Pravega: A Tiered Storage System for Data Streams

Raúl Gracia-Tinedo
Flavio Junqueira
raul.gracia@dell.com
flavio.junqueira@dell.com
Dell Technologies
Barcelona, Spain

Tom Kaitchuck
Sachin Joshi
tom.kaitchuck@dell.com
sachin.joshi@dell.com
Dell Technologies
Seattle, USA

ABSTRACT

The growing popularity of the data stream abstraction entails new challenging requirements when it comes to data ingestion and storage. Many organizations expect to retain data streams for extended periods of time and to store such stream data in a cost-effective manner. It is also crucial to reconcile apparently opposite properties, like data durability and consistency, along with high performance. Furthermore, data streams should not only deal with a high degree of parallelism, but also adapt to fluctuating workloads with little or no admin intervention. To our knowledge, no storage system for data streams fully copes with all these requirements.

In this paper, we present Pravega: a distributed, tiered storage system for data streams. Pravega streams are unbounded by design and cost-effective, as the system automatically moves data to a long-term storage tier (*e.g.*, S3, NFS) and transparently manages it for the user. Pravega guarantees no duplicate or missing events, as well as per routing-key event ordering, while providing high performance streaming IO and historical reads. As a unique feature, Pravega streams are elastic: they can automatically change their degree of parallelism based on the ingestion workload. We compared the performance of Pravega with Apache Kafka and Apache Pulsar on AWS. Our results certify that Pravega can deliver performance improvements over them in many scenarios.

CCS CONCEPTS

• **Information systems → Hierarchical storage management**; **Distributed storage**; **Data streaming**.

KEYWORDS

data streams, distributed storage, storage tiering, performance

ACM Reference Format:
Raúl Gracia-Tinedo, Flavio Junqueira, Tom Kaitchuck, and Sachin Joshi. 2023. Pravega: A Tiered Storage System for Data Streams. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3590140.3629113

1 INTRODUCTION

The volume and variety of streaming data sources is growing at unprecedented rates. Ranging from classical use cases, such as social networks [44, 52] or retail shopping [1], to emerging ones like IoT and edge analytics [50, 54, 57], the amount of *events* or *messages* that both humans and machines generate on a daily basis is challenging to harness. However, as difficult as it may seem, such a data deluge can be exploited to get valuable insights to the benefit of users and organizations.

Not surprisingly, according to a recent report from Gartner [5], about two-thirds of companies use stream processing for low-latency stream analytics. This is possible, to a large extent, due to the efforts from the research community and the industry to implement more sophisticated streaming processing engines [11, 16] and services [6, 18] that unlock the value of data streams.

While stream processing engines have been in the spotlight due to the pressing need for extracting value from data, less attention has been paid to researching the specific challenges of stream data ingestion and storage. As soon as the need for storing stream data for longer periods of time appeared, traditional queuing systems were replaced by a new family of messaging systems [33], such as Apache Kafka [12, 43] and Apache Pulsar [14]. These systems expose the *topic* or *stream* abstraction, which allows applications to durably write events and read them back. This new approach to storing data streams aligns with the semantics of stream processing engines, thus fostering their broad adoption.

Challenges. Although current messaging systems are a good fit for some streaming use cases, we argue that many modern streaming data sources and processing workloads may bring new data ingestion and storage challenges not anticipated by these systems. Concretely, we identify the following ones: *(c1)* achieving cost-effectiveness when storing unbounded stream data, *(c2)* providing data durability and consistency with high performance, *(c3)* handling high levels of parallelism, and *(c4)* adapting to fluctuating workloads with no human intervention.

(c1) First, organizations may expect to store stream data indefinitely and in a cost-effective manner. Today's messaging systems are provisioned in clusters with drives to store *log files* or *partitions* of topics. Naturally, provisioning fast, local drives can substantially improve write latency in these systems. However, if we think of storing data for extended periods of time, having expensive drives for storing mostly cold stream data is not cost-effective. The root cause of the problem lies deeply in the fact that the design of these systems couple performance and storage capacity concerns within the same infrastructure. To partially mitigate this problem, systems

like Pulsar recently provided add-ons for offloading data to external storage [15]. As we show in this paper, our proposal of integrated storage tiering can provide faster historical reads than Pulsar.

(c2) Data durability and consistency —no duplicates or missing events, event order— are customary, as they are key properties to guarantee correct and reproducible streaming computations. However, in many cases, these properties are assumed to be a given and performance is not expected to be impacted. This may not be true in some cases: for instance, Kafka by default does not flush data to persistent media when acknowledging writes to favor performance [13]. Not enabling per-write flushes might compromise data durability upon correlated failures. In this paper, we show that enabling data durability in Kafka has a significant performance toll.

(c3) Streaming applications may ingest data from a wide range of elements —*e.g.*, sensors, users, servers— concurrently to form a single stream of events. Therefore, messaging systems storing data should be able to handle a high degree of parallelism, typically by splitting a topic or stream into parallel partitions. Note that this is not only relevant to ingest stream data but also to data processing. We believe that the solution to this problem requires the system to multiplex operations from multiple partitions into a single log file. However, the design of systems like Pulsar, and specially Kafka, do not fully exploit multiplexing for using drives more efficiently. We show that operation multiplexing has a significant impact when ingesting stream data for large numbers of clients and partitions.

(c4) It is unrealistic to expect that the workload for a data stream will remain constant indefinitely. As pointed out by numerous measurements [37, 45], workloads can greatly fluctuate due to many reasons, including popularity spikes and daily patterns. Therefore, users should be able to modify the degree of stream parallelism to accommodate a sudden increase in the ingestion workload. Most importantly, this needs to incur little or no human intervention to prevent higher administration costs and response time. Unfortunately, to our knowledge, no messaging system can do this today.

Contributions. In this work, we present Pravega: a distributed, tiered storage system for data streams. Pravega streams are potentially unbounded and cost-effective, as the system automatically moves data to a long-term storage tier (*e.g.*, S3, NFS) and transparently manages it from the client's viewpoint. Pravega guarantees no duplicate or missing events, as well as per routing-key event ordering, while providing high performance in streaming IO and historical reads. As a unique feature, Pravega streams are elastic: they can automatically change their degree of parallelism based on the ingestion workload. The key contributions of this paper are:

- The design of Pravega and the mechanics of Pravega streams to provide data durability, consistency, and auto-scaling.
- The implementation of Pravega, putting special emphasis on the performance of the IO path and storage tiering.
- The evaluation of Pravega against Kafka and Pulsar on AWS.

Our results show that Pravega satisfies the aforementioned requirements and still outperforms existing messaging systems in many scenarios. For instance, in our experiments, a single Pravega writer exhibits similar or lower write latency while guaranteeing durability compared to a Kafka producer with no durability. Pravega is reported to sustain the highest levels of workload parallelism (*e.g.*, 5K partitions, 100 writers), while other systems saturate much earlier. We also measured historical reads in Pravega to be up to 8x faster than in Pulsar and we demonstrate promising results of stream auto-scaling. We conclude that Pravega is an attractive alternative to store stream data and serve stream processing pipelines.

The remainder of the paper is organized as follows. In Section §2, we describe the design of Pravega. Section §3 covers the operation of Pravega streams and how they achieve event ordering and consistency. In Section §4, we describe the IO path of Pravega. We show our results in Section §5 and review the related work in Section §6. Finally, we conclude the paper in Section §7.

2 PRAVEGA DESIGN

2.1 Preliminary Concepts

Pravega is a distributed, tiered storage system for data streams. The main goal for Pravega is to allow applications to durably write and read *events* (Pravega also supports byte streams [22], but it is out of the scope of this work). Pravega provides users with *client libraries* that implement the Pravega APIs, including *event writer* and *event reader*. Note that applications make sense of events using (de)serializers as internally Pravega does not keep the notion of events (*i.e.*, Pravega does not internally track event boundaries).

Pravega stores events in *streams* (like a "topic" in messaging systems). A stream is a durable, elastic, append-only, unbounded sequence of bytes achieving good performance and consistency. Although events in a stream cannot be modified, there are other valid operations to change the state of a stream, such as *seal* (*i.e.*, make a stream read-only), *truncate* (*i.e.*, delete data from an arbitrary point in the stream up to its "head"), *scale* (*i.e.*, change in parallelism), and *delete*. Groups of streams are organized into *scopes*, which act as stream namespaces.

Internally, streams are divided into *segments*. A stream segment is a shard or partition of the data within a stream. Similarly to the case of streams, the allowed operations on segments include append, truncate, seal, merge, and delete (but not update). It is worth mentioning that a stream may have multiple segments open for appending events at a given time, which enables higher throughput. When working with streams having parallel segments, users requiring event order are expected to use *routing keys* for writing data. To wit, parallel segments in a stream are assigned to partitions of the key space as a result of a hash function (*e.g.*, $h(k) \in [0, 1)$). The writer API accepts as input a user-provided routing key to consistently select the segment for appending events to and order writes with the same key.

An interesting feature of Pravega streams is that they are policy-driven. Pravega currently provides two types of *stream policies*: *retention policies*, which automatically truncate a stream based on size or time bounds; and *auto-scaling policies*, which allow the system to automatically change the segment parallelism of a stream based on the ingestion workload (events/bytes per second). Note that stream policies can be updated along the stream life-cycle.

2.2 Architecture

The architecture of Pravega is illustrated in Fig. 1. We identify the following system components. First, Pravega clients can interact with Pravega server instances from inside or outside the cluster.

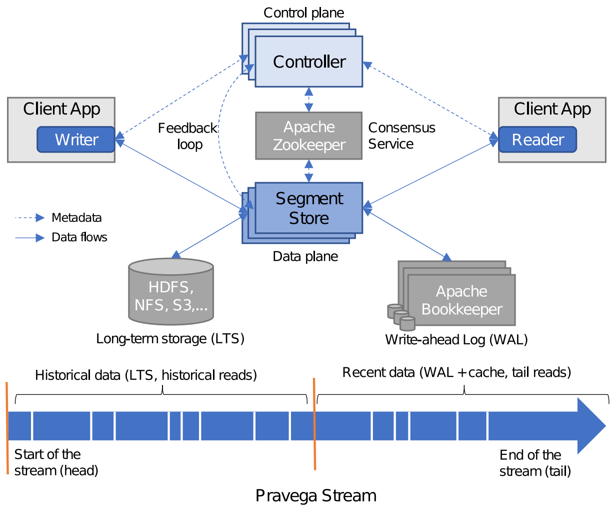


Figure 1: High-level overview of Pravega.

On the server side, we find the Pravega *control plane* formed by one or multiple *controller instances*. The control plane is primarily responsible for orchestrating all stream life-cycle operations, like creating, updating, scaling, and deleting streams. Furthermore, the control plane enforces stream policies, including truncating streams according to the retention period/size defined and orchestrating the scale-up/down operations for a stream. In the latter case, Pravega builds a feedback loop between the control and data planes, so the control plane can react to the load monitored by the data plane. Pravega can manage a large number of streams by partitioning and distributing stream management tasks across the controller instances. Concretely, a stream is associated with one stream management partition from the number of partitions defined. Stream management partitions are then distributed and owned by controller instances in an attempt to balance the stream management load. Moreover, controller instances maintain the stream metadata (which is stored in Pravega itself via the key-value API built on top of streams [24]) and reply to metadata requests about streams.

The *data plane* handles data requests from clients and is formed by *segment store instances*. Segment stores play a critical role in making segment data durable and serving it efficiently. Note that segment stores only work with segments and they are agnostic to the concept of stream, which is an abstraction built at the control plane. The data plane distributes the segment-related load based on *segment containers*. Segment containers are the components that do the heavy lifting on segments and the main role of segment store instances is to host segment containers. A segment is mapped during its entire life to a segment container using a stateless, uniform hash function that is known by the control plane. The *segment ids* resulting from the hash function belong to a *key space* that is partitioned across the available segment containers to balance load.

The segment store has two primary storage dependencies: *Write-Ahead Log (WAL)* and *Long-Term Storage (LTS)*. The main goal of WAL is to guarantee durability and low latency of incoming writes and keep that data temporarily for recovery purposes (see §4.4). Making a write durable means that once the application is acknowledged that the write has succeeded, the system guarantees that the data is written on persistent media and replicated. Pravega uses Apache Bookkeeper [7, 40] as WAL. Bookkeeper provides excellent write latency for small appends while replicating data.

Segment stores asynchronously migrate data to LTS. Once some data is moved to LTS, the corresponding log file from WAL is truncated. Pravega has an LTS tier for a couple of key assumptions that determined its design: first, data streams are *unbounded* by design and the system should be able to store a large number of segments in a *cost-effective manner*. Consequently, we need a storage like a scale-out file or object store for historical stream data. Second, we need to provide *high throughput and parallelism* for reading *historical data* if applications need to catch up with the stream. The combination of low-latency streaming writes (WAL) and reads (in-memory cache), plus high throughput historical reads (LTS) allow Pravega to achieve an excellent throughput-latency trade-off.

Finally, Pravega uses a consensus service (Apache Zookeeper [17, 38]) for leader election and general cluster management purposes. Recall that Pravega stores metadata in key-value tables backed by Pravega streams [24], meaning that Zookeeper is not a bottleneck.

3 PRAVEGA STREAMS

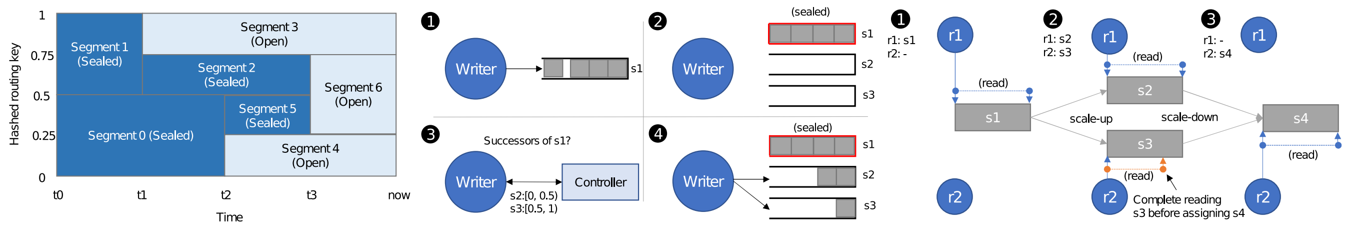
Next, we provide an overview of the mechanics involving clients and server-side instances when operating with streams, such as writing and reading. Also, we describe the guarantees that Pravega provides, which are critical to modern streaming applications.

3.1 Stream Auto-scaling

Stream auto-scaling is a unique feature of Pravega. It allows Pravega to automatically change the segment parallelism of a stream based on the current load and a policy that determines when to scale up or down the number of parallel segments. While being a key feature, it also influences how writers and readers interact with the system.

An example of stream auto-scaling is depicted in Fig. 2a. A stream starts at time $t0$ with two parallel segments. If the rate of data written to the stream segments is close to the one defined in the scaling policy (*e.g.*, 1MBps, 100e/s), there will be no changes in the number of segments. However, at time $t1$, the data plane realized a sustained increase in the ingestion rate for segment $s1$. When notified, the control plane *seals* $s1$ (no further writes are allowed) and *splits* it into two new segments (stream scale-up). Note that before $t1$, events written to the stream with a routing key k that hashes to $h(k) \in [0.5, 1)$ belong to $s1$, whereas events written to routing keys that hash to $h(k) \in [0, 0.5)$ belong to $s0$. After $t1$, events with routing key $h(k) \in [0.75, 1)$ are written to $s3$ and those $h(k) \in [0.5, 0.75)$ are written to $s2$. Fig. 2a shows another instance of a scale-up event on $s0$ at time $t2$. As one can infer, this process naturally distributes the load across more segment containers.

Interestingly, segments covering a contiguous range of the key space can also be *merged* in the case they consistently receive low load (stream scale-down). For instance, at time $t3$, $s2$'s range and $s5$'s range are merged into a new segment $s6$ to accommodate a decrease in the load on the stream on that key range. As we show in the next section, Pravega allows clients to work with auto-scaling streams while preserving ordering and consistency guarantees.



(a) Example of stream auto-scaling process. (b) Controller-writer protocol upon scale up. (c) Reader group dynamics under auto-scaling.

Figure 2: Example of stream auto-scaling and its influence on writer and reader interactions with the system.

3.2 Writing to a Stream

The Pravega writer interacts with the control plane to know the stream segments available to write at any given moment, as well as the segment store hosts that run the segment containers owning the segments. With this information, the writer can directly contact the right segment store host to write events for a given segment.

Pravega guarantees that events with the same routing key are read in the order they were written. To materialize this guarantee, Pravega enforces that the assignment of routing keys to segments is consistent even under stream auto-scaling. Between two stream scaling events, all events written to a stream with the same routing key are mapped to the same segment. The control plane builds the metadata that orders segments across scaling events. Let's retake the example in Fig. 2a. The system scales up from one segment $s1$ to segments $s2$ and $s3$. The key space of $s1$ exactly overlaps with the ones of $s2$ and $s3$, but $s2$ and $s3$ have no intersection. In Pravega, segments $s2$ and $s3$ are defined as the *successors* of $s1$. As visible in Fig. 2b, the protocol between the writer and the control plane enforces that no append happens to $s2$ and $s3$ until $s1$ is sealed, and this generalizes to any number of segments before and after a scaling event. Consequently, once segments are sealed due to a scaling event, future events are appended to the successors of the sealed segments, preserving routing key order.

Duplicates or missing events in a stream can be problematic: they can induce incorrect results or incorrect behavior in general. To avoid this problem, writers internally have a *writer id* used to determine the last event written upon a re-connection. When the writer has events to append, it initiates the writing of a *batch of events* (see §4). Once finished appending the batch, the writer sends a "batch end" command with the number of events written and the last event number. The segment store must maintain the last event number for any given writer id to detect duplicates on segments. Specifically, it persists the ⟨`writer id, event number`⟩ pair in a per-segment data structure called *segment attributes* [2] as part of processing the append request. Upon a writer's re-connection, the segment store fetches this attribute and returns the last event number written as part of the handshake with the writer. This response from the segment store enables the writer to resume from the correct event in the case it had appends outstanding.

3.3 Reading from a Stream

Reading a stream requires events to be processed only once, and consequently, a group of readers needs to coordinate the distribution of segments across the group. To enable multiple readers to read one or more streams in a coordinated manner, Pravega introduces the concept of *reader groups*. A reader group RG consists of a set of readers associated to a set of streams S. Moreover, we define $s(r)$ as the set of segments assigned to the reader r, and $c(s)$ is the current set of active segments of a stream (non-sealed segments enabled for reading). In a reader group RG, for each $r \in RG, s(r) \subseteq \cup_{s \in S} c(s)$. At any time and for any two distinct readers $r, r' \in RG, s(r) \cap s(r')$ is empty. Note that this definition does not imply that all segments in $\cup_{s \in S} c(s)$ are assigned to some reader at any time. A reader may have released a segment while no other has acquired it yet or a new segment has not been acquired yet by any reader. The contract specifies that any segment in $\cup_{s \in S} c(s)$ is eventually assigned. As such, the reader group does not guarantee that at any time $\cup_{s \in S} c(s) = \cup_{r \in RG} s(r)$, although we strive for liveness that for all $x \in \cup_{s \in S} c(s)$, eventually x is assigned to some reader.

The assignment of segments to readers in the group is built upon the distributed coordination mechanism we expose in Pravega called *state synchronizer* [27]. The state synchronizer is an API built on top of Pravega streams that enables readers to have a consistent view of a distributed state via optimistic concurrency. Readers use it to agree on changes to the state of the group, *e.g.*, the assignment of segments to readers. The distribution of segments attempts to achieve fairness (*i.e.*, number of segments) across readers.

To guarantee that readers read events with the same routing key in append order, the readers follow a similar procedure as the writers. As an example, the stream in Fig. 2c has a single segment $s1$, and it eventually scales up, resulting in $s1$ splitting into $s2$ and $s3$. Once the reader $r1$ arrives at the end of $s1$, both $r1$ and $r2$ request the successors to the controller, and they start reading from the new segments $s2$ and $s3$. More interestingly, let's inspect what happens to readers $r1$ and $r2$ upon a stream scale-down event. At this point, $r1$ is reading $s2$ and $r2$ is reading $s3$. The segments merge into $s4$ ($s2$ and $s3$ are sealed). Reader $r1$ arrives at end of $s2$ and requests its successors, segment $s4$. As reader $r2$ is not yet done with segment $s2$, reader $r1$ cannot proceed. If either $r1$ or $r2$ proceed to read $s4$ before $r2$ finishes reading $s2$, then we could be breaking our promise of reading events with the same key in append order. Consequently, to satisfy our order guarantee, we put $s3$ on hold until $r2$ flags that it is done with $s2$. Only then $s4$ can be assigned and read.

4 PRAVEGA IO PATH

We next describe the design decisions that allow Pravega to achieve high IO performance regardless of the event size and for high levels of parallelism, as we demonstrate in our experiments.

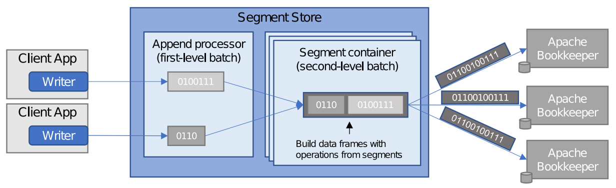


Figure 3: Overview of Pravega's write path and batching.

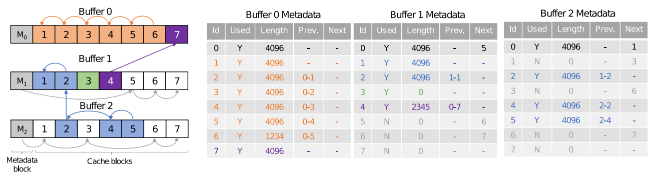
4.1 Write Path

In Fig. 3, we illustrate the Pravega write path. Writers append applications' data and they batch such data to the extent possible. Conversely to other systems that batch data by holding it on the client and waiting to transmit it, the Pravega writer starts sending a batch before it has sufficient data to fill it and batch data is *collected on the server side*. The writer controls batch sizes using a tracking heuristic that performs estimates based on input rate and feedback from the responses. Specifically, the batch size is estimated as the minimum between the defined maximum batch size (*e.g.*, 1MB) and half the server round trip time. With such estimates, the writer determines when to close batches. By doing this, the batch data is held not in the client's memory, but is a mix of data in-flight on the wire and data collected at the server, thus reducing write latency.

In the segment store, every request that modifies a segment is converted into an *operation* and queued up for processing. There are multiple types of operations, each indicating a different modification to the segment (append, truncate, etc.). A segment container has a single, dedicated WAL log to which it writes all operations it receives. Many segments can be mapped to a single segment container, so all operations from a container's segments are *multiplexed into that single log*. This is a crucial design feature that enables Pravega to support a large number of segments, as it does not need to allocate physical resources on a per-segment basis.

The segment container aggregates multiple segment operations into a *data frame* and initiates the WAL write for it. When the WAL append is acknowledged, the segment container asynchronously accepts all the operations in the data frame into its internal state. In this sense, the segment container builds a *second level of batching* by dynamically determining the size of data frames. When the segment container sees that there are no more operations to pick from the processing queue, it uses recent WAL latency information and write sizes to calculate the amount of time to wait as follows: $Delay = RecentLatency \cdot (1 - \frac{AvgWriteSize}{MaxFrameSize})$. The delay to keep adding operations to a data frame is directly proportional to the recent WAL latency and inversely proportional to the recent average write size. If recent data frames had a high fill rate, then the system is already maximizing throughput. On the other hand, if recent data frames were underutilized, it is desirable to wait for a little longer (up to some defined bound), so that more operations may arrive and be batched together, potentially improving throughput.

WAL logs in Pravega are a metadata abstraction built on top of Apache Bookkeeper ledgers. A bookie, which is the Bookkeeper storage server, journals requests to append data to a ledger, and it performs another level of aggregation before appending to its journal. This third level of aggregation is another opportunity to batch data coming from different segment containers.



Buffer 0 Metadata

Id	Used	Length	Prev.	Next
0	Y	4096	-	-
1	Y	4096	-	-
2	Y	4096	0-1	-
3	Y	4096	0-2	-
4	Y	4096	0-3	-
5	Y	4096	0-4	-
6	Y	1234	0-5	-
7	Y	4096	-	-

Buffer 1 Metadata

Id	Used	Length	Prev.	Next
0	Y	4096	-	5
1	Y	4096	-	-
2	Y	4096	1-1	-
3	Y	0	-	-
4	Y	2345	0-7	-
5	N	0	-	6
6	N	0	-	7
7	N	0	-	-

Buffer 2 Metadata

Id	Used	Length	Prev.	Next
0	Y	4096	-	1
1	N	0	-	3
2	Y	4096	1-2	-
3	N	0	-	6
4	Y	4096	2-2	-
5	Y	4096	2-4	-
6	N	0	-	7
7	N	0	-	-

Figure 4: A sample cache layout with 3 cache buffers (each with 8 4-KB blocks) and 4 cache entries stored (in colors).

4.2 Read Path

Readers issue read requests to segment store instances. The *read index* is an essential component of the segment container that provides a complete view of all the data in a segment, both from WAL and LTS, without the reader having to know where such data resides. The read index can randomly access segment data and contains an entry per active segment in the segment container. When a read request is received, the read index returns a read iterator that will return data until the read request parameters are satisfied. The iterator will either fetch data that is immediately available in the in-memory cache, request data from LTS (and bring it to cache) or, if it reached the current end of the segment, return a future that will be completed when new data is added (thus providing *tail reads*). A main data structure of the read index is a sorted index of entries per segment (indexed by their start offsets) which is used to locate the requested data. The index is implemented via a custom AVL search tree [29] to minimize memory usage while not sacrificing access performance. The entries themselves contain some small amount of metadata that is used to locate the data in the cache and to determine usage patterns to favor cache evictions.

The read index is backed by a local *in-memory cache* designed from scratch for Pravega. Traditional cache solutions treat each entry as an immutable blob of data, which poses problems for the append-heavy ingestion workloads that are common in streaming scenarios. Each event appended to a stream would either require its own cache entry or need an expensive read-modify-write operation to be included in the cache.

We divide our cache into equal-sized cache blocks, where each block is uniquely addressable using a 32-bit pointer. Cache blocks are daisy-chained together to form cache entries. Each cache block has a pointer to the block immediately before it in the chain. Since each block has an address, we can choose the address of the last block in the chain to be the address of the entry itself. We can then reference this address from the read index. Note that pointing to the last block enables us to immediately locate that and perform appends, by either writing directly to it (if it still has capacity) or finding a new empty block and adding that to the chain. Similarly to the blocks used in cache entries, empty cache blocks are also chained together, which makes locating an available block an $O(1)$ operation. To prevent memory fragmentation and keep metadata overhead low, the cache pre-allocates during initialization contiguous regions of off-heap memory into cache buffers (*e.g.*, a 2MB buffer can hold 512 4KB blocks). Regarding empty cache blocks, keeping a single list of such blocks across all buffers would quickly run into concurrency issues while modifying it. Thus, we have chosen to only keep a list of empty cache blocks within each buffer (smaller concurrency domain). Across buffers, the cache uses a queue of cache buffers with available blocks that are added and removed based on whether they have available blocks or not. For clarity, Figure 4 depicts a cache with four entries (colored blocks) and three buffers following the described cache layout, along with a tabular representation of the metadata.

4.3 Storage Tiering

WAL is by no means the final destination of data. Instead, one of the key goals of Pravega is to *asynchronously move data to LTS* (*e.g.*, S3, HDFS, NFS) for cost and throughput reasons. And it does so in a unique manner: the storage tiering process is *integrated into the write path*. If LTS is not available or is temporarily slow, Pravega can throttle writers to prevent backlogs of data from growing indefinitely waiting to be moved to LTS.

In the segment container, the *storage writer* is the component in charge of de-multiplexing the operations written to WAL, grouping them by segment, and applying them in LTS. To maximize throughput, it buffers small appends into larger writes to LTS. Once the storage writer flushes a set of operations to LTS, it notifies the segment container that the WAL log can be truncated up to that point. This translates into deleting Bookkeeper ledgers when needed.

The storage writer interacts with the storage subsystem in charge of writing data to LTS and keeps the metadata of segments in LTS. In LTS, Pravega stores *chunks* (*i.e.*, contiguous range of segment bytes) and segments are made up of a sequence of non-overlapping chunks. Note that chunks themselves do not include additional metadata. Similar to the case of the control plane storing the metadata of streams, the metadata of chunks in LTS is also stored in Pravega itself via the key-value tables API. All LTS metadata operations are performed using conditional updates and using transactions to update multiple keys at once [25]. This guarantees that concurrent operations will never leave the metadata in an inconsistent state.

4.4 Failure Handling

Pravega should assume the occurrence of failures in its instances or its dependencies. If a severe error is detected within a segment container (*e.g.*, out of memory) or with a dependency (*e.g.*, Bookkeeper or LTS are unreachable), the segment container shuts down. This implies that no further operation is allowed and it attempts to perform a *recovery*. Recovery is part of the initialization process of a segment container and it consists of i) starting its internal components, and ii) reading the WAL log to rebuild the internal state just before the crash. In fact, the main goal of WAL is to persistently store the operations not yet written to LTS to recover the state of the segment container. It is worth mentioning that the segment container periodically writes a special operation to the WAL log called *metadata checkpoints*, which are snapshots of the segment container metadata at a given point in time. To recover its state, the segment container just needs to read the last metadata checkpoint and sequentially apply the subsequent operations in the WAL log.

If a whole segment store instance crashes, all the segment containers it was running are redistributed across the remaining instances. In this scenario, there can be cases in which more than one segment store instance attempts to run the same segment container, thus leading to potential data corruption. Pravega guarantees that a segment container writes data to WAL and LTS exclusively by: i) keeping the assignment of segment containers to segment stores in a consistent store, *i.e.*, Apache Zookeeper; ii) having segment containers implement fencing [53] to ensure exclusive access to WAL logs [8].

5 EXPERIMENTAL RESULTS

Next, we describe our experiments that focus on: i) Pravega writer performance and the implications of data durability (§5.2); ii) our adaptive batching algorithm (§5.3); iii) writer performance for larger events (§5.4); iv) tail-read performance and the impact of routing key dispersion (§5.5); v) Pravega's ability to handle high segment and writer parallelism (§5.6); vi) historical read performance (§5.7), and vii) stream auto-scaling (§5.8).

5.1 Setup

We summarize here the configuration used in our experiments on AWS (see Table 1). We compare the performance of Pravega against the most popular counterpart systems: Apache Kafka [12] and Apache Pulsar [14]. The workloads in all cases are executed with OpenMessaging Benchmark [20]. The full configuration, data and reproduction steps for the experiments are publicly available [3].

Implementation. Pravega is an open-source CNCF project in sandbox state [21]. The project was open-sourced in 2016 and it provides to the public not only the core storage engine but also deployment tools (*e.g.*, Kubernetes operators) and a connector ecosystem to integrate Pravega with a variety of analytics engines.

Deployment. We employ the same type of EC2 instances for deploying the analogous components of each system. It is especially relevant to mention that for the components in charge of writing data (*i.e.*, brokers, Apache Bookkeeper), we use an instance type with access to local NVMe drives. To evaluate the efficiency of the write path across these systems, we use one drive for the Bookkeeper journal (Pulsar/Pravega) and the Kafka broker log.

Replication. The data replication schemes differ between Kafka (leader-follower model) and Pravega/Pulsar (Bookkeeper replicates across bookies [40]). Still, we can achieve a similar data redundancy layout across them. We configured all the systems to create 3 replicas of every message, requiring at least 2 of such replicas to be confirmed by the servers before considering a write as successful.

Durability. There is an important difference between the default behavior of Kafka versus Pravega/Pulsar on data durability. Kafka by default does not flush data (*i.e.*, `fsync` system call), thus trading-off durability in favor of performance. We also evaluate Kafka with similar durability guarantees that both Pravega and Pulsar satisfy by default (*i.e.*, by setting `flush.messages=1`, `flush.ms=0`).

Storage tiering. Pravega inherently moves ingested data to LTS. In our experiments, we used an NFS volume backed up by an AWS Elastic File Service (EFS) instance. For fairness, we also have enabled the storage tiering plug-in in Pulsar to compare against Pravega unless otherwise stated. We have configured for Pulsar an AWS S3 bucket and defined the ledger rollover to happen within 1 and 5 minutes, so tiering activity occurs while the benchmark takes place. We also set Pulsar topics to start the offloading process immediately (`setOffloadThreshold=0`), as well as to remove the data from Bookkeeper as soon as it is migrated to long-term storage

	Pravega	**Kafka**	**Pulsar**
Version	Pravega 0.9.0, Bookkeeper 4.11.1	Kafka 2.6.0	Pulsar 2.6.0, Bookkeeper 4.11.1
Replication	ensemble=3, writeQuorum=3, ackQuorum=2	replication=3, acks=all, min.insync.replicas=2	ensemble=3, writeQuorum=3, ackQuorum=2
Durability	Yes (default)	No (default)	Yes (default)
Tiering	Yes (AWS EFS)	No	Yes (AWS S3)
Instances	Controller (m5.large)=1, Segment Store + Bookie (i3.4xlarge)=3, Zookeeper (t3.small)=3, Benchmark (c5.4xlarge)=2	Broker (`i3.4xlarge`)=3, Zookeeper (`t3.small`)=3, Benchmark (`c5.4xlarge`)=2	Broker + Bookie (`i3.4xlarge`)=3, Zookeeper (`t3.small`)=3, Benchmark (`c5.4xlarge`)=2
Journal Drives	1 NVMe	1 NVMe	1 NVMe
Client Batching	Yes (dynamic)	Yes (time/size based)	Yes (time/size based)

Table 1: Experiments configuration unless otherwise stated.

(`setOffloadDeleteLag=0`). At the time of this writing, Kafka did not provide storage tiering in its open-source edition.

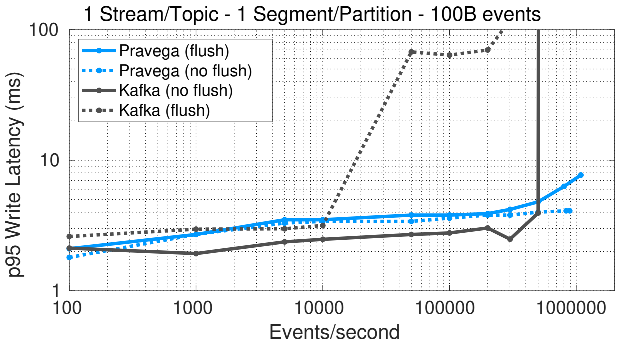
Routing keys. By default, our workloads use (random) routing keys on writes. We use routing keys in our workloads to ensure per-key event order, frequently a requirement of streaming applications for correctness. We also execute workloads without routing keys to understand the potential impact of routing keys on performance.

Client configuration. Pulsar and Kafka clients implement a batching mechanism that can be parameterized via "knobs" that enable the producer to buffer a certain number of messages or wait until some timeout before performing the actual write against the broker. The goal of this feature is to improve a producer's throughput for small messages, despite inducing extra latency in scenarios where the workload is not throughput-oriented. By default, we use in both systems a similar configuration: 128KB as batch size and 1ms as batch time. We also compare the behavior of the Pulsar producer with and without this feature, as well as the impact of larger batches in Kafka. We did not enable compression in the Pulsar/Kafka clients as such techniques may have an important role in performance depending on the data at hand [36].

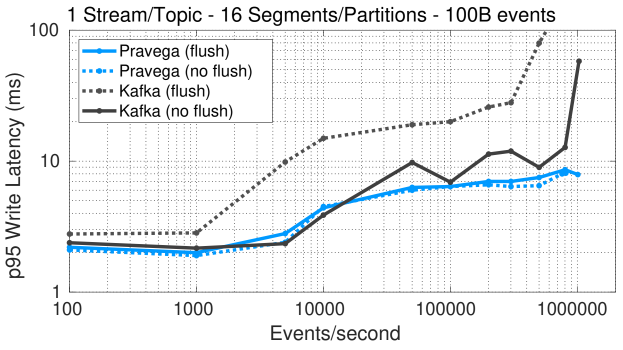
Workloads. We are interested in understanding the behavior of these systems based on two parameters: *event size* and *number of partitions/segments*. We use event sizes from 100B to 10KB, as they can be considered typical in many streaming applications (*e.g.*, IoT, logs, social media posts). We configure our experiments to run OpenMessaging Benchmark [20] producer and consumer threads distributed across the benchmark VMs. Each of the producer and consumer threads uses a dedicated Kafka, Pulsar, or Pravega client instance. Benchmark producer threads use producers (Kafka and Pulsar) or writers (Pravega), while benchmark consumer threads use consumers (Kafka and Pulsar) or readers (Pravega).

5.2 Writer Performance and Data Durability

It is natural for an application to expect that data is available for reading once writing the data is acknowledged, despite failures. Durability is critical for applications to reason about correctness. While Pravega provides durability by default (*i.e.*, flushing data upon acknowledgement), this is not the case for Kafka. Not enabling data durability may be undesirable for enterprise applications as correlated failures do happen and servers do not always stop gracefully [58].



(a) 1 segment/partition, 1 writer/producer.



(b) 16 segments/partitions, 1 writer/producer.

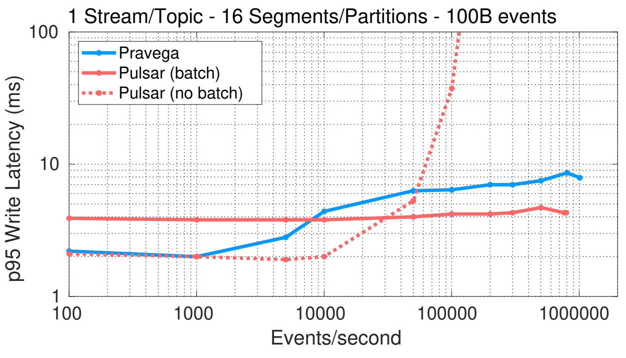
Figure 5: Impact of data durability on write performance.

In Fig. 5, we show latency and throughput to compare the performance implications of non-default durability options. Specifically, we run a set of experiments comparing enabling and disabling durability in Pravega (disabling journal flushes in Bookkeeper, namely "`no flush`") and Kafka (by setting `flush.messages=1`, `flush.ms=0` to enable durability, namely "`flush`"). For simplicity, these experiments are executed with a single producer/writer.

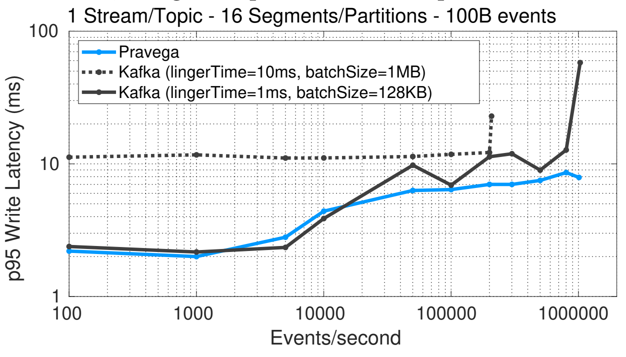
Visibly, Fig. 5a shows that for a stream/topic with one segment/partition, the Pravega writer (`flush`) reaches a maximum throughput 73% higher than Kafka (`no flush`). Note that this performance improvement in Pravega is achieved while guaranteeing data durability. Following this comparison, for 16 segments/partitions (see Fig. 5b) the maximum throughput of both Pravega and Kafka is similar (over 1 million events/second for a single writer/producer).

Note that enforcing data durability for Kafka (`flush`) has a major performance impact on write latency. This is especially visible for moderate/high throughput rates. Kafka flushes messages according to a time (`log.flush.interval.ms`) or a message (`log.flush.interval.messages`) interval. When the message interval is set to 1, all messages are flushed individually before being acknowledged, inducing a significant performance penalty. With Bookkeeper, data is persisted before being acknowledged, but they are opportunistically grouped upon flushes [40].

Regarding write latency, Kafka (`no flush`) only gets consistently lower (≈ 1ms at p95) values than Pravega (`flush`) for 1 segment/partition up to 500k e/s. In the rest of the cases, the Pravega writer shows lower latency than Kafka. The performance gain for

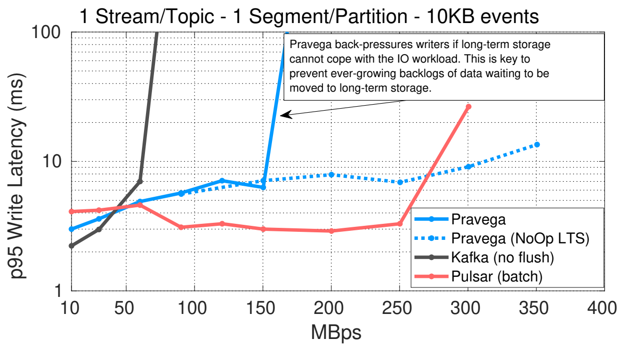


(a) 16 segments/partitions, 1 writer/producer.

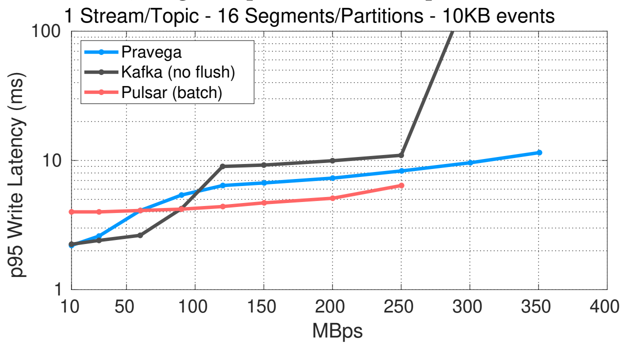


(b) 16 segments/partitions, 1 writer/producer.

Figure 6: Evaluation of client batching strategies.



(a) 1 segment/partition, 1 writer/producer.



(b) 16 segments/partitions, 1 writer/producer.

Figure 7: Write performance for larger events.

Pravega of not flushing data to the drive in Bookkeeper writes is modest, which justifies providing durability by default.

Summary: The Pravega writer achieves good write performance compared to the Kafka producer while guaranteeing data durability.

5.3 Evaluating Batching Strategies

Batching enables a trade-off between throughput and latency. Ideally, applications should not need to reason about convoluted parameters to benefit from batching. Instead, it should be the work of the client along with the server to perform this task. The dynamic batching heuristic the Pravega writer implements has the goal of calibrating the batch sizes over time.

In Fig. 6, we plot latency and throughput to understand the impact of Pravega writer batching on performance. We compare Pravega against Pulsar and Kafka (`no flush` by default), which are systems that require an application to choose whether to use batching or not, and configure it accordingly.

Fig 6a shows that the Pulsar producer is able to target either low latency or high throughput, but not both. This forces the user to choose between a latency-oriented (namely `no batch`) or a throughput-oriented configuration (namely `batch`). In contrast, the Pravega writer achieves both lower write latency than Pulsar (`batch`) for the lower-end throughput rates (*e.g.*, < 10k e/s) and higher maximum throughput than Pulsar (`no batch`).

Interestingly, increasing the batch size and the wait time for Kafka (10ms linger time, 1MB batch size) to enable more batching has the opposite expected effect (see Fig. 6b). The throughput drops compared to the default configuration with 1ms linger time and 128KB batch size. To understand this result, we inspected the maximum throughput of a Kafka producer writing to a 16-partition topic and using the same batch configuration, but not using routing keys when writing data. In this case, the client achieved a throughput 6*x* higher (120MBps). We consequently attribute the lower batching performance observed to the use of (random) routing keys. We focus on this aspect specifically in §5.5.

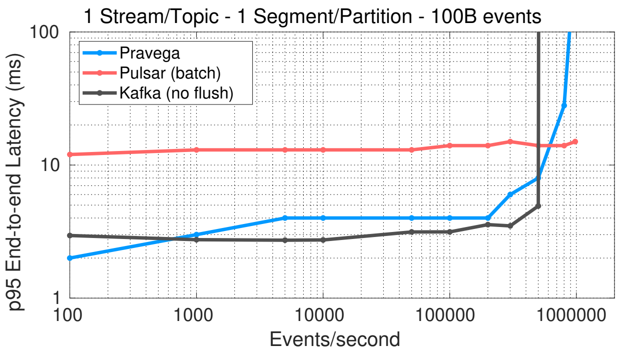
Summary: Dynamic batching in Pravega allows writers to strike an excellent balance between latency and throughput. More importantly, Pravega does not require the user to decide the performance configuration of the writer ahead of time.

5.4 Writer Performance for Large Events

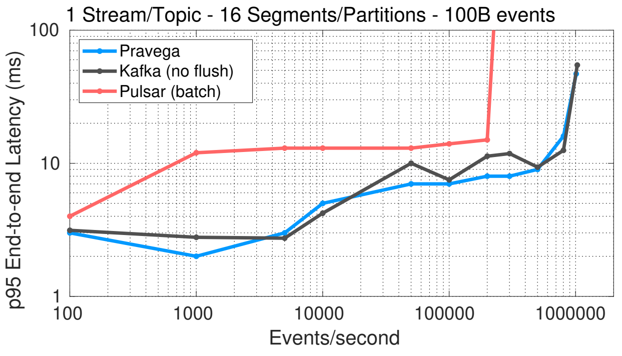
Events are often small in real applications (*e.g.*, < 1KB), but there are also use cases using larger events. For such scenarios, batching is less effective, and the key metric is the byte throughput. In Fig. 7, we use 10KB events and we compare latency and byte throughout of the Pravega writer against both Pulsar and Kafka producers.

For a single-segment/partition stream/topic (see Fig. 7a), Pravega (160MBps) and Pulsar (300MBps) writers achieve a much higher write throughput compared to Kafka (70MBps). In the case of 16 segments/partitions (see Fig. 7b), Pravega shows the highest throughput (350MBps) compared to Kafka (330MBps) and Pulsar (250MBps).

However, the Pravega writer cannot get more than 160MBps for the single-segment case. The reason is that it is bottlenecked by the movement of data to LTS (AWS EFS). To validate this statement,



(a) 1 segment/partition, 1 writer/producer, 1 reader/consumer.



(b) 16 segments/partitions, 1 writer/producer, 1 reader/cons.

Figure 8: Performance of a tail readers/consumers.

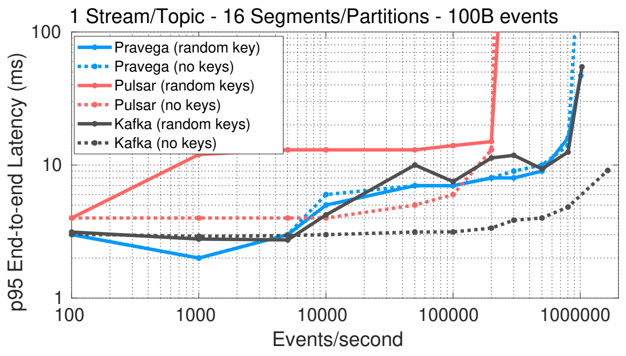


Figure 9: Impact of routing keys on read performance.

we have conducted an experiment using a test feature that allows Pravega to write only metadata to LTS and no data (namely, NoOp LTS). From the results in Fig. 7a, skipping the data writes to LTS enables much higher throughput for Pravega. Pulsar performs better with storage tiering enabled for the single-segment case because it does not throttle producers when LTS is saturated. That is, storage tiering is not an integral part of Pulsar's ingestion pipeline as it is in Pravega. While this might be seen as an advantage now, not throttling writers if LTS is saturated can lead to an ever-growing backlog of data waiting to be moved to LTS. The problem becomes evident when we present historical read results in §5.7.

Summary: The Pravega writer achieves high write throughput when using multiple segments. The throughput of Pravega depends on the throughput capacity of LTS, and in the case LTS saturates, Pravega applies backpressure to avoid building a backlog of data.

5.5 Tail Reads and Routing Keys

For many applications, the time must be short between the event being generated and the time that it is available for reading and processing. We refer to such a time interval as end-to-end latency. In Fig. 8, we plot the end-to-end latency and throughput for 100B events of Pravega, Pulsar, and Kafka readers/consumers.

In Fig. 8a, Pravega and Kafka exhibit lower end-to-end latency compared to Pulsar up to the saturation point. In fact, Pulsar does not achieve end-to-end latency values under 12ms (95th percentile), even with batching. On the other hand, read throughput for a stream/topic with a single segment/partition is much higher for Pravega (72%) and Pulsar (56%) than for Kafka.

Interestingly, in the case of 16 segments/partitions (see Fig. 8b), Pulsar shows a 76% drop in read throughput than the single partition case, despite configuring one consumer thread per segment/partition in all systems (higher read parallelism). This may be a limiting factor in highly-parallel scenarios. In the case of Kafka and Pravega, managing more segments increases end-to-end latency for medium to high throughput rates.

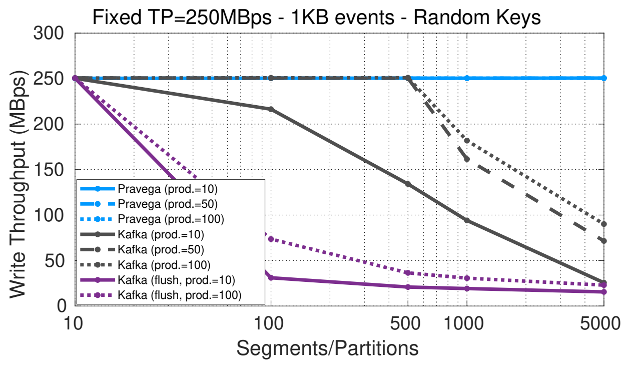
We also want to understand the impact of using routing keys when writing and reading data from these systems. This is important as per-key event ordering is desirable for a number of applications to enable the correct processing of the events while providing parallelism. All of Pravega, Pulsar and Kafka use routing keys and guarantee total order per key. In Fig. 9 we depict the difference in performance of the readers based on whether no routing keys or random routing keys are used in the workload.

Fig. 9 shows that using random routing keys across several topic partitions induces a significant read latency overhead in Pulsar compared to not using routing keys (*e.g.*, 3.25*x* higher p95 end-to-end latency at 10k e/s). Still, Pulsar's read throughput remains the same even without using routing keys, which indicates that the performance limitation has another root cause. Similarly, when Kafka does not guarantee order or durability, read (and write) throughput is 59.6% higher. In contrast, Pravega performance is consistent irrespective of the use of routing keys. As many applications use routing keys for ordering purposes, it is crucial to highlight the performance differences induced by routing key access distributions.

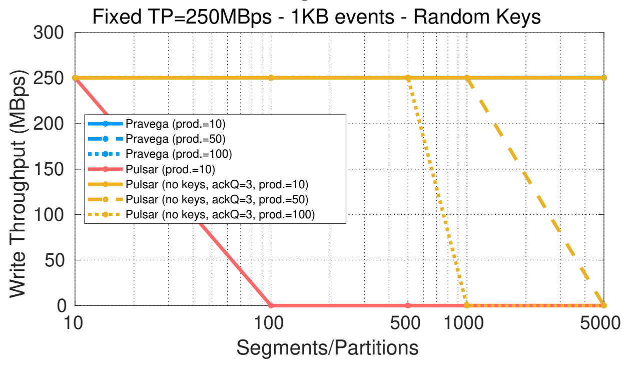
Summary: The Pravega reader achieves both low end-to-end latency and high throughput compared to Kafka and Pulsar for the cases tested. Pravega is virtually insensitive to the distribution of routing keys, as opposed to the other systems.

5.6 Handling High Parallelism

Next, we focus on the performance evaluation of Pravega in the presence of multiple writers appending to streams with many segments. We are interested in the append path, which is critical for ingesting streaming data effectively. We chose to fix an ingest workload rate of 250MBps (1KB events) and show how the different systems behave when varying the number of writers/producers and segments/partitions. Also, this set of experiments slightly differs



(a) Pravega vs Kafka.



(b) Pravega vs Pulsar.

Figure 10: Impact of parallelism on write performance.

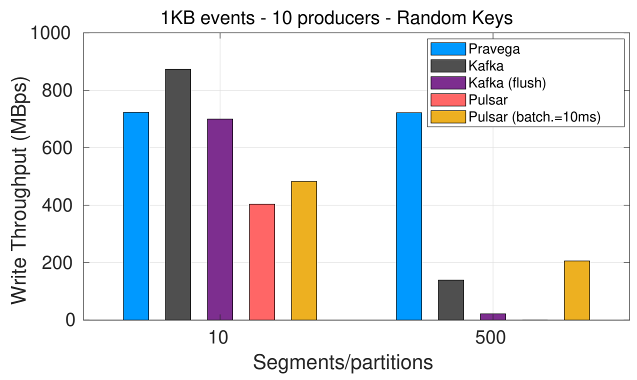


Figure 11: Max throughput achieved by systems under test.

in deployment compared to the previous ones: i) the number of benchmark instances is 10 to support multiple clients (instead of 2); ii) to prevent CPU bottlenecks we use `i3.16xlarge` instances in segment stores, brokers, and bookies (instead of `i3.4xlarge`). In this experiment, Pulsar does not perform storage tiering.

Fig. 10 shows throughput for Pravega, Kafka, and Pulsar with a varying number of stream segments and producers. Each line corresponds to a workload with a different number of producers appending to a single stream/topic. For Kafka and Pulsar, we also plot lines for alternative configurations that give more favorable results to those systems, at the cost of functionality.

Visibly, Fig. 10 shows that Pravega is the only system able to sustain the target throughput rate of 250MBps for streams with up to 5k segments in a stream and 100 writers. It suggests that the design of the append path of Pravega, and specifically, the batching and multiplexing of small appends from many writers and segments at segment containers, is efficiently handling workload parallelism.

The Kafka throughput drops as we increase the number of topic partitions (see Fig. 10a). Adding producers for Kafka yields higher throughput up to a limit. There is a significant difference between 10 and 50 producers, while between 50 and 100 producers, the throughput difference for Kafka is marginal. This result is likely because the lack of partition multiplexing in Kafka. To wit, high levels of write parallelism directly translate into an equivalent number of log files writing to the drive that can lead to degraded performance. Furthermore, when we enforce durability in Kafka (`flush`), throughput is much lower (*e.g.*, −80% for 100 producers and 500 partitions). While some penalty is expected from flushing messages to the drive, this experiment shows that enforcing durability for more than ten topic partitions penalizes throughput significantly.

Unfortunately, Pulsar crashed in most configurations we have experimented with (see Fig. 10b). To understand the root cause of Pulsar's stability problems, we tried a more favorable configuration that: i) waits for all acknowledgments (`ackQ=3`) from bookies to prevent out-of-memory errors; ii) do not use routing keys to write events (*i.e.*, sacrifices event ordering and reduces the actual parallelism on writes). With this new configuration, Pulsar can get better results compared to the base scenario. However, it is still showing degraded performance and eventual instability when the experiment reaches high parallelism, especially when increasing the number of producers. Note that not using routing keys on writes seems to be the main contributor to Pulsar's improvement with the favorable configuration.

While we have used a fixed target rate in our previous experiments, we also want to understand the maximum throughput that these systems can achieve in our scenario. To narrow down the analysis, in Fig. 11 we pick 10 and 500 segments/partitions as a baseline, along with 10 producers and 1KB events.

Pravega can get a maximum throughput of 720MBps from the benchmark perspective for both 10 and 500 segments, translating into roughly 780MBps at the drive level. The difference is due to the metadata overhead added by Pravega (*e.g.*, segment attributes) and Bookkeeper. Note that this is very close to the maximum throughput we can get with synchronous writes on the drives used (we measured it via dd tool to approximately 800MBps). This confirms that Pravega (and Bookkeeper) can efficiently use the drives.

For Pulsar, with the defined configuration, we can reliably get almost 400MBps of throughput at the benchmark level. We have also explored increasing the client batching time to 10ms, which translates into a moderate improvement in throughput (20%). Still, we observe that this is far from the maximum capacity of drives, and we suspect that it is due to the use of routing keys, as it reduces the batching opportunities for Pulsar clients. Even worse, we also see that the Pulsar throughput is significantly limited as we increase the number of partitions. This result suggests that relying mainly on the client for aggregating data has important limitations.

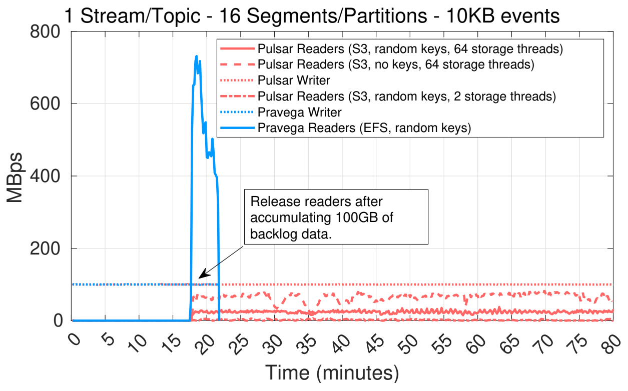


Figure 12: Historical read performance.

For the 10-partition case, we observe that Kafka can achieve up to 700MBps and 900MBps, when it guarantees durability and when it does not, respectively. In the latter case, writing to "page cache" and letting the OS write data in larger blocks to the drive helps to get a higher maximum throughput. But note that this only happens for low parallelism, as for 500 segments, the throughput drops dramatically to 22MBps and 140MBps, respectively.

Summary: Pravega is the only system that achieves consistent throughput for the number of producers and segments tested, while guaranteeing event ordering and data durability. Also, it efficiently exploits drive throughput irrespective of the degree of parallelism.

5.7 Historical Read Performance

In this set of experiments, we analyze the performance of readers when requesting historical data from LTS in Pravega and Pulsar (at the time of this writing, Kafka did not provide this functionality in open source). We designed the experiment as follows. OpenMessaging Benchmark has an option that holds readers until writers have written the amount of events specified. Readers are subsequently released, and the experiment is complete when the backlog of events is consumed. We exercised this option by configuring writers to write 100MBps (10KB events) to a 16-partition/segment topic/stream at a constant rate until achieving a backlog of 100GB. Note that writers continue to write data when readers are released, so readers should read faster to eventually catch up.

In Fig. 12, we observe that Pravega achieves much higher historical read throughput than Pulsar by exploiting parallel chunk reads (peaking at 731MBps). For Pulsar, none of the configurations tested resulted in a historical read throughput higher than the write throughput. While parameters like the number of offloading threads or the routing keys used in Pulsar influence the performance of historical reads, we did not find clear guidelines for users to configure tiered storage. We also have discarded that the reason for the read performance difference is due to the LTS system used, as we tested both EFS and S3 to achieve very similar throughput rates for file/object transfers (*i.e.*, 160MBps approx.).

Another interesting observation from Fig. 12 is the following: Pulsar does not throttle writers in case LTS cannot absorb the ingestion throughput, which may lead to situations of imbalance across storage tiers. To wit, if writers write to Bookkeeper faster than the brokers offload data to LTS, the backlog of events waiting to be moved to LTS would grow without bounds. In the long run, this may have negative consequences for users who rely on timely data offloading to LTS.

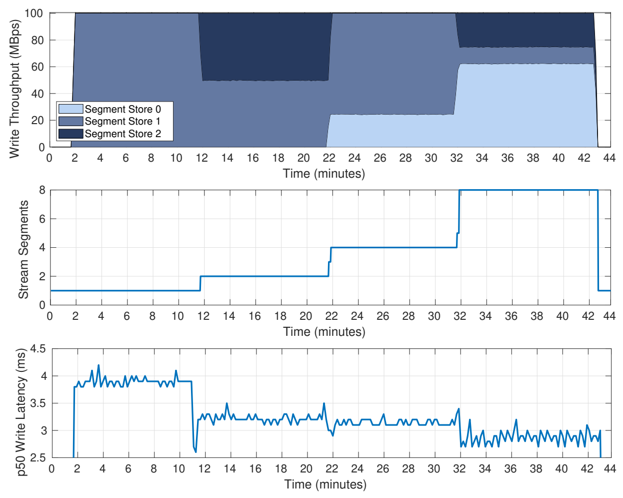


Figure 13: View of stream auto-scaling role on performance.

Summary: Both Pravega and Pulsar provide means to move historical data to long-term storage. However, Pravega achieves much higher historical read throughput compared to Pulsar without user intervention or complicated configuration settings.

5.8 Stream Auto-scaling

Accommodating workload fluctuations over time through stream auto-scaling is a unique feature of Pravega. The absence of this feature in systems like Pulsar and Kafka, which are primarily on the front line of big data ingestion, may induce considerable operational pain to users, particularly at scale.

We focus in this section on the performance implications of stream auto-scaling. We configured auto-scaling in the test stream and we set a target rate of events per second on segments to 2k (or 20MBps, given that we used 10KB events). The benchmark tool wrote at a speed of 100MBps and the stream initially had one segment. Note that to generate some of the plots below we used the Pravega metrics exports. The three plots in Fig. 13 show different aspects of stream auto-scaling in Pravega: i) the write workload per segment store, ii) the number of segments in the stream, and iii) the write latency (p50) perceived by the benchmark instance.

Fig. 13 reveals that as the stream splits and adds more segments, the load is distributed across the available segment stores, thus causing latency to drop. As in this experiment all the segments receive the same number of writes, the actual load distribution across segment stores mainly depends on the placement of segments across segment containers. As the actual placement strategy of segments is stateless (based on consistent hashing), for a small number of segments there may be situations of load imbalance. This is the reason why not all the segment stores receive the same amount of load as stream auto-scaling progresses. Fortunately, a larger number of segments increases the probability of an even distribution of load across segment stores and segment containers [35].

Summary: Pravega is the first streaming storage system that provides elastic streams: data streams that are automatically repartitioned according to the ingestion load and the scaling policy.

6 RELATED WORK

To solve the administration-related costs of the Lambda Architecture [42], in recent years there have been numerous efforts from the research community and the industry to provide unified data processing frameworks for streaming and batch analytics [30, 31, 39, 52, 59]. Currently, systems like Apache Flink [11], Apache Spark [16], and Apache Druid [10] are, among others, standard data processing engines that allow users to run seamlessly streaming and batch analytics via clear abstractions and APIs. However, there is significantly less research work focused on the storage challenges of unifying access to tail and historical stream data [49]. This is precisely the gap that Pravega aims to fill. One of the key contributions of Pravega is to unify tail and historical access to data streams, both at the API and infrastructure levels.

Pravega advocates for decoupling storage and processing concerns in streaming pipelines. In a production cluster [32], Pravega handles data ingestion and storage tasks, whereas stream processing engines like Apache Flink [23] or Apache Spark [26] execute analytics jobs. Therefore, Pravega is a storage substrate for stream processing engines. This approach differs from messaging systems like Apache Kafka [43] or Apache Pulsar [14], which couple both storage and event processing concerns in the same system. Similarly, while Pravega streams can be used as a data source for SQL queries, there should be an external system taking care of such tasks (*e.g.*, Presto/Trino [4]). This also differs from databases that offer the stream abstraction, such as InfluxDB [19] or Cassandra [9], which can ingest events and execute queries over ingested data.

With the need for persistently storing streaming data, traditional queuing systems were replaced by streaming or messaging systems [33], such as Apache Kafka [12, 43], Apache Pulsar [14], and more recently, RedPanda [28]. Naturally, Pravega shares some commonalities with these systems as they all aim at satisfying key requirements not only related to storage but also serving data processing frameworks [55]. That is, all of them expose an (unbounded) topic or stream abstraction to clients, which is internally implemented as a sequence of (bounded) partitions or segments. All these systems can also support key-based event ordering and data durability, despite the assumption of the default durability model differs among them. Moreover, they converge in providing a scalable distributed architecture —consisting of clients and server instances— that takes especial care of IO path performance.

However, Pravega departs from a unique set of assumptions regarding data ingestion and storage that led to key differences in the system design: i) One key design decision in Pravega is to not constrain the retention of streaming data. Data can be retained for as long as there is storage capacity in the scale-out LTS, which is the primary storage for streaming data. This design decision leads to Pravega making storage tiering a first-class element of the ingestion path, whereas other systems provide it as an afterthought and perform a best-effort offloading of partitions to external storage; ii) Handling high parallelism is another key assumption in Pravega. The unit of parallelism in Pravega is the segment container, which aggregates data from multiple segments to the same WAL log file (*i.e.*, segment multiplexing). Conversely, other systems may not exploit multiplexing to that extent, which can limit their parallelism due to saturating underlying drives with too many parallel writes [48]; iii) Being able to automatically handle fluctuating workloads led to the design of the auto-scaling feature in Pravega (rule 7 in [55]), which no other system provides. Despite the above list of distinguishing design decisions is not exhaustive, we believe it helps to understand our contributions in Pravega.

In this paper, we also provide a storage-centric performance evaluation of Pravega compared to Kafka and Pulsar. While there are works devoted to benchmarking stream processing systems [41, 46, 47, 51, 56], they are mostly focused on the performance of compute pipelines, thus overlooking the storage performance aspects of managing data streams. In [33], the authors perform a complete comparison of RabbitMQ against Kafka. Authors in [34] go further by including in their analysis Pulsar as well as newer queuing systems, such as RocketMQ and ActiveMQ. We believe that this work complements existing performance evaluations by contributing insights into storage-related aspects of these systems that have not been evaluated before, such as historical reads and the role of routing keys on performance, to name a few.

7 CONCLUSIONS

Streaming workloads impose stringent data ingestion and storage requirements that event streaming systems need to meet, such as long data retention, high parallelism, elasticity, durability and consistency, all of it with little to no performance penalty. Existing messaging systems presently fall short of these requirements, which motivated us to design and develop Pravega.

In this paper, we have highlighted key design decisions that make Pravega depart from existing systems designed for messaging. One in particular is the introduction of tiered storage, as Pravega is the first system to propose and implement it. The main consequence of such a design choice is the fundamental move from messaging systems that buffer ingested streaming data to real stream storage, which enables virtually unbounded data retention periods. Pravega makes additional contributions to the design of streaming storage, such as the multiple levels of batching, segment multiplexing, and stream auto-scaling, while providing high performance. Our experiments on AWS show that Pravega can deliver performance improvements over Apache Kafka and Apache Pulsar in multiple scenarios, indicating that Pravega is an attractive streaming storage substrate for stream processing engines and applications.

ACKNOWLEDGMENTS

We thank Andrei Paduroiu, Srikanth Satya, Shivesh Ranjan, Sandeep Shridhar, and the rest of the Pravega community for their invaluable contributions. We also thank the Cloud Native Computing Foundation (CNCF) for sponsoring the Pravega project. We thank the anonymous reviewers for their insightful comments on this paper. This work has received funding from the European Union Horizon Europe research and innovation programme under grant agreements 101092644 (NEARDATA) and 101092646 (CLOUDSKIN).

REFERENCES

[1] 2016. Blink: How Alibaba Uses Apache Flink. https://www.ververica.com/blog/blink-flink-alibaba-search.

[2] 2019. Pravega Blog - Segment Attributes. https://cncf.pravega.io/blog/2019/11/21/segment-attributes/.

[3] 2020. Pravega - Performance Blog Post. https://cncf.pravega.io/blog/2020/10/01/when-speeding-makes-sense-fast-consistent-durable-and-scalable-streaming-data-with-pravega.

[4] 2021. Trino - Episode 28: Autoscaling streaming ingestion to Trino with Pravega. https://trino.io/episodes/28.html.

[5] 2022. Market Guide for Event Stream Processing. https://www.gartner.com/en/documents/4347499.

[6] 2023. Amazon Kinesis. https://aws.amazon.com/es/kinesis.

[7] 2023. Apache Bookkeeper. https://bookkeeper.apache.org.

[8] 2023. Apache Bookkeeper - Protocol. https://bookkeeper.apache.org/docs/development/protocol.

[9] 2023. Apache Cassandra. https://cassandra.apache.org.

[10] 2023. Apache Druid. https://druid.apache.org.

[11] 2023. Apache Flink. https://flink.apache.org.

[12] 2023. Apache Kafka. https://kafka.apache.org.

[13] 2023. Apache Kafka - Documentation. https://kafka.apache.org/documentation.

[14] 2023. Apache Pulsar. https://pulsar.apache.org.

[15] 2023. Apache Pulsar - Overview of tiered storage. https://pulsar.apache.org/docs/2.11.x/tiered-storage-overview.

[16] 2023. Apache Spark. https://spark.apache.org.

[17] 2023. Apache Zookeeper. https://zookeeper.apache.org.

[18] 2023. Dell Streaming Data Platform. https://www.dell.com/en-us/dt/storage/streaming-data-platform.htm.

[19] 2023. InfluxDB. https://www.influxdata.com.

[20] 2023. OpenMessaging Benchmark. https://github.com/openmessaging/benchmark.

[21] 2023. Pravega. https://cncf.pravega.io.

[22] 2023. Pravega - ByteStream API Javadoc. https://pravega.io/docs/latest/javadoc/clients/io/pravega/client/byteStream/package-summary.html.

[23] 2023. Pravega - Flink Connector. https://github.com/pravega/flink-connector.

[24] 2023. Pravega - KeyValueTable Javadoc. https://cncf.pravega.io/docs/latest/javadoc/clients/io/pravega/client/tables/KeyValueTable.html.

[25] 2023. Pravega - Simplified LTS. https://github.com/pravega/pravega/wiki/PDP-34-(Simplified-Tier-2).

[26] 2023. Pravega - Spark Connector. https://github.com/pravega/spark-connectors.

[27] 2023. Pravega - StateSynchronizer Javadoc. https://cncf.pravega.io/docs/latest/javadoc/clients/io/pravega/client/state/StateSynchronizer.html.

[28] 2023. RedPanda. https://redpanda.com.

[29] Georgii Maksimovich Adeleson-Velskii and Evgenii Mikhailovich Landis. 1962. An algorithm for organization of information. In *Doklady Akademii Nauk*, Vol. 146. Russian Academy of Sciences, 263–266.

[30] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A framework for integrating batch and online mapreduce computations. *VLDB Endowment* 7, 13 (2014), 1441–1451.

[31] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[32] Dell Technologies. 2023. Dell Streaming Data Platform: Architecture, Configuration, and Considerations. https://www.delltechnologies.com/asset/en-sg/products/storage/industry-market/h18162-streaming-data-platform-architecture.pdf.

[33] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In *ACM DEBS'17*. 227–238.

[34] Guo Fu, Yanfeng Zhang, and Ge Yu. 2020. A fair comparison of message queuing systems. *IEEE Access* 9 (2020), 421–432.

[35] Gaston H Gonnet. 1981. Expected length of the longest probe sequence in hash code searching. *J. ACM* 28, 2 (1981), 289–304.

[36] Raúl Gracia-Tinedo, Danny Harnik, Dalit Naor, Dmitry Sotnikov, Sivan Toledo, and Aviad Zuck. 2015. SDGen: Mimicking datasets for content generation in storage benchmarks. In *USENIX FAST'15*. 317–330.

[37] Raúl Gracia-Tinedo, Yongchao Tian, Josep Sampé, Hamza Harkous, John Lenton, Pedro García-López, Marc Sánchez-Artigas, and Marko Vukolic. 2015. Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end. In *ACM IMC'15*. 155–168.

[38] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems.. In *USENIX ATC'10*, Vol. 8.

[39] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A survey of distributed data stream processing frameworks. *IEEE Access* 7 (2019), 154300–154316.

[40] Flavio P Junqueira, Ivan Kelly, and Benjamin Reed. 2013. Durability with bookkeeper. *ACM SIGOPS operating systems review* 47, 1 (2013), 9–15.

[41] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *IEEE ICDE'18*. 1507–1518.

[42] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. 2015. Lambda architecture for cost-effective batch and speed big data processing. In *IEEE International Conference on Big Data*. 2785–2792.

[43] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *NetDB'11*, Vol. 11. 1–7.

[44] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *ACM SIGMOD'15*. 239–250.

[45] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. 2008. Measurement and Analysis of Large-Scale Network File System Workloads. In *USENIX ATC'08*, Vol. 1. 5–2.

[46] Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, and Otto Carlos MB Duarte. 2016. A performance comparison of open-source stream processing platforms. In *IEEE GLOBECOM'16*. 1–6.

[47] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. 2014. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *IEEE/ACM International Conference on Utility and Cloud Computing*. 69–78.

[48] Ovidiu-Cristian Marcu, Alexandru Costan, Bogdan Nicolae, and Gabriel Antonin. 2021. Virtual Log-Structured Storage for High-Performance Streaming. In *IEEE CLUSTER'21*. 135–145.

[49] John Meehan, Cansu Aslantas, Stan Zdonik, Nesime Tatbul, and Jiang Du. 2017. Data Ingestion for the Connected World. In *CIDR'17*, Vol. 17. 8–11.

[50] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. 2018. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials* 20, 4 (2018), 2923–2960.

[51] Hamid Nasiri, Saeed Nasehi, and Maziar Goudarzi. 2019. Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities. *Journal of Big Data* 6 (2019), 1–24.

[52] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *VLDB Endowment* 10, 12 (2017), 1634–1645.

[53] Alan Robertson. 2001. Resource fencing using STONITH. *White Paper, August* (2001).

[54] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. 2015. Edge analytics in the internet of things. *IEEE Pervasive Computing* 14, 2 (2015), 24–31.

[55] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *ACM SIGMOD'05* 34, 4 (2005), 42–47.

[56] Giselle Van Dongen and Dirk Van den Poel. 2020. Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1845–1858.

[57] Shusen Yang. 2017. IoT stream processing and analytics in the fog. *IEEE Communications Magazine* 55, 8 (2017), 21–27.

[58] Nezih Yigitbasi, Matthieu Gallet, Derrick Kondo, Alexandru Iosup, and Dick Epema. 2010. Analysis and modeling of time-correlated failures in large-scale distributed systems. In *IEEE/ACM International Conference on Grid Computing*. 65–72.

[59] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *USENIX HotCloud'10* 10, 10-10 (2010), 95.