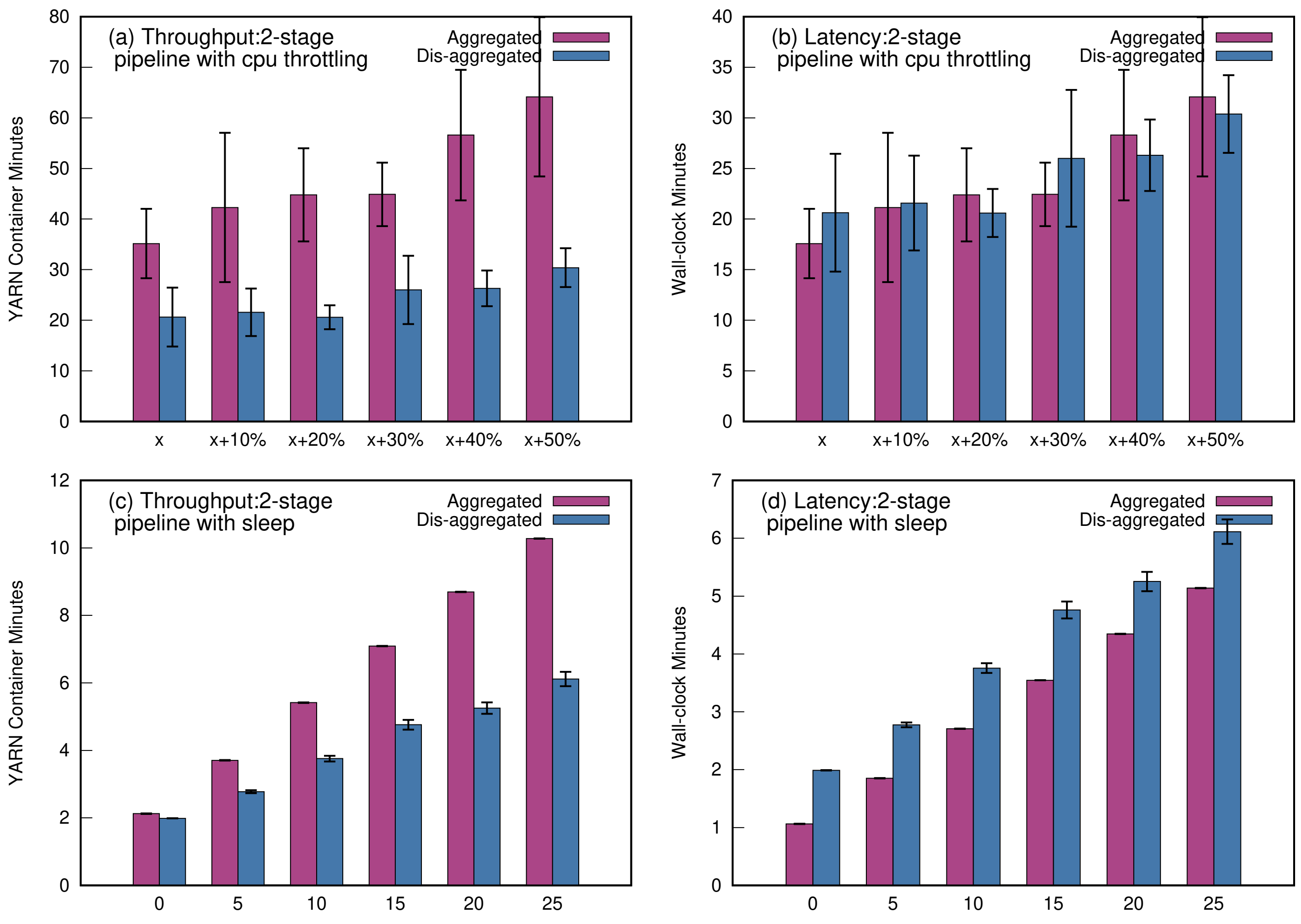

FIGURE 9 Comparing aggregated (left bar) and de-aggregated (right) processing, with delay added by cpulimit (top row) and sleep minutes (bottom row). x is baseline skew between two configurations.

4.3 | Pipeline Disaggregation Addresses Structural Skew

Pipeline disaggregation (§2.7) addresses structural skew by allocating additional workers to slower stages to run in parallel.

Structural skew occurs when two stages have unbalanced run-times and they are forced to run together, allowing progress at only the rate of the slowest stage. Concurrent run of stages is possible due to streaming nature of operators and is often a preferred execution strategy for end-users due to ease of coding (using FIFOs) and belief it providing low latency. Here we first demonstrate the problem, then show how pipeline disaggregation addresses it.

The problem of structural skew: To demonstrate structural skew we use a two-stage pipeline where first stage decompresses snzip-compressed input and re-compresses 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 runs in about 20 minutes, but the second runs in less than 1 minute.)

We compare two pipeline configurations: aggregated and disaggregated. With an aggregated pipeline, both 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 core 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 skew in two ways: first reducing the available CPU in the OS with cpulimit, and by lengthening 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

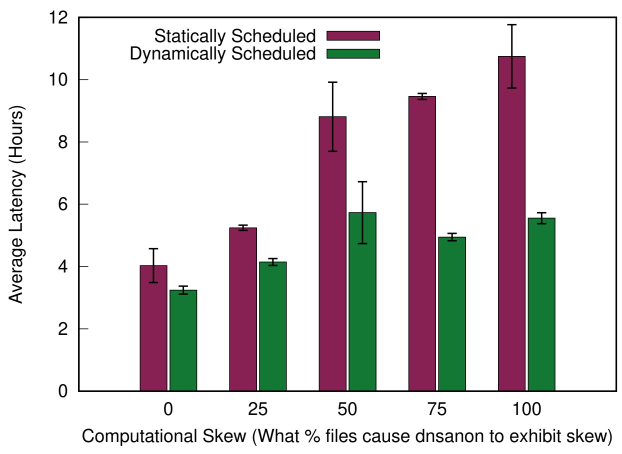

FIGURE 10 Dynamically scheduled workers always beat static assignment and margin increasing with increasing skew. (compare the left bar with the right bar).

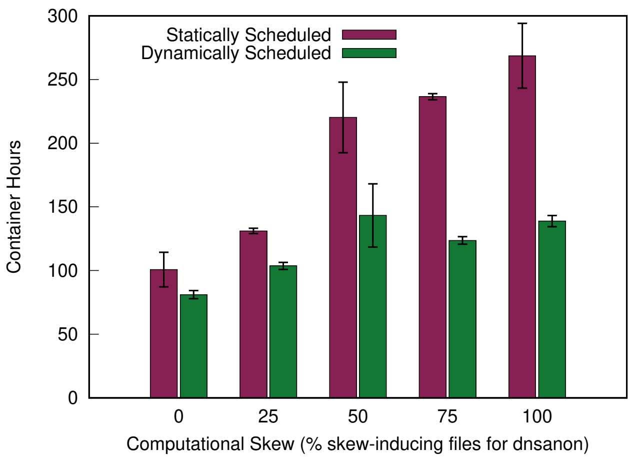

FIGURE 11 As skew increases, static allocation wastes more resources. (compare the left bar with the right bar).

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 some latency (compare the right bar to the left in the bottom graph), but only a fixed amount. This latency reflects queuing intermediate data on disk.

We see generally similar results for both methods of slowdown, except that CPU throttling shows much greater variance. 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 skew, although at the cost of slightly higher latency.

4.4 | Disaggregation Addresses of Computational Skew

Disaggregation is also important to address computational skew. With computational skew, 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 (§2.8). We next demonstrate the importance of this optimization by replaying a scenario drawn from our test application.

We encountered computational skew 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 of the 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 10 shows latency in this experiment. (Throughput, measured by container hours, is similar in shape as shown in Figure 11.) The strong result is that dynamic scheduling greatly reduces latency as computational skew 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 computational skew (the fraction of DDoS input files). Each row of graphs shows one aspect 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 skew increases.

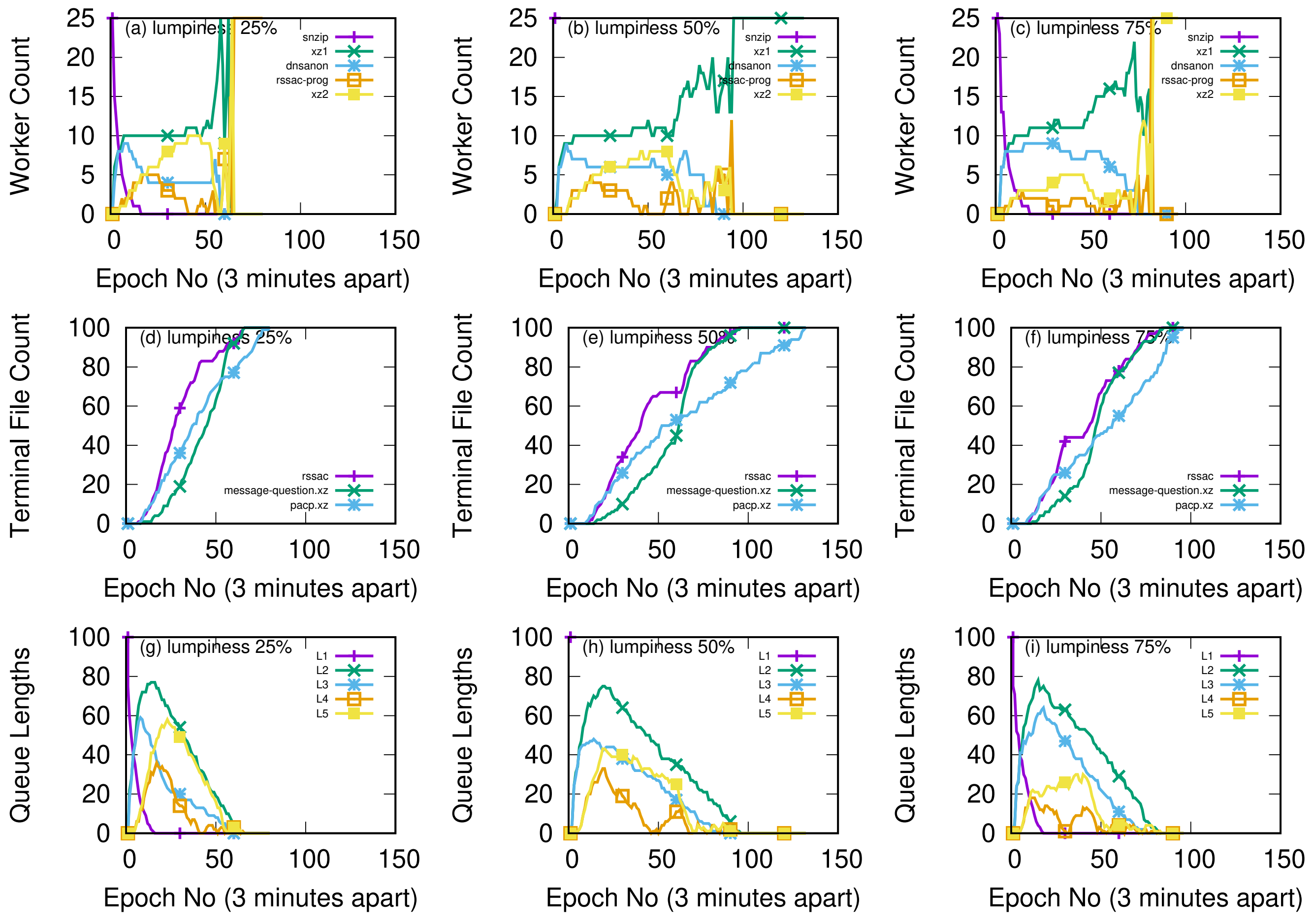

FIGURE 12 Dynamic scheduling at 0%, 50% and 100% skew.

4.5 | 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 cannot directly compare our system with streaming systems like Spark for several reasons. Our LBS workloads don't map to Spark's distributed shared memory (RDD) model, our working set changes rapidly (hence not benefiting from in-memory cache), and Spark has no analog for our multi-user data and code de-deuplication.)

We use data from two full days of B-Root DNS: 2016-06-22, a typical day with 1.8 TB of data in 896 files, and 2016-06-28, a day when there was a large DDoS attack[15]. Because of network rate-limiting during the attack, the second day has *less* data, with about 1.4 TB of traffic in 711 files. To account for this shortfall, we construct a synthetic attack day where we replicate one attack input 896 times to get the same traffic volume as the normal day, 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 fast and has minimal effect on the workload.)

We process this data on our compute cluster with a fixed 100 containers, while using all optimizations (de-duplication and skew mitigation) and I/O-bound stage 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 without limiting 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 core-fragmentation: we must over-size

scenario	date	input	latency
normal day	2016-06-22	1.8 TB / 896 files	8.30h
attack day	2016-06-25	1.4 TB / 711 files	6.20h
simulated attack	—	1.8 TB / 896 files	11.75h

TABLE 4 Plumb latency:One day DNS data using 100 cores

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 day of DDoS attacks was processed faster than the typical day (6.2h vs. 8.3h), but the savings follows from less saved data. This drop in traffic is due to a link-layer problem with B-Root 's upstream provider where we were throttling attack traffic before collection, 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. We discuss deployment status in §5.

4.6 | Comparing Design Alternatives for Stages per Container

In §2.7 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 core-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 hand-built pipeline. 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 under-utilization 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.

4.7 | Improving End-to-End Latency

Plumb's goal is to minimize latency of streaming block data. To evaluate latency, we compared end-to-end latency for Plumb and our prior Hadoop-based system over 24 hours of data (1393 files, each 2 GB) with our DNS workflow (Figure 1) in Figure 14. Current Plumb latency (left) is *much* lower with median latency of 695 s, instead of 2724 s for RSSAC files. In addition, Plumb latency is much more *consistent*, with standard deviation of 50 s instead of 614 s (compare the narrow range of Plumb against the wide range of batch). Plumb latency is lower because it processes blocks as they arrive, rather than batching them. Plumb latency is good at about twice the theoretical minimums (see Figure 1), but it has some room for optimization.

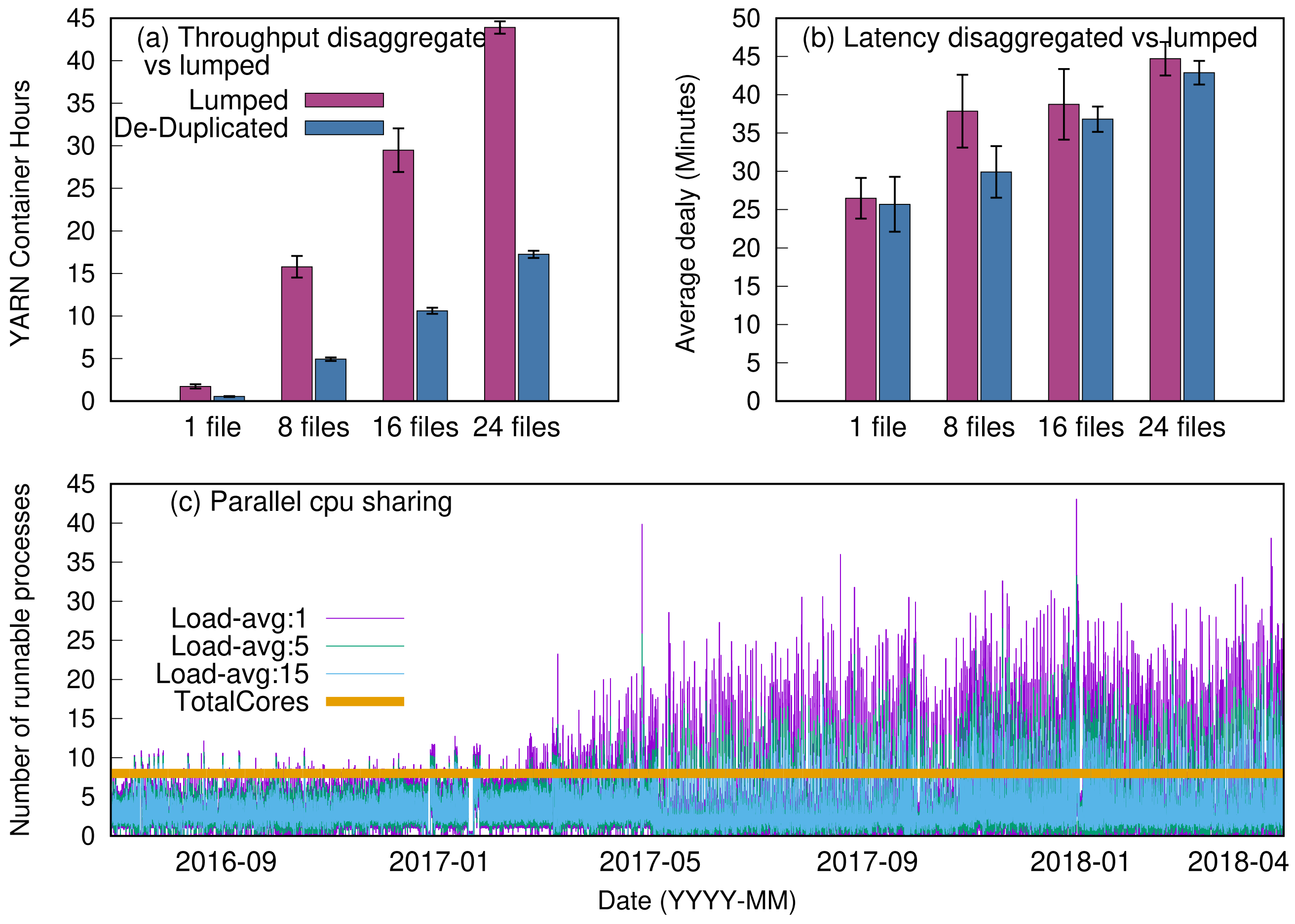

FIGURE 13 Comparing alternative configurations of stages per container. Parallel/multi-stage/multi-core with limits marked as lumped. The graph in the bottom row is from one server, while the others in the cluster are similar.

5 | DEPLOYMENT STATUS

Plumb is currently in production use at B-Root for DNS analysis. Plumb keeps up with real-time processing with fewer resources than our prior system, and with much lower latency per-file. At steady state, Plumb's queue is about 20 files deep, while our prior system varied from 50 to 150 over the day, and overall loads are much more consistent without over- or under-utilization. Plumb's deployment has prompted new applications for multiple users to run on Plumb infrastructure.

6 | 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, de-duplicating arbitrarily complex code and data, and coping with classes of performance problems (structural and computational skew) in a framework.

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[16]) we only use workflow to capture data flow dependency to facilitate de-duplication. Like some big-data workflow systems[17], our stage programs can bring together computation from different systems (perhaps MapReduce and Spark) into one workflow; but with the benefits of de-duplication. Other systems place greater constraints on components (such as[18]), allowing component-specific optimizations (for example, joins); our model avoids this level of integration to allow users to work in

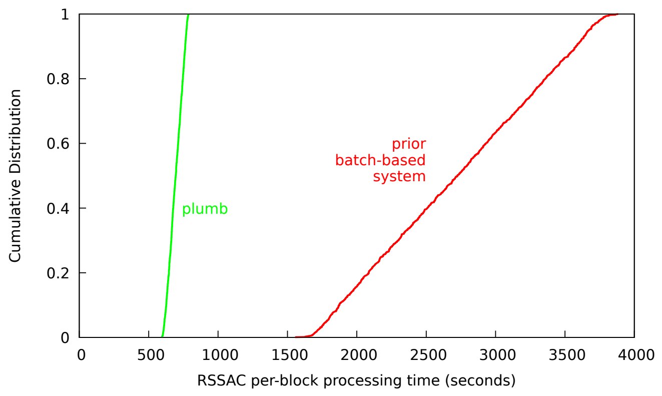

FIGURE 14 Latency:Plumb VS batch-based processing.

different languages and frameworks. We consider operators as black box and data as an opaque binary stream with unique naming scheme to find and unlock data and processing duplication. We use Apache YARN[11] for scheduling.

Batch systems: Batch systems, such as MapReduce[9] and Dryad[19] 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[20]. 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: Several systems focus on low-latency processing of small, streaming data, including Kafka[21], MillWheel[22] and Spark Streaming[23], but without multi-user optimization. They often strive to provide transaction-like-semantics and exactly-once evaluation, and stream data in small pieces to minimize latency. These systems focus on processing small objects (records), while we instead focus on large-block streaming. Large-blocks are critical to our application and similar applications that need a broader view than a single record provides, and because large blocks provide clean accountability for fault-tolerance and completeness (each block is processed or not) and more comfortable debugging (for example manually examining few large-blocks as compared to millions of records). Larger blocks of data raise issues of structural and computational skew that differ from systems with fine-grain processing; we address skew with specific optimizations.

In addition to different abstractions, systems like Spark streaming make assumptions about how much data fits into memory and how quickly the working-set changes. Our applications generally make one pass over data that far exceeds memory, making Spark's optimizations ineffective.

Streaming systems like Flink[24] optimize for throughput or latency by configuration; we focus on throughput while considering latency as a secondary goal.

Programming-language integration: Several programming languages provide abstractions and optimizations for big-data processing (for example, [25,26]). These systems often optimize a specific job but not those of multiple users; we optimize multi-user workloads. Compilers can easily match an I/O pattern to a suitable optimization when framework enforces structure on I/O consumption[27]. SQL-compilers for declarative languages[28,29] and Parallel databases[30] match language abstractions to database access patterns and optimizations. Multi-query optimization[31,32] tries to find data sharing in compiled plans of relational data.

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

Big-data schedulers: Different schedulers have been proposed to optimize resource consumption or delay[33,34], or to enhance data locality[35]. We work with existing schedulers, optimizing inside our framework to mitigate skew.

Stragglers: Straggler handling is a special case of what we call skew, with several prior solutions. Speculative execution[9] recovers from general faults. Static sampling of data[36], predictive performance models[37], dynamic key-range adjustment[19], and aggressive data division[38] seek to detect or address computational skew. These systems are often optimized around specific data types or computation models and assume structured data. Our system can be thought of as an approach to addressing this problem while making very few assumptions about the underlying data. The large-block data consumption need of our applications preclude any adaptive data-sharding schemes.

Resource optimization for high throughput: Several systems exploit close control and custom scheduling of cluster I/O[39] or memory[40] 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[23]. While we consider sharing across branches of a multi-user pipeline, our workloads are streaming and so are not amenable to caching.

Data de-duplication: Many systems rely on consistent hashing of data for duplication detection[41,42] in shared data repositories (like file servers) and multi-user computation. The fundamental shortcoming is that they generate duplicate data nonetheless (albeit deduplicated afterward). For streaming workloads, such a scheme will cause un-necessary network traffic (due to inline three-way replication) and disk bandwidth use. We detect all duplication at the user DAG optimization time, and never generate duplicate data.

The Sed-Dedup system[43] allows modification in the data and uses delta de-duplication to reduce storage. We do not allow direct data modification (processing writes a new data block after consuming input). Such a delta duplication scheme is more applicable to long-term data archives than a streaming system.

The Edison-Data project[44] uses a data de-duplication algorithm similar to ours. System can label data for different scientific domains after semantic analyses. That implies that same data gets similar labels from the system, and hence marked as a possible duplicate (though removal of duplicates done manually by end-users). End users can search the repository based on these system-generated labels. As a consequence, de-duplication in Edison-Data depends on programmer's manually searching from the available library. They do not manage workflow skew as well. We automatically infer code similarity from the input/output data types and manage computational and structural skew in the workflow.

The SDAM system[45] uses programmer-generated annotations to detect common parts of the code to optimize their in-memory, small-record workloads. Our system is similar to SDAM annotations where our de-duplication strategy depends on domain-specific naming. Though Plumb's names being explicit in user DAGs is much simpler than careful code annotation, and more robust to incorrect naming (a shared, optimized DAG is easier to review and to catch naming problems).

Multi-user systems: Like our work, Task Fusion[46] 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 computational skew. The Nectar system[7] examines work de-duplication, as we do, but we also expose data to users to promote data de-duplication and encourage data sharing. We also address the problem of workload skew. Several systems suggest frameworks or libraries to improve cluster sharing and utilization[47]. Some of them resemble our optimized pipeline, but we focus on a very simple streaming API and loosely coupled jobs.

7 | FUTURE WORK

We are extending Plumb's ability to allow multiple abstractions in a workflow. Some of our users need sophisticated reductions that operate on more than one data block at a time, typically to align data with a fixed time window (hours of the day, or a 24-hour period). Currently, they utilize Plumb's API to extract data out of a queue, process it with their desired system (for example, MapReduce), and feed the resulting data as large blocks back into the Plumb. Work is currently under progress to allow users to utilize other abstractions in place on the variable amount of data, either in terms of the number of blocks or time duration.

8 | CONCLUSIONS

Plumb is designed for processing large-block, streaming data in a multi-user environment. Plumb's novelty comes from integrating workflows from multiple users while de-duplicating computation and storage, and its use of dynamic scheduling to accommodate structural and computational skew.

Plumb is in production use today handling all B-Root DNS data, and we are in the process of deploying it for additional workflows. Plumb provides lower latency with less resources compared to our prior system, while its ease-of-use supports new users and analysis. We believe Plumb can apply to several similar workflows.

Plumb shows how the right combination of simple ideas (pipe-graph and LBS abstractions, use of naming) can solve challenging problems (multi-user duplication and skew). Plumb's abstractions provide a shared (but secure) canvas to its users to collaboratively, but not requiring their simultaneous presence, use available data with less delay and high resource utilization.

9 | ACKNOWLEDGMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-18-2-0280 and contract number W911NF-17-C-0011 with the U.S. Army Contracting Command–Aberdeen Proving Ground (ACC-APG) and the Defense Advanced Research Projects Agency (DARPA). The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of ACC-APG, DARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

We thank Yuri Pradkin at Information Sciences Institute for helping us factor out a stable queuing API using his experiences from the LANDER system. We also thank Wes Hardaker as an early user of our system.

References

1. B-Root DNS Statistics (https://b.root-servers.org/rssac/). website; .

2. Fukuda K, Heidemann J, Qadeer A. Detecting Malicious Activity With DNS Backscatter Over Time. *IEEE/ACM Trans. Netw.* 2017; 25(5): 3203–3218. doi: 10.1109/TNET.2017.2724506

3. Toshniwal A, Taneja S, Shukla A, et al. Storm@twitter. In: SIGMOD '14. Association for Computing Machinery; 2014; New York, NY, USA: 147–156

4. Schlaipfer M, Rajan K, Lal A, Samak M. Optimizing Big-Data Queries Using Program Synthesis. In: SOSP '17. Association for Computing Machinery; 2017; New York, NY, USA: 631–646

5. Chu S, Murphy B, Roesch J, Cheung A, Suciu D. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 2018; 11(11): 1482–1495. doi: 10.14778/3236187.3236200

6. Jindal A, Qiao S, Patel H, et al. Computation Reuse in Analytics Job Service at Microsoft. In: SIGMOD '18. Association for Computing Machinery; 2018; New York, NY, USA: 191–203

7. Gunda PK, Ravindranath L, Thekkath CA, Yu Y, Zhuang L. Nectar: Automatic Management of Data and Computation in Datacenters. In: OSDI'10. USENIX Association; 2010; USA: 75–88.

8. Coppa E, Finocchi I. On Data Skewness, Stragglers, and MapReduce Progress Indicators. In: SoCC '15. Association for Computing Machinery; 2015; New York, NY, USA: 139–152

9. Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. In: OSDI'04. USENIX Association; 2004; Berkeley, CA, USA: 10–10.

10. Sipser M. *Introduction to the Theory of Computation, Theorem 5.4 EQ_{TM} is undecidable.* Boston, MA: Course Technology. third ed. 2013.

11. Vavilapalli VK, Murthy AC, Douglas C, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In: SOCC '13. ACM. Association for Computing Machinery; 2013; New York, NY, USA

12. Fox A, Gribble SD, Chawathe Y, Brewer EA, Gauthier P. Cluster-Based Scalable Network Services. In: . volume 31 of *SIGOPS Oper. Syst. Rev.* Association for Computing Machinery; 1997; New York, NY, USA: 78–91

13. Zhang X, Tune E, Hagmann R, Jnagal R, Gokhale V, Wilkes J. CPI2: CPU Performance Isolation for Shared Compute Clusters. In: EuroSys '13. Association for Computing Machinery; 2013; New York, NY, USA: 379–391

14. DNS-OARC . Day In the Life of the Internet (DITL) 2014. https://www.dns-oarc.net/oarc/data/ditl; 2014.

15. Root Server Operators . Events of 2016-06-25. tech. rep., Root Server Operators; ISI: 2016.

16. Deelman E, Singh G, Su MH, et al. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Sci. Program.* 2005; 13(3): 219–237. doi: 10.1155/2005/128026

17. Islam M, Huang AK, Battisha M, et al. Oozie: Towards a Scalable Workflow Management System for Hadoop. In: SWEET '12. Association for Computing Machinery; 2012; New York, NY, USA

18. Murray DG, McSherry F, Isaacs R, Isard M, Barham P, Abadi M. Naiad: A Timely Dataflow System. In: SOSP '13. Association for Computing Machinery; 2013; New York, NY, USA: 439–455

19. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In: . 41. ACM. Association for Computing Machinery; 2007; New York, NY, USA: 59–72

20. Dennison D. Continuous Pipelines at Google. *SRECon* 2015.

21. Kreps J, Corp L, Narkhede N, Rao J, Corp L. Kafka: A distributed messaging system for log processing. *Proceedings of the NetDB* 2011: 1–7.

22. Akidau T, Balikov A, Bekiroundefinedlu K, et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In: . 6. VLDB Endowment; 2013: 1033–1044

23. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In: SOSP '13. ACM. Association for Computing Machinery; 2013; New York, NY, USA: 423–438

24. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache Flink^{TM}: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 2015; 38: 28-38.

25. Yu Y, Isard M, Fetterly D, et al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In: OSDI'08. USENIX Association; 2008; USA: 1–14.

26. Chambers C, Raniwala A, Perry F, et al. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In: PLDI '10. Association for Computing Machinery; 2010; New York, NY, USA: 363–375

27. Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 2014; 46(4). doi: 10.1145/2528412

28. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig Latin: A Not-so-Foreign Language for Data Processing. In: SIGMOD '08. Association for Computing Machinery; 2008; New York, NY, USA: 1099–1110

29. Pike R, Dorward S, Griesemer R, Quinlan S. Interpreting the Data: Parallel Analysis with Sawzall. *Sci. Program.* 2005; 13(4): 277–298. doi: 10.1155/2005/962135

30. Pavlo A, Paulson E, Rasin A, et al. A Comparison of Approaches to Large-Scale Data Analysis. In: SIGMOD '09. Association for Computing Machinery; 2009; New York, NY, USA: 165–178

31. Sellis TK. Multiple-Query Optimization. *ACM Trans. Database Syst.* 1988; 13(1): 23–52. doi: 10.1145/42201.42203

32. O'Gorman K, Abbadi AE, Agrawal D. Multiple query optimization in middleware using query teamwork. *Wiley Journal of Software, practice & experience.* 2005; 35(4): 361-391. doi: 10.1002/spe.640

33. Ehsan M, Sion R. LiPS: A Cost-Efficient Data and Task Co-Scheduler for MapReduce. *IPDPS workshop* 2013: 2230–2233. doi: 10.1109/IPDPSW.2013.175

34. Delgado P, Dinu F, Kermarrec AM, Zwaenepoel W. Hawk: Hybrid Datacenter Scheduling. In: USENIX ATC '15. USENIX Association; 2015; USA: 499–510.

35. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In: EuroSys '10. ACM. Association for Computing Machinery; 2010; New York, NY, USA: 265–278

36. Venkataraman S, Yang Z, Franklin M, Recht B, Stoica I. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In: NSDI'16. USENIX Association; 2016; USA: 363–378.

37. Morton K, Balazinska M, Grossman D. ParaTimer: A Progress Indicator for MapReduce DAGs. In: SIGMOD '10. Association for Computing Machinery; 2010; New York, NY, USA: 507–518

38. Ousterhout K, Panda A, Rosen J, et al. The Case for Tiny Tasks in Compute Clusters. In: HotOS'13. USENIX Association; 2013; USA: 14.

39. Rasmussen A, Lam VT, Conley M, Porter G, Kapoor R, Vahdat A. Themis: An I/O-Efficient MapReduce. In: SoCC '12. Association for Computing Machinery; 2012; New York, NY, USA

40. Li H, Ghodsi A, Zaharia M, Shenker S, Stoica I. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In: SOCC '14. ACM. Association for Computing Machinery; 2014; New York, NY, USA: 1–15

41. Periasamy J, Latha B. Efficient hash function-based duplication detection algorithm for data Deduplication deduction and reduction. *Wiley Journal of Concurrency and computation.* 2019; e5213(2): 1-9. doi: 10.1002/cpe.5213

42. Ogata M, Komoda N. Improvement of Deduplication Efficiency by Two-Layer Deduplication System. *Wiley Periodicals of Electronics and Communications in Japan.* 2016; e11781(2): 1-9. doi: 10.1002/ecj.11781

43. Tian W, Li R, Xu CZ, Xu Z. Sed-Dedup: An efficient secure deduplication system with data modifications. *Wiley Journal of Concurrency and computation.* 2019; e5350(4): 1-14. doi: 10.1002/cpe.5350

44. Ahn S, Lee J, Kim J, Lee JR. EDISON-DATA: A flexible and extensible platform for processing and analysis of computational science data. *Wiley Journal of Software, practice & experience.* 2019; 49(10): 1509-1530. doi: 10.1002/spe.2732

45. Cappellari P, Roantree M, Chun SA. Optimizing data stream processing for large-scale applications. *Wiley Journal of Software, practice & experience.* 2018; 48(9): 1607-1641. doi: 10.1002/spe.2596

46. Dyer R. Task Fusion: Improving Utilization of Multi-user Clusters. In: SPLASH '13. ACM; 2013; New York, NY, USA: 117–118

47. Palkar S, Thomas JJ, Shanbhag A, et al. Weld: A Common Runtime for High Performance Data Analytics. In: CIDR '17. ; 2017.

AUTHOR BIOGRAPHY

Abdul Qadeer is a PhD candidate at University of Southern California, Los Angeles, USA. He works with Analysis of Network Traffic group at USC's Information Sciences Institute, Marina Del Rey. His broad research interests include distributed systems, computer networks, operating systems, and computer architecture.

John Heidemann is Research Professor and Principal Scientist at USC / ISI, Marina Del Rey, California, USA. He received his PhD in 1995 at University of California, Los Angeles. His research interests include observing and analyzing Internet topology and traffic to improve network reliability, security, protocols, and critical services.

How to cite this article: Abdul Qadeer, and John Heidemann, Plumb: Efficient Stream Processing of Multi-User Pipelines, *Journal of Software: Practice and Experience*, *2020;50:1–24*.