Efficient Processing of Streaming Data using Multiple Abstractions

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

***Abstract*—Large websites and distributed systems employ sophisticated analytics to evaluate successes to celebrate and problems to be addressed. As analytics grow, different teams often require different frameworks, with dozens of packages supporting with streaming and batch processing, SQL and no-SQL. Bringing multiple frameworks to bear on a large, changing dataset often create challenges where data transitions—these impedance mismatches can create brittle glue logic and performance problems that consume developer time. We propose Plumb, a meta-framework that can bridge three different abstractions to meet the needs of a large class of applications in a common workflow. *Large-block streaming* (Block-Streaming) is suitable for single-pass applications that care about the temporal and spatial locality. *Windowed-Streaming* allows applications to process a group of data and many reductions. *Stateful-Streaming* enables applications to keep a long-term state and always-on behavior. We show that it is possible to bridge abstractions, with a common, high-level workflow specification, while the system transitions data batch processing and block- and record-level streaming as required. The challenge in bridging abstractions is to minimize latency while allowing applications to select between sequential and parallel operation, while handling out-of-order data delivery, component failures, and providing clear semantics in the face of missing data. We demonstrate these abstractions evaluating a 10-stage workflow of DNS analytics that has been in production use with Plumb for 2 years, comparing to a brittle hand-built system that has run for more than 3 years.**

***Index Terms*—programming model, abstraction, big-data abstraction, large-block streaming, programmer productivity**

I. Introduction

Services like websites, web crawling, on-line advertising, advertising action systems, and the Domain Name System (DNS) generate continuous data streams of user and system activity. Analysis of this activity provides business intelligence, performance analysis, intrusion detection, and security monitoring that helps monetize, optimize, and secure these services. Such analytics often employ multi-step workflows, and mature systems will have analytic components that are developed by different teams, requiring multiple, different software frameworks to provide efficient, effective, and timely results.

Large workflows often require multiple analytic frameworks. A recent survey showed that, on average, enterprise data pipelines use seven different tools in a workflow [11], sometimes employing custom integration methods to bridge frameworks [16]. There are three reasons applications benefit from using multiple frameworks—ease-of-coding, efficiency, and working with the existing codebase.

When the framework matches the problem, development will be easier, with shorter and less complicated code. As an example, employing a complex algorithm like DNS TCP reassembly in a packet trace is easier as a MapReduce [12] job than in Hive [32] job, since TCP semantics do not map to SQL. Such ease of expression increases code velocity and programmer efficiency.

Second, framework choice often affects performance. Some frameworks are optimized for throughput (data processed per hour per server), while others emphasize low latency (data in to results out). As an example, Spark [34] is a better fit for workloads that carry out multiple steps over the same data, Flink is optimized streams of small records [8], and MapReduce is optimized for single-pass, batch processing.

Third, organizations often have large, long-lived bodies of code that solve their specific problems. Integrating existing code with big-data processing is important to leverage this investment. An example is a serial toolchain for network data analyses that cannot be easily rewritten for better parallelism but still needs to adapt to a cluster environment.

Our goal is to support migration between multiple abstractions with a common streaming workflow such that processing is correct, efficient, and easy-to-use. While developers can sometimes shoehorn one abstraction into another system, a mismatch often compromises correctness when handling the many corner cases that arise in an operational system, and it can be difficult to provide good performance as data moves between abstractions. Finally, developers must implement special-purpose code to bridge abstractions and address correctness and performance needs, adding development time.

We see three different abstractions are important to cover a broad range of big data analytics. We identify abstractions of large-block streaming (Block-Streaming), windowed large-block streaming (Windowed-Streaming), and continuous streaming (Stateful-Streaming) to meet our goal. These abstractions cover a range of tasks in our case study, a DNS processing and analytics workflow (§II-A), as well as other applications we have considered.

With *Block-Streaming*, large data-blocks (often 0.2 to 2 GB in size) are sent to the user processing functions that emit new large blocks as output. When blocks can be processed out-of-order, Block-Streaming can exploit easy parallelism by processing blocks as they arrive, in parallel, across multiple cores or machines. In addition to supporting parallelism, Block-Streaming simplifies error handling, since the processing status

of each block can be tracked and when processing fails for a block, it can be automatically retried. This abstraction support applications with single-pass ingestion, leveraging temporal or spatial locality of data, and with block-level parallelism. We have found Block-Streaming is a good fit for many aspects of statistical analysis of network traffic such as DNS. For example, DNS queries have short-lived TCP flows, with different TCP segments of the same flow are near in time. Our Block-Streaming abstraction can exploit temporal and spatial locality during TCP reassembly, where MapReduce would require much larger I/O because it shuffles all data due to grouping by 5-tuple, and a Spark-based system would require memory to cache the entire dataset.

Some applications require processing a fixed window of data—often a particular time period (a day or an hour), or a large amount of data. For these applications, we see *Windowed-Streaming* as a second important abstraction. *Windowed-Streaming* enforces time ordering on blocks and provides a window of such data to developer for processing, that can then operate on the entire window in one workflow step. Error handling in Windowed-Streaming is more complicated than with Block-Streaming, since blocks may arrive late, violating processing time constraints, and if blocks are lost, a window may never be complete. A Windowed-Streaming can automate error handling, providing common methods to handle completeness and timeouts based on the developer's requirements. We have found windowed processing is necessary to match the requirement that reporting occur at regular times (perhaps daily), while bulk data is more efficiently handled with fixed-size Block-Streaming. Periodic MapReduce analysis often assumes a time-based window of input data.

Finally, some applications require tracking state overall time. Such applications require all data arrive, in-order. *Stateful-Streaming* streams blocks with lower latency than Block-Streaming to a specific instance of an application that keeps long-running state. Rather than running to completion, this application runs continuously, consuming data and tracking long-running state. Application such as intrusion detection often benefit from such state.

Prior systems support each of these abstractions, but typically provide only *one* abstraction. For example, Spark [35] and Kafka [23] support record-level streaming, MapReduce [12] works well for periodic analysis of windowed data (but leaves windows management to the developer), and most network tools such as intrusion detection systems (IDS) [26] expect to work directly on an infinite stream of data.

The contribution of this paper is to show that *one* framework can support *multiple classes of applications*—we can efficiently bridge data between block-level streaming, time- or data-based windows, and also support stateful-streaming when required. The challenge in reaching this goal is to provide good performance as data moves between very different formats (large blocks of MB or GB, very large windows of hours or days and many GB or TB, and continuous streams), while providing clear error semantics and a simple programming workflow. We demonstrate three abstractions: Block-Streaming, Windowed-Streaming, and Stateful-Streaming. We demonstrate this approach in a system that has been operational for over 5 years (initially with custom workflow, now with our system) and has processed more than 12 PB of DNS packet data (when uncompressed), daily statistics for 5 years, and is now being used for streaming intrusion detection. Plumb has cut per-block latency by 74% (§III-A) and daily statistics by 97% (§III-B3), while reducing code size by 58% (§III-B1) and lowering operator intervention to handle problems by 73% (§III-B2).

Plumb is open source and available for download from our website [27].

II. System Design

Plumb make it easy to move data between three abstractions: Block-Streaming, Windowed-Streaming, and Stateful-Streaming to support a large class of applications. In this section we describe the goals of our system and these abstractions, illustrated with a DNS as a case study that we describe first.

A. Case Study: A DNS Workflow

We use DNS (the domain name system that maps human-readable names to machine-friendly addresses) as a case study to show the need for multiple frameworks in a real workflow. Figure 1 shows the workflow, with most stages requiring Block-Streaming, but the two on the right requiring our other two abstractions. Versions of this workflow have been in use since 2015, so it provides a good example of how we manage workflow with and without multiple abstractions.

The main goal of this workflow is to compute daily statistics and archive trace information in two formats, but it has grown to include different kinds of intrusion detection as well. This workflow has evolved over 5 years, and different individuals are responsible for different components.

DNS data from geographically distributed B-Root sites arrive at our processing cluster (A in Figure 1). Initial few steps of processing need Block-Streaming based processing to harness DNS flow's temporal and spacial locality (marked as steps B, C and F in Figure 1). Blocks are processed into rssacint-format [18] (C in the figure), a summary with information needed for daily statistics. We accumulate a 24-hour window of rssacint files to drive daily statistics (from step C to D). Currently this step is done with custom code to bridge Block-Streaming abstraction to MapReduce; we plan to move to Windowed-Streaming, as we evaluate in our prototype in §III-B.

Applications like network intrusion detection need to run continuously to consume streaming data, keeping long-term state, and detecting and alerting about any interesting events. Workflow steps A to E in Figure 1 depict one such application where our Stateful-Streaming abstraction makes it easy to build the application with low delay. Our B-Root DNS infrastructure often comes under malicious attacks, and our early studies show that there is often a reconnaissance pre-activity before the actual attack [17]. To allow reaction to

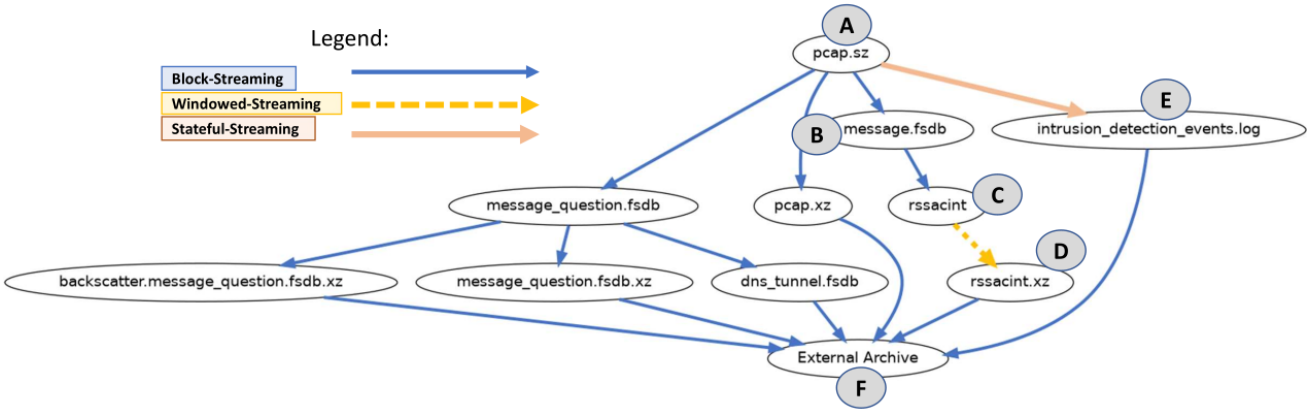

Fig. 1: The DNS processing pipeline, our case study described in §II-A. Intermediate and final data are ovals, computation occurs on each arcs. Different steps use one of our abstractions to achieve diverse processing goals.

security problems, intrusion detection must run as quickly as possible, in near-real-time.

B. Plumb Goals and Requirements

The primary goal of our work is to enable developers to process data correctly in an efficient, and flexible way, applying the best applicable abstraction in each step of the workflow.

We improve *correctness* by avoiding ad-hoc code that is otherwise required to bridge abstractions. In our pre-Plumb system (a hand-tuned but often a brittle system), we required significant code to confirm that a window is complete before processing when going from Block-Streaming to Windowed-Streaming. Our pre-Plumb system is an example where simple solutions were deployed to emulate streaming. It also show the challenges without framework support for streaming: incomplete information prevents the pre-Plumb system from handling corner-cases such as detecting missing data and deduplicating work. With Plumb-managed windowing, system administrators can improve algorithms over time without any change in the developer code.

Our second goal is to provide *efficient* processing by reducing latency. Custom developer code either redundantly exercise large amount of data and processing to achieve lower delay or sacrifice low latency by running windowing algorithms after long intervals. In §III-B3 we show that Plumb reduces latency of a site by 97%.

Ease-of-use is also an explicit goal. Our developers can easily specify their processing needs using one of our abstractions in a YAML job description, and our framework takes care of all the data transport, accumulation, scheduling, and fault details. Developer don't need to employ custom cronjobs to schedule their jobs. In §III-B1 we show that Plumb reduces code by 58%.

Figure 2 shows logical architecture of our system. Developer applications use one or more of our abstractions in their workflow. Three abstraction of Block-Streaming, Windowed-Streaming, and Stateful-Streaming are our approach to achieve

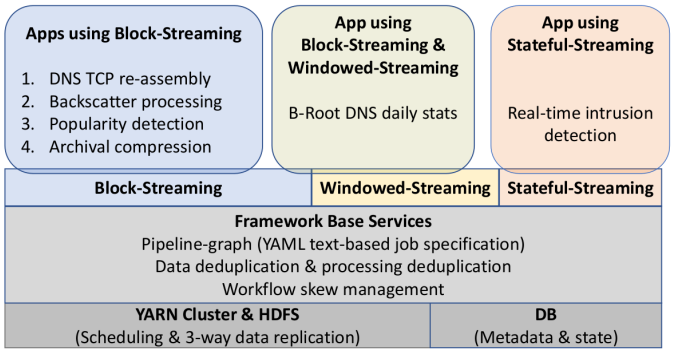

Fig. 2: System architecture.

```
# following 3 stages via Block-Streaming
-
        input:     pcap.sz
        program:   "/user/aqadeer/jobserver/user-progs/snzip_merged.sh"
        output:    pcap.xz
-
        input:     pcap.sz
        program:   "/user/aqadeer/jobserver/user-progs/dnsanon_wrapper_snzip.sh"
        output:    [ message_question.fsdb, message.fsdb ]

-
        input:     message.fsdb
        program:   "/user/aqadeer/jobserver/user-progs/rssac_wrapper.sh"
        output:    rssacint
# following stage via Windowed-Streaming
-
        input:     rssacint
        program:   "/mapreduce_wrapper.sh"
        output:    rssac_mapreduce_stage1
        window:
              size:                                "24 hours"
              start:                               "2020-04-01-00-00-UTC"
              max_wait:                            "1 hours"
              cutoff_delivery_requirement:         "Cutoff_Exactly_Once_To_Current_Window"
              use_queue_sequence_number_instance:  "Yes"
              queue_sequence_number_instance_list: "lax,mia,ari,ams,iad,sin"
# following stage via Stateful-Streaming
-
        input:    pcap.sz
        program:  "zeek_wrapper.sh"
        output:   [weird.log, notice.log]
        abstraction: "stateful_streaming"
```

Fig. 3: Three abstraction in use in the DNS workflow.

above mentioned requirements. In the following we explain our abstractions in detail. Table I is a summary of them as they compare with each other.

C. Large-Block Streaming

The Large-Block Streaming abstraction divides the stream data into fixed-sized blocks for processing. Our network

	Unit of consumption	Processing order	Call-back function	I/O size control	State	Canonical usecase
Block-Streaming	1 full block	None	-Read 1 or more blocks -Write one or more blocks -Run to completion in bounded time	No direct control	No except emit in output block(s)	-Single pass apps -Temporal and spacial locality -Block level parallelism -Block level fault isolation -Example: DNS TCP reassembly
Windowed-Streaming	1 full window of time ordered blocks	Strict ordering or out-of-order across windows as per request	-Process input windows -Write output blocks -Run to completion in bounded time	Yes, by specifying window size	No except emit in output block(s)	-Data accumulation -Reductions like MapReduce -Multi-input ordering -Example: DNS daily stats
Stateful-Streaming	Never ending data stream	Oldest data first or strict monotonic stream order as per request	-Read input blocks -Write output block(s) -Application continuously run	No, but app can flow control incoming data	Yes, long-term state	-Lower latency apps -Always on processing -Example: Real-time intrusion detection

TABLE I: Comparison of three abstractions.

observer captures data as a stream of 2 GB-blocks of packet captures. Under Block-Streaming, we process each block independently, hence providing good block-level parallelism and fault isolation. As developer call-back functions process these blocks, they generate new large-blocks, whose size depends on the amount of data emitted by an application. Developers don't have direct control on the size of the block.

Block-Streaming is suitable for applications that need single pass processing, and they leverage spatial or temporal locality. Additionally, Block-Streaming blocks facilitate low-overhead deduplication of computation and storage [28]. The size of the block is domain-specific and configurable as per the need.

Plumb streams one large block to an instance of developer call-back function. Developer code is expected to process data and stream out one or more data blocks. Block-Streaming requires developer call-back function to run to completion in a bounded time. Plumb uses a pre-defined upper run-time limit for Block-Streaming based developer code.

Developers specify their job requirement in a pipeline-graph (see Figure 3) and our framework takes care of streaming in, streaming out, and retries on failures. The first stage of the workflow in Figure 3 gets captured network data as sz compression, and changes it to more compact xz format for long-term archival by utilizing spatial locality. The second stage also consumes captured DNS data and does TCP reassembly to generate DNS records for output. This stage utilizes temporal locality to correctly stitch DNS TCP flows, because multiple packets of a TCP flow are near-by in the data.

D. Windowed-Streaming Abstraction

A large class of analytics require a particular duration of data—a time window. (As a related problem, some applications want to operate on a very large chunk of data.) With Block-Streaming, processing happens in fixed-size blocks, providing a more uniform unit of work, and allowing different blocks to be processed in parallel. By contrast, time-based windows are often aligned with reporting requirements, and process varying amounts of work.

Windowed-Streaming must bridge the gap between data arriving in discrete blocks, possibly out of order, while providing the developer this simplifying abstraction of a complete window of data. In addition, developers can specify windowing variations: are windows measured by time or data, how much, what phase, can different windows be processed in parallel, how boundary conditions are handled (data that crosses the window boundary), how long should we wait or how much missing data should we allow before timing out and processing an incomplete window, and how we recover from errors. Table II lists these parameters and their defaults. We review these design options below.

To minimize latency while providing simple and clean error handling, Plumb's primary job is to trigger window processing when the window is complete, and to handle error detection and recovery. Since Plumb tracks arrival of new data to the window, can checks for completeness as soon as new data arrives and can schedule window processing as soon as a window completes. By contrast, pre-Plumb systems without framework support for windowing periodically probe to check for window completion, with infrequent polling unnecessarily adding latency.

Error handling is more difficult. Developers specify a timeout of hours after the window should be complete, and require that the number of missing blocks be below a threshold or percentage. Plumb requires that applications provide a sequence number and data start time for each block in a data stream. These values mean it can identify a window may be complete from block start times (the window will be complete when timestamps bracket the window start and end times), and confirm it is complete by insuring all sequence numbers are present. This information also allows Plumb to handle error conditions: has too long passed after the window should be complete, and if so, is a large-enough percentage of blocks present to process anyway?

Systematic error handling and tracking window completion is a major advance in Plumb compared to our pre-Plumb systems—its ad hoc code judged completeness by counting "enough" blocks per day, based on estimates of typical counts. Such estimates are fragile in the face of operational changes such as sites being closed for maintenance or large increases or decreases in traffic from external changes. Centralizing error handling simplified our implementation, reducing code size by 58% (97 vs 231 lines-of-code) (§III-B1). Most operator interventions in our pre-Plumb system were due to violations of our ad-hoc windowing implementation due to unexpected operational changes; use of Plumb would have lowering operator interventions by 73% (§III-B2).

Error recovery is necessarily specific to applications. We allow developers to specify a recovery function to list the blocks that are present and missing. That function can choose to ignore missing data, or can estimate what that data would

Window Parameter	Semantics	Mandatory?	Allowed Values	Default Value
size	What is the size of the window in terms of time or number of blocks	Yes	X hours or Y blocks	N/A
start	Where the window should start. This property must be used along with size	No	YYYY:MM:DD:hh:mm UTC	Head of the queue
start_and_end	When to start and finish windowing. Either size or this property required	Yes	<start_time-end_time>	N/A
max_wait	Max wait time or missing blocks or percentage of missing blocks before moving on	No	x hours or y blocks or z%	1 hour
across_window_ordering	If windows complete out-of-order, are they scheduled in order or not	No	Yes or No	No
cutoff_delivery_requirement	For time-based windows, how to deliver cut block	No	Cutoff_Exactly_Once_To_Current_Window Cutoff_Exactly_Once_To_Next_Window Cutoff_Duplicate_Delivery	Cutoff_Exactly_Once_To_Current_Window
use_queue_sequence_number_instance	Make sub-windows based on developer provided label in block names such as sites	No	Yes or No	No
queue_sequence_number_instance_list	Developer can provide specific labels for sub-windowing that occur in a pre-defined place in a block name, or if not given framework learns them over time from block names	No	Comma separates list of text labels	N/A
framework	For future use: allows organizations to performance tune wrappers for specific frameworks	No	Not implemented yet	N/A

TABLE II: Properties of a window.

have said. We are still experimenting with error handling, but being able to make an informed response to missing data is a step forward. Our pre-Plumb do not do error recovery because it lacked information to estimate what was missing.

Plumb supports parallelism between windows (the default), or allows the developer to process windows sequentially. Sequential processing is important if a window must consider state saved from the prior window, but it requires that window processing be faster-than-real-time.

With time-based windows, phase and boundary conditions must be considered. The phase of the window is its start time relative to an absolute clock: should the window run every 24 hours based on when it began, or should it run midnight-UTC-to-midnight? When data arrives in blocks, the first and last blocks of the window will usually only partially overlap the window's time frame. By default, the first block in a window starts in its time frame, so each block is processed once but the window may start slightly late. The developer can also request that blocks that straddle window boundaries be delivered to both windows, allowing the application to get a 100% complete window after discard data outside the time frame. Our pre-Plumb system did not handle boundary blocks.

We have several examples of window-based processing in our workflows. In our DNS case study (§II-A), RSSAC statistics concern a midnight-UTC-to-midnight window. In another deployment (pre-dating Plumb) we processed 24- or 12-hour windows of packets to convert to net flow. Conversion to Plumb has reduced the code developers must maintain, reduced latency by 74% (§III-A), and lowering operator intervention to handle problems by 73% (§III-B2).

In summary, Windowed-Streaming enables developers to write correct, low-latency application with ease.

E. The Stateful-Streaming Abstraction

The Stateful-Streaming abstraction is suitable for those applications that need long-term state and and along-running process for near-real-time consumption of streaming data. Intrusion detection is an example application that benefits from this abstraction, as are other systems that often capture data directly from the network. Plumb's Stateful-Streaming enable these applications to share a single network tap. Design choices include handling data ordering and missing data, parallelization, and fault-tolerance and crash recovery of the long-running process. We discuss them one by one.

Plumb streams data in-order to the developer's code, re-ordering blocks that arrive out-of-order. However, missing or very late blocks require other choices—should Plumb ignore any absent data and move on, or should Plumb wait, and how long? The developer can specify how long to wait for missing data, and what action to take to handle a time out. Missing out on data means different things for different applications—a financial application might not be willing to lose any data while an application counting a long-term average might be willing to let go some data.

An application's ability to process data in time depends on stream arrival rate, and speed of the underlying hardware and software. While a single-instance application is easier to write and manage, at some point aggregate data may exceed the capacity of a single processing instance. Some streaming data tools (for example, Zeek) include support for parallel operation. Alternatively, Plumb can hash traffic into separate partitions to support parallelism across multiple instances of the data stream. In this case, hashing would be a Plumb workflow step that sends data to several separate queues (each with Stateful-Streaming), and Plumb will handle parallelism and fault tolerance. We currently support single instance operation and plan support for these forms of parallelism.

Finally, we need to plan for failure of even long-running processes. In some cases, a Stateful-Streaming worker can restart, but in other cases it may need to retain and restore state. If the application is prepared to periodically checkpoint its state, but requires the replay of recent data to recover, Plumb can assist by retaining and replaying recent data (similar to Kafka [23]).

As an example of Stateful-Streaming in our case study, we pass all traffic through Zeek [26]. Normally Zeek directly reads from the network, but we use Plumb to multiplex data collected for our existing workflow across Zeek. (Zeek runs indefinitely as if it was capturing directly from the network.) Zeek output can be fed back into Plumb as an output queue, or logged independently.

In summary Stateful-Streaming allows applications to process continuously coming data efficiently. We evaluate our design choices in §III-C by using a network intrusion detection system Zeek on our DNS workflow (last stage in Figure 3).

F. Fault Tolerance

Failures happen due to a myriad of reasons, so automatic masking and graceful tolerance of these failures are important to make a system robust and easy-to-use. Inside our system, we can categorize failures into two classes—compute failures and data failures.

1) Compute Failures: Compute failures happen when a stage fails unexpectedly, probably due to some hardware or software problem. We use retries like MapReduce [12]; rerunning after delay recovers from intermittent problems such as network or storage failures. Since stage execution is idempotent, possible duplicate retries do not cause problems other than wasting a small amount of capacity. We cannot do automatic recovery for state-bearing stages, but hooks allow a developer to save and restore checkpoints with custom recovery. Our system allows stages to store their state inside our meta-data store or use an external state management system while keeping a pointer to the external state inside Plumb. If a given input fails after multiple retries, we mark the data block as faulty and request a manual inspection. By default we support four retries (same as in Apache Hadoop [33]); we plan to make this value customizable. Our system schedules and executes each stage separately, so failure of one stage does not impact other stages.

2) Data Failures: Data failures happen when some blocks come in late or never arrive. On an asynchronous network like the Internet, such data faults are common, and we need to manage them for our three abstractions (Block-Streaming, Windowed-Streaming and Stateful-Streaming abstractions).

Our Block-Streaming abstraction provides out-of-order block execution and so is not affected by missing blocks. Stages using the Block-Streaming might ignore missing data to achieve high parallelism and low latency now and to deal with data faults in a later stage.

For the Windowed-Streaming abstraction, developers need to process complete data in a window and they also need low latency. Data completeness and latency have a tradeoff between them—waiting longer might help getting a complete window but at the expense of added latency. We enable developers to choose a trade-off between data completeness and latency for their windows and allow developers to set parameters to run windows with partial data. Developers can specify missing data either as an absolute value or a percentage of data in a window. Developers can bound their wait time using a timeout value. To facilitate data completeness, each block has a unique identity in our system so that we can keep a tally which blocks of a window are present and which are missing. As soon as all the blocks of a window are present, our system schedules the window for processing.

A developer can specify maximum wait time, maximum number of missing blocks, or percentage of missing blocks allowed to trigger a window for processing. To dial the tradeoff between data completeness and latency, we provide *max_wait* argument (Table II) in our Windowed-Streaming abstraction so that our system could automatically manage missing or late blocks in a window. Developers can control latency or completeness using above mentioned parameter. If a window is unable to trigger for processing (for example, because a developer requested availability of all data and something is missing), we inform related developer of such cases and provide tools to interact with the window for a manual directive (for example, process with whatever is available).

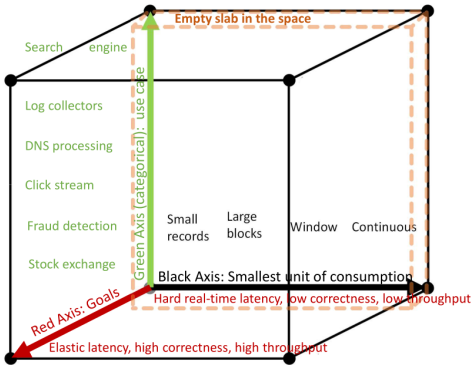

Fig. 4: Design dimensions in analytics (unit of consumption, processing goals, business use-cases).

For the Stateful-Streaming abstraction, developers need an in-order stream. We divide the stream into blocks and assign each block a unique, monotonically increasing sequence number so that our system knows the order of the stream, and any missing blocks. Our system provides an API so that stage programs could detect gaps in the in-order stream, and developers can directly manage their wait time. Stages need to invoke our API after each block consumption to know if the next block is available. If it is not, the stage can wait and timeout, or move on and skip the block. These options allow latency-sensitive processing with Windowed-Streaming, and fault recovery is necessarily application-specific.

We evaluate data completeness and latency tradeoff in §III-B2, and show that our system provides consistently low latency under faults in §III-B3.

G. Application Coverage

Here we argue that our three abstractions apply to large classes of streaming applications. We classify stream processing based on multiple dimensions—business domain, smallest unit of input consumption, and processing goals (see Figure 4). Developers might use our abstractions to efficiently process their data from different business domains, for variable-sized input consumption, and for broad efficiency goals. The green axis in Figure 4 present example applications for all of the above dimensions to show applicability of our abstractions. Our system is applicable to many classes of applications except when the need is hard-real-time.

III. Evaluation

We use applications from our DNS workflow to evaluate our system. This workflow (shown in Figure 1) shows all three Plumb abstractions: it uses large-block streaming for statistics generation and archiving (nine steps on the left of the figure). It uses Windowed-Streaming to accumulate a 24 hours window of data that is then processed with map/reduce. Finally, it uses Stateful-Streaming to send a virtual stream of all data to Zeek, a standard logging and intrusion detection tool.

A. Block-Streaming Enables Low Latency Processing

To evaluate Block-Streaming's latency, we use steps A to C of our B-Root DNS workflow (Figure 1). We de-compress

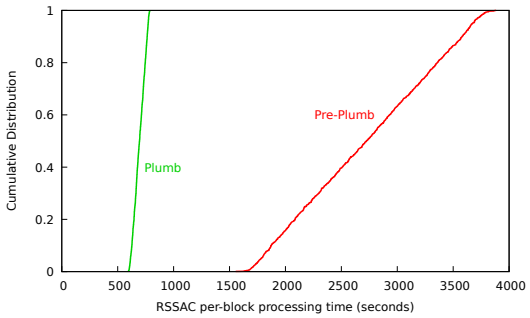

Fig. 5: Latency: Plumb vs. Pre-Plumb processing, as each blocks is processed by three steps to get statistics. Delay includes queuing and processing time. This graph is from our earlier work [28]when only Block-Streaming was available in Plumb. Re-reporting for completeness.

and decode DNS (steps A and B), then compute intermediate statistics (step B to C), both using Block-Streaming.

We evaluate latency from step A to step C (via step B), comparing our pre-Plumb, batch-based system with Plumb using Block-Streaming. We evaluate one day's worth of real data (2018-10-23, 1393 files, each 2 GB when uncompressed). For this experiment, we used a cluster with 37 servers, 544 cores, 1853 GB memory, 224 TB HDFS storage, and 1 Gb/s network bandwidth between the servers. Figure 5 shows the distribution of latency for this workflow over all 1393 files.

Current Plumb latency (the left, green line in Figure 5) is much lower, with median latency of 695 s instead of 2724 s in our prior system. 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 pre-Plumb).

Plumb latency is lower because it processes blocks as soon as they arrive, while our pre-Plumb system grouped blocks into batches to use Hadoop-based parallelism. Plumb latency is good, approaching the theoretical minimum of 388 s from step A to C without any queuing delays, when run on an idle server. (But there is some room for optimization.) Plumb's delays are either due to queuing (all compute resources currently busy) or thundering herd phenomenon when there are many readers and writers on its HDFS shared file system, with 3-way replication (see [28] for more details).

B. Windowed-Streaming Enables Efficient Reduction

We next show ease-of-use, correctness, and low latency of analytics via Windowed-Streaming and compare the results with a pre-Plumb system.

1) Windowed-Streaming is Easy-to-use: One of our design goals is that our system be easy-to-use, so we next compare a pre-Plumb solution that bridges Block-Streaming and windowing with Plumb. For this experiment we compare lines of code to do 24-hour RSSAC statistics [18] in our pre-Plumb, hand-coded windowing mechanism to that with Plumb. We use pygount [2] tool to correctly count the lines-of-code for bash and python scripts.

Developer Code	lines of code pre-Plumb	lines of code Plumb	Reduction
Window Creation	163	28	83%
Statistics	66	59	11%
Scheduling	2	0	100%
Job Specification	0	10	(-100%)

TABLE III: Comparison of number of lines-of-code in pre-Plumb RSSAC (with developer custom windowing) and Plumb RSSAC (with Windowed-Streaming abstraction).

Our experiment in Table III shows that Plumb requires only 41% of the code of our pre-Plumb implementation (a code reduction of 58%). Processing requires four components: window creation, statistics, scheduling, and YAML job description. Window creation is 83% shorter than pre-Plumb because Plumb only calls developer code when a window is complete and ready-to-run. Plumb code is simpler since the framework handles windowing and the many corner cases of incomplete input and error handling (see §III-B2). Window creation code now contains API calls to interface with Plumb and statistics processing.

Statistics code is smaller by 11% because Plumb automates forwarding output results to downstream in the workflow and archiving the output on disk in the final stage. Two lines of scheduling code (a cron job) are eliminated because Plumb automates scheduling across the cluster. Avoiding cron-based polling also reduces latency, as we show in §III-B3. The main new code is the Plumb specification and windowing requirements, 10 lines of YAML (Figure 3).

In summary, Windowed-Streaming is easy-to-use and enables developers to offload data completion and scheduling responsibilities to Plumb while concentrating on their analytics.

2) Windowed-Streaming Improves the Correctness-latency Tradeoff: Analytics should consider all data that is collected. Yet in real-world data collection systems data can be late or out-of-order, or even missing, as it is captured and relayed to the data warehouse and components or networks fail and disks fill. The developer must chose between waiting for perfect data (which may never come if there was a failure), or processing data after a reasonable time to allow for retries and perhaps operator intervention. This tradeoff between latency, correctness, and degree of operator intervention is common to most distributed systems and something for which Plumb should allow control.

We next evaluate Plumb's correctness and the degree of operator intervention when compared our pre-Plumb system with a custom-designed but ad hoc algorithm to estimate completeness, and in the next section (§III-B3) we will see how it affects latency.

Table IV summarizes the tradeoff between data completeness and latency for pre-Plumb and Plumb algorithms. We use real-world B-Root data from 2019-08-19 to 2020-10-10 (14 months). Over this time B-Root expanded, from 3 sites (LAX, MIA and ARI starting 2020-08-19, adding AMS on 2020-01-20 and IAD and SIN on 2020-01-22). This data reflects typical operational issues that arose over that time, including occasional site network failures and shutdown for planned

Site	Pre-Plumb RSSAC Missed (early trig.)	Pre-Plumb RSSAC Stalls	Plumb RSSAC Missed (∞ wt.)	Plumb RSSAC Missed (1 h wt.)	Plumb RSSAC Stalls	Total Days (100%)
LAX	0	7 (1.7%)	0	13 (3.1%)	2 (0.5%)	416
MIA	0	8 (1.9%)	0	19 (4.6%)	4 (1.0%)	416
ARI	1 (0.2%)	25 (6.0%)	0	45 (10.8%)	1 (0.2%)	416
AMS	1 (0.4%)	16 (6.1%)	0	76 (28.8%)	4 (1.5%)	264
IAD	0	1 (0.4%)	0	22 (8.5%)	3 (1.2%)	260
SIN	0	1 (0.4%)	0	33 (12.7%)	2 (0.8%)	260
all	2 (0.1%)	58 (2.8%)	0	208 (10.2%)	16 (0.8%)	2032

TABLE IV: Comparing pre-Plumb and Plumb data stalls and number of missed data events over 14 months.

Site	Latency (in hours) pre-Pl. 50%ile	Plumb 50%ile	pre-Pl. 90%ile	Plumb 90%ile	pre-Pl. 99%ile	Plumb 99%ile
LAX	9.72	0.29	12.72	0.41	53.91	7.32
MIA	9.75	0.37	13.07	0.52	89.64	26.64
ARI	15.04	0.41	17.58	1.26	666.04	169.36
AMS	10.76	0.43	34.47	18.68	90.14	91.00
IAD	10.44	0.79	13.95	0.81	39.17	12.22
SIN	10.5	0.70	15.1	1.98	39.18	30.32

TABLE V: Comparison of pre-Plumb and Plumb latency (in hours) at 50, 90, and 99%ile for RSSAC processing.

maintenance. Each site computes separate rssacint intermediate statistics, these are then later combined on another system.

We use same data with two systems: pre-Plum and Plumb. The input gives data generation time and the time it is delivered to our analytics cluster. For Plumb, we extract information of interest from our HDFS logs. For pre-Plumb, we replay data to find latency. This replay preserved the data arrival order and time, but did not emulate inter-arrival waits to expedite loading of 14 months of data (420409 input files, 73 MB each), and because our latency calculation use timestamps associated with input data. Some input data might get lost due to hardware/software issues at the capturing site. We know about missing input files using a unique, monotonically increasing sequence numbers in the input names. Our pre-Plumb system uses our custom-built but ad-hoc algorithm to estimate completeness. These algorithms use conservative estimates from historical data to estimate how many blocks each site should generate each day to detect data relay stalls. If these checks fail, they alert the operator and block progress until they can investigate and confirm or correct any problem. Our Plumb-based system allows the operator to select the tradeoff in completeness and delay in the Plumb specification, and allows Plumb to infer completeness based on file times and sequence numbers. Plumb checks are therefore much more accurate and robust to changes in site load over time.

We first evaluate correctness in Table IV, then later consider latency in Table V. Pre-Plumb rarely misses available data (column 2 in Table IV) because it adopts a very conservative algorithm. We compare to Plumb with two settings: with infinite wait, it never misses data (but requires operator intervention on a stall), and with a 1 h threshold we see it misses data about 10% of the time. Here both have similar behavior when correct, but due to different reasons—pre-Plumb's conservative wait time coupled with long-term traffic rate mostly works while Plumb relies on monotonically increasing sequence numbers of the underlying blocks for completion checks. With an aggressive threshold, Plumb can be configured to meet a strict (real-time) deadline at the cost of incomplete results. In some cases (if timely results are more important than perfect) this choice may be preferred.

Pre-Plumb stalls falsely many times (58 events, 3% of the time, column 3 in Table IV). These stalls occur because traffic changes, usually due to intentional engineering, making prior traffic estimates invalid. Each of these events require an operator intervention to reset the thresholds. By contrast, Plumb stalls 16 times, always because some data file was blocked due to an error or network problem.

Finally, we see that pre-Plumb's ad hoc algorithm silently misses some actual stalls. Plumb reports 3 and 2 for sites IAD and SIN, while pre-Plumb stalled only once for each, missing 3 events. Pre-Plumb's traffic estimation algorithm cannot detect single file losses, while Plumb's check of sequence numbers is complete. Although missing one file has only a small effect on statistics, detecting errors is important.

In summary, Plumb only produces true stalls for developer intervention, while allowing developers to pick a suitable wait timeout value—hence allowing a developer-guided tradeoff between completeness and latency.

3) Windowed-Streaming Enables Low Latency Processing: Our second design goal is low latency, so we next evaluate latency using the same pre-Plumb and Plumb with Windowed-Streaming implementations as we did in §III-B2, again comparing RSSAC statistics for the 6 B-Root sites.

As before, we collect times of RSSAC input files (step C in Figure 1) and the output of rssacint summary files from HDFS logs. We examined all six sites, but here we report data for two sites, LAX and AMS. (Other sites are similar to LAX.)

Figure 6 shows the CDF of latency for RSSAC output at LAX with Plumb (the left, green line) and pre-Plumb (the right, red line) over 14 months. Figure 7 shows similar results for AMS for nearly 9 months. Table V summarizes median and tail-latency of all six sites.

Plumb has much lower latency at all sites. At LAX, median and 90%ile latency are only 3% of their prior values (to 0.29 hours from 9.72 at 50%ile, and 0.41 from 12.72 for 90%ile). The tail for both systems is long, but 99%ile latency is 33% of its prior value (left curve compared to right curve in Figure 6 and column six vs. seven in Table V). The LAX site behaves with Plumb as we hope, usually processing data within the hour it's available, and only occasionally requiring longer.

AMS and some other sites show much longer tail latency, with AMS 99%ile latency of nearly four days. These delays represent a few days spent data back-haul for this site when it was coming on-line.

Finally, we see very long latency for the ARI site: 99%ile latency is 169 hours (more than a week). There were events when ARI had traffic engineering that dropped its data rate to zero. Correcting this problem brought ARI latency back to our goal of less than one hour latency per day.

Finally, median latencies of older sites (LAX, MIA) are lower than newer sites (for example: 0.29% for LAX while 0.43% for AMS) because with the addition of newer sites,

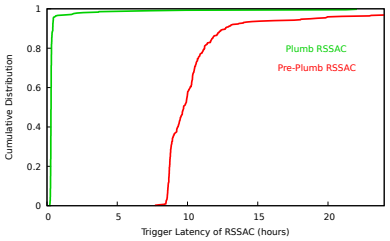

Fig. 6: Latency comparison at LAX site.

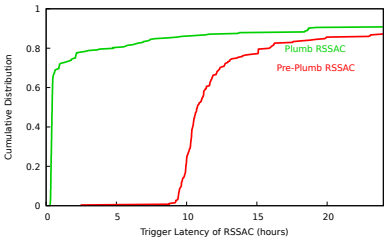

Fig. 7: Latency comparison at newer AMS site.

we are getting more data. Our publicly available RSSAC stats [18]show that traffic increased by about 50% with the addition of the three newest sites and corresponding lower latencies, while our analytics cluster's processing capacity did not change during this time.

The tail latency in graphs reflects our conservative choice of requiring all data (or operator intervention) before processing statistics. If we wished to guarantee processing each day, even with incomplete data, Plumb makes it easy to set a 1 hour timeout and processes whatever data is available. Plumb's infinite and 1 hour columns of Table IV show the implication: about 10% of cases would run with incomplete data, a choice that might be preferred to stalling if approximate results were required by operational needs.

In summary, we see that RSSAC processing based on our Windowed-Streaming improves correctness and makes it easy to provide fixed-latency results when required. More importantly, it relieves the developers and operators of ad hoc, error-prone windowing algorithms and makes it easy (a configuration parameter) to balance correctness and latency.

C. Stateful-Streaming Enables Continuous Applications

Block-Streaming works well for processing that requires little state, and windowing supports data with predicable amounts of state, but some applications expect *continuous state*. Intrusion detection systems and long-term logging like Zeek fall in to this third category, so we next consider how Plumb can drive Zeek with Stateful-Streaming.

As we described in §II-E, any Stateful-Streaming will be limited by the ability of one compute to keep up. What that exact rate is depends on the workload and the hardware doing the processing. As one example, we evaluate data from one B-Root site over 24 hours starting on 2020-11-27, considering about 1TB of data (538 blocks, each 2 GB before compression). Figure 8 shows the distribution of block arrival and Zeek processing to see how well it keeps up with real-time on our hardware, using just one core.

For our hardware and this application, data arrives every three minutes, and is processed in 9.7 minutes (both median). Only about 3% of blocks are handled faster than incoming traffic—our hardware cannot keep up with this workload. To process this workload will require either Plumb-level flow-based hashing, use of cluster-mode Zeek where it does flow hashing, or addition of windowing to allow different days to process concurrently (without retaining state between days).

Either Plumb- or Zeek-based flow hashing would support this workload with 4-way parallelism (where each way uses one core, and four instances can be on different machines on the cluster) since processing time is consistently between 8 and 10 minutes. Processing time is so consistent because input data is fixed size (2 GB per block). Our initial prototype that uses two developer-level stages—first for hashing and splitting [30] and the second for Zeek processing confirms (see Figure 9) that four instances can keep up with the LAX real-time traffic.

In summary, Stateful-Streaming was able to support our state bearing application, though initially we found out that we need 4-way parallelism to keep up with the traffic rate with our hardware. This need for processing parallelism suggests we should provide Plumb-level flow splitting as a primitive, and allow Plumb jobs to schedule multiple cores to support cluster-mode Zeek. We are able to achieve parallelism by using an additional stateful state before Zeek queues. We are extending Plumb for better support for hashing/splitting input.

IV. Related Work

The primary goal of our work is to enable our developers to process data using multiple abstractions easily and with good performance. We compare our effort with related systems to evaluate the differences and similarities.

Facebook's data processing system with Puma, Stylus, and Swift [10] provides multiple abstractions like us, but it uses three independent frameworks—making a workflow that might use all of them is left to the developers. Plumb allows developers to mix abstractions in a single workflow. Facebook's Puma system allows developers to do operations on an arbitrary size of data, but it uses SQL queries to combine many small records and relies on query optimizations to get locality and good performance. Our Block-Streaming abstraction allows developers to write code to benefit from single pass processing for temporal and spatial locality, at the cost of a lower-level abstraction. Facebook's Stylus system provides data ordering but they do not specify how they handle late, missing, or duplicated data. In Windowed-Streaming we allow the developer to choose completeness criteria and handle errors. Facebook's Swift system provides continuous streaming like our Stateful-Streaming, but underlying data management [14] is different—their data moves out to long-term archives automatically over time, while Plumb keeps track of each data item and proactively either delete or archive

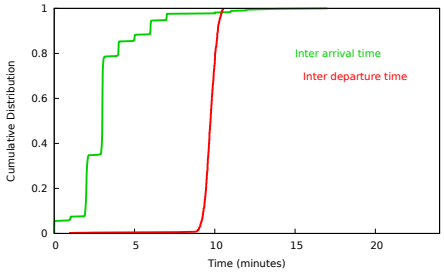

Fig. 8: Comparison of LAX inter-arrival time (for blocks entering E in Figure 1 VS inter-departure time (for blocks exiting E in Figure 1) on one day data.

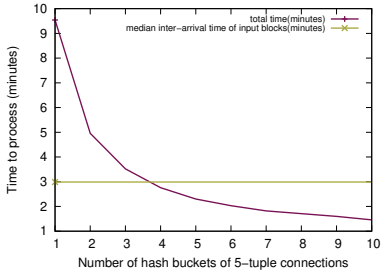

Fig. 9: 4-way connection hashing (on data from A to E in Figure 1) and using 4 Zeek instances on the cluster (as provided by Plumb) is enough to keep up LAX real-time traffic rate.

data right after last-use; hence keeping free storage to absorb any DoS attack traffic. Facebook's approaches reflect the need for multiple abstractions, but they do not describe integration to combine those across a workflow as we do.

Google's dataflow system [4] maps a developer's SQL code to one of the underlying frameworks (MapReduce [12], FlumeJava [9], MillWheel [3])—only allowing to use abstractions in any one framework at a time. Plumb allows its developers to use a different abstraction on each step of a workflow. Their frameworks don't support Block-Streaming and implementing complex functions like TCP reassembly is not straightforward in their SQL-like language.

To provide developers multiple frameworks on a single shared cluster, Nexus [20] (a precursor of Mesos system [19]) provides an isolation boundary between different frameworks (MapReduce [12], Dryad [21]) and provide each framework a slot abstraction. We go beyond their work to provide three different abstractions, and to support the developer in moving data between abstractions in our framework.

Spark streaming [34] provides a distributed shared memory abstraction for continuous stream and allows its developers to perform series of operations on a group of data. They provide abstractions similar to us that are applicable to in-memory data only and iterative workloads that repeatedly apply operations on slowly changing data for sub-second latency. Our Block-Streaming allows to leverage spacial and temporal locality for single-pass operations. Plumb allows developers to use Spark as a framework with our Stateful-Streaming.

Streaming systems like Flink [8] optimize for throughput or latency by configuration; we focus on throughput while considering latency as a secondary goal. Flink is optimized for streaming small records, depends on data sharding, and does not provide multi-user collaboration or deduplication abilities. Plumb is optimized for large blocks, manages skew when data sharding is not possible and provides multi-user collaboration and deduplication. Flink's APIs are integrated with a specific programming language, limiting application to functionality available inside that framework. Plumb's pipeline graphs provide language-independent job descriptions with multiple abstractions, supporting more diverse components and abstractions from multiple frameworks.

Airflow [6] and Oozie [22] are workflow managers to glue Hadoop-based frameworks. They do not provide higher-level abstractions, like ours, and they leave moving data between abstractions and windowing to the developer.

SECRET and TSpoon [1], parts of the system described by Lorenzo Affetti et al, use data windowing to provide interoperability between relational databases and streaming data, and allowing database operators on streaming data. They don't provide Block-Streaming and Stateful-Streaming. They provide ACID transactional guarantees while Plumb operations provide at-least once semantics.

There are many systems (examples include [15], [31], [5], [24], [29]) that provide glue between two specific frameworks. Plumb provides three abstractions that cover a broad part of the design space, and glue to move data between them.

Scientific workflow systems such as Pegasus [13] work well for big-science applications [25], [7] that run periodically. Plumb is instead optimized for mixes of continuously streaming data with windowed computation. Potential future work is to compare applications across these frameworks.

V. Conclusions

Plumb makes it easy for developers to access multiple abstractions in their big-data workflows, simplifying code, lowering latency, and improving correctness. To do so, it bridges data streams between three different abstractions (Block-Streaming, Windowed-Streaming, and Stateful-Streaming), while managing error handling and allowing developers to dial the trade-offs between latency and completeness. These abstractions optimize for different goals (Block-Streaming for high parallelism when some context is needed, Windowed-Streaming for time-dependent processing, and Stateful-Streaming for long-running applications with state). Versions of Plumb have been in production use at B-Root for more than two years and has already reduced latency by 74% (blocks) or 97% (daily statistics), while reducing code size and improving correctness. Plumb is open source and available for use.

Acknowledgements

This research was partially supported by contract W911NF-17-C-0011 with the U.S. Army Contracting Command–Aberdeen Proving Ground (ACC-APG) and the Defense Advanced Research Projects Agency (DARPA). It is also partially supported by the National Science Foundation (project CNS-1925737, DIINER). 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 Government purposes notwithstanding any copyright notation hereon.

References

[1] L. Affetti. *New Horizons for Stream Processing*. PhD thesis, Politecnico Di Milano Dipartimento Di Elettronica, Informazione E Bioingegneria, 2019.

[2] T. Aglassinger. Pygount project at PyPI. https://pypi.org/project/pygount/. (Accessed on 12/08/2020).

[3] 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 *Proc. Very Large Data Bases*, pages 734–746, 2013.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[5] G. Barbier and L. Cibot. Zenaton — workflow builder for developers. https://zenaton.com/. (Accessed on 07/16/2021).

[6] M. Beauchemin. Apache airflow. https://airflow.apache.org/. (Accessed on 07/16/2021).

[7] B. Berriman and J. Good. Montage – pegasus wms. https://pegasus.isi.edu/application-showcase/montage/. (Accessed on 07/08/2021).

[8] 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.

[9] 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.

[10] 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.

[11] Databricks. CIO survey: Top 3 challenges adopting AI and how to overcome them. https://pages.databricks.com/rs/094-YMS-629/images/DatabricksIDG_eBK0911[1].pdf. (Accessed on 02/10/2020).

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

[13] 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.

[15] D. L. Freire, R. Z. Frantz, and F. Roos-Frantz. Ranking enterprise application integration platforms from a performance perspective: An experience report. *Software: Practice and Experience*, 49(5):921–941, 2019.

[14] F. Engineering and I. Team. Scribe: Transporting petabytes per hour - facebook engineering. https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/. (Accessed on 07/16/2021).

[16] D. L. Freire, R. Z. Frantz, F. Roos-Frantz, and S. Sawicki. Survey on the run-time systems of enterprise application integration platforms focusing on performance. *Software: Practice and Experience*, 49(3):341–360, 2019.

[17] K. Fukuda, J. Heidemann, and A. Qadeer. Detecting malicious activity with DNS backscatter over time. *ACM/IEEE Transactions on Networking*, 25(5):3203–3218, Aug. 2017.

[18] J. Heidemann and A. Qadeer. B-root dns statistics. https://b.root-servers.org/rssac/. (Accessed on 07/16/2021).

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, S. Shenker, and I. Stoica. Nexus: A common substrate for cluster computing. In *Workshop on Hot Topics in Cloud Computing*, 2009.

[21] 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.

[22] 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.

[23] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[24] LinkedIn. Azkaban: Linkedin workflow manager. https://azkaban.github.io/. (Accessed on 07/16/2021).

[25] J. Livny. Sipht – pegasus wms. https://pegasus.isi.edu/portfolio/sipht/. (Accessed on 07/08/2021).

[26] V. Paxon. The Zeek network security monitor. https://zeek.org/. (Accessed on 12/06/2020).

[27] A. Qadeer and J. Heidemann. Ant plumb. https://ant.isi.edu/software/plumb/index.html. (Accessed on 07/08/2021).

[28] A. Qadeer and J. Heidemann. Plumb: Efficient stream processing of multi-user pipelines. *Software: Practice and Experience*, 51(2):385–408, 2020.

[29] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag. Streams and tables: Two sides of the same coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, BIRTE '18, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Seladb. Pcapplusplus/examples/pcapsplitter at master · seladb/pcapplusplus · github. https://github.com/seladb/PcapPlusPlus/tree/master/Examples/PcapSplitter. (Accessed on 12/14/2020).

[31] Spotify. Luigi: Python module for building complex pipelines of batch jobs. https://github.com/spotify/luigi. (Accessed on 02/12/2021).

[32] 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.

[33] 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.

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

[35] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.