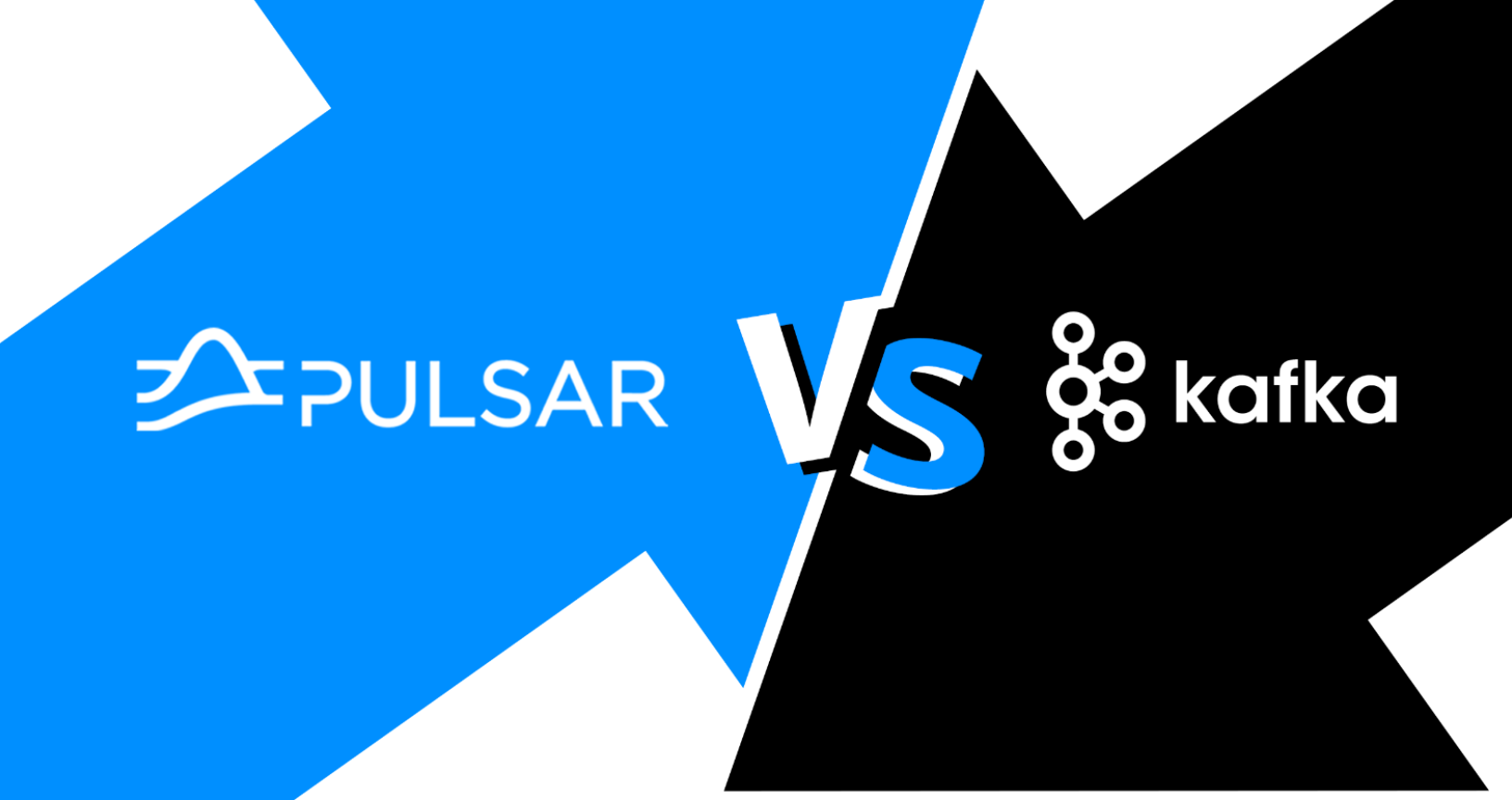


Apache Pulsar™ vs. Apache Kafka® 2022 Benchmark

Matteo Merli
Apache Pulsar PMC Chairperson, CTO at StreamNative

Penghui Li
Apache Pulsar PMC, Engineering Lead at StreamNative

Stream Native

Table of Contents

Executive Summary

As we move into 2022, the Apache Pulsar™ versus Apache Kafka® debate continues. Organizations often make comparisons based on features, capabilities, size of the community, and a number of other metrics of varying importance. This report focuses purely on comparing the technical performance based on benchmark tests.

The last widely published Pulsar versus Kafka benchmark was performed in 2020, and a lot has happened since then. In 2021, Pulsar ranked as a Top 5 Apache Software Foundation project and surpassed Apache Kafka in monthly active contributors as shown in the chart below. Pulsar also averaged more monthly active contributors than Kafka for most of the past 18 months.

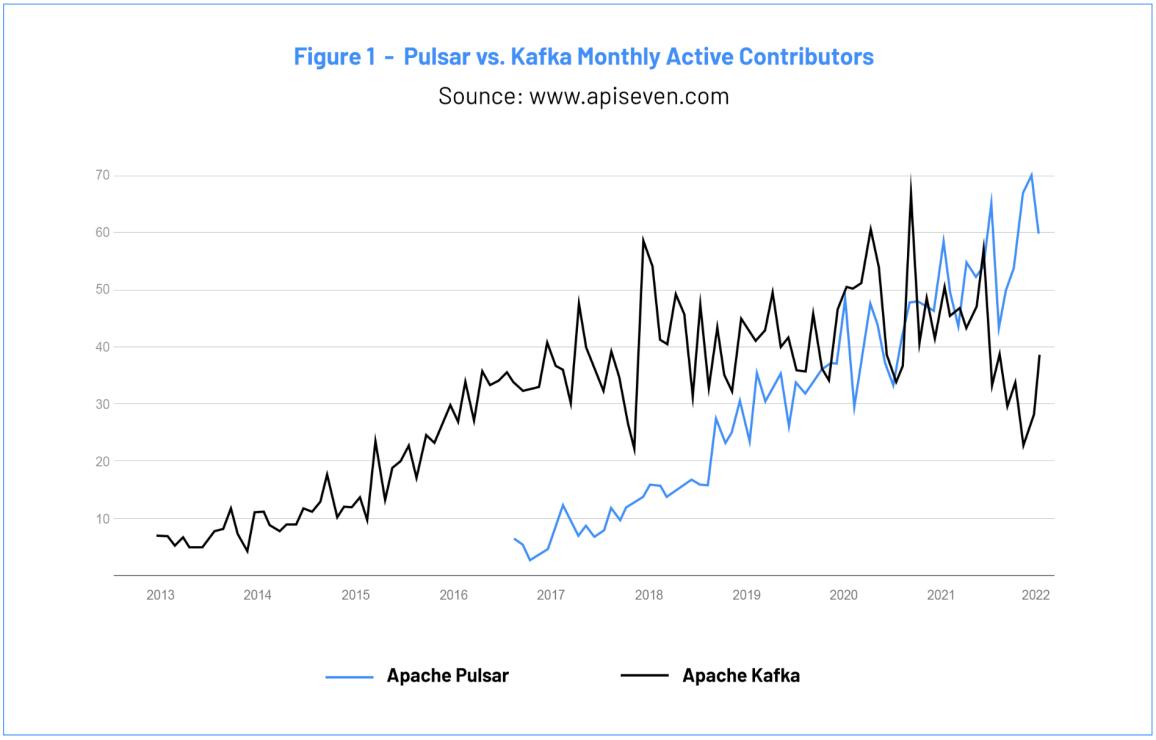


Figure 1: Pulsar vs. Kafka Monthly Active Contributors

These contributions led to major performance improvements for Pulsar. To measure the impact of the improvements, the engineering team at StreamNative, led by Matteo Merli, one of the original creators of Apache Pulsar, Apache Pulsar PMC Chairperson, performed a benchmark study using the Linux Foundation Open Messaging benchmark.

The team measured Pulsar performance in terms of throughput and latency, and then performed the same tests on Kafka. We've included the testing framework and details below and encourage anyone who is interested in validating the tests to do so.

Let's take a look at three key findings before jumping into the full results.

I. Key Benchmark Findings

2.5x
Maximum throughput compared to Kafka

Pulsar is able to achieve 2.5 times the maximum throughput compared to Kafka. This is a significant advantage for use cases that ingest and process large volumes of data, such as log analysis, cybersecurity, and sensor data collection. Higher throughput means less hardware, resulting in lower operational costs.

100x
Lower single-digit publish latency than Kafka

Pulsar provides consistent single-digit publish latency that is 100x lower than Kafka at P99.99 (ms). Low publish latency is important because it enables systems to hand off messages to a message bus quickly. Once a message is published, the data is safe and the "action" will be executed.

1.5x
Faster historical read rate than Kafka

With a historical read rate that is 1.5 times faster than Kafka, applications using Pulsar as their messaging system can catch-up after an unexpected interruption in half the time. Read throughput is critically important for use cases such as Database Migration/Replication where you are feeding data into a system of record.

Below we provide details on how the benchmark was performed and its results.

Benchmark Tests

Using the Linux Foundation Open Messaging benchmark [1], we ran the latest versions of Apache Pulsar (2.9.1) and Apache Kafka (3.0.0). To ensure an objective baseline comparison, each test in this Benchmark Report compares Kafka to Pulsar in two scenarios: Pulsar with Journaling and Pulsar without Journaling.

Pulsar's default configuration includes Journaling, which offers a higher durability guarantee than Kafka's default configuration. Pulsar without Journaling provides the same durability guarantees as the default Kafka configuration, which results in an apples-to-apples comparison.

I. What We Tested

For this benchmark, we selected a handful of tests to represent common patterns in the messaging and streaming domains and to test the limits of each system:

A. Maximum Sustainable Throughput

This test measures the maximum data throughput the system can deliver when consumers are keeping up with the incoming traffic.

We ran this test in two scenarios to test the upper boundary performance and to test the cost profile for each system:

1. **Topic with a single partition**. This scenario tests the upper boundary performance for a total-order use case or, in the worst case, where partition keys' data is skewed. At some scale, the design of a system that relies upon single ordering or handling large amounts of skewed data will need to be reconsidered. Pulsar has the ability to handle situations where total ordering is required at higher scale or large amounts of skew arise.
2. **Topic with 100 partitions**. With more partitions to stress available resources, this test illustrates how well a system scales horizontally (by adding more machines) and its cost effectiveness. For example, by modeling the hardware cost per 1GB/s of traffic, it is easy to derive the cost profile for each system.

B. Publish Latency at a Fixed Throughput

For this test, we set a fixed rate for the incoming traffic and measured the publish latency profile. Publish latency begins at the moment when a producer tries to publish a message and ends at the moment when it receives confirmation from the brokers that the message is stored and replicated.

In many real-world applications, it is required to guarantee a certain latency SLA (service-level agreement). In particular, this is true in cases where the message is published as the result of some user interaction, or when the user is waiting for the confirmation.

C. Catch-up Reads / Backlog Draining

One of the primary purposes of a messaging bus is to act as a "buffer" between different applications or systems. When the consumers are not available, or when there are not enough of them, the system accumulates the data.

In these situations, the system must be able to let the consumers drain the backlog of accumulated data and catch up with the newly produced data as fast as possible.

While this catch-up is happening, it is important that there is no impact on the performance of existing producers (in terms of throughput and latency) on the same topic or in other topics that are present in the cluster.

In all the tests, producers and consumers are always running from a dedicated pool of nodes, and all messages contain a 1KB payload. Additionally, in each test, both Pulsar and Kafka are configured to provide two guaranteed copies of each message.

Note: Pulsar also supports message queuing, complex routing, individual and negative acknowledgments, delayed message delivery, and dead-letter-queues (features not available in Kafka). This benchmark does not evaluate these features.

II. How We Set up the Tests

The benchmark uses the Linux Foundation Open Messaging Benchmark suite [1]. You can find all deployments, configurations, and workloads in the Open Messaging Benchmark Github repo [2].

The testbed for the OpenMessaging Benchmark is set up as follows:

1. 3 Broker VMs of type i3en.6xlarge, with 24-cores, 192GB of memory, 25Gbps guaranteed networking, and two NVMe SSD devices that support up to 1GB/s write throughput on each disk.
2. 4 Client (producers and consumers) VMs of type m5n.8xlarge, with 32-cores and with 25Gbps of guaranteed networking throughput and 128GB of memory to ensure the bottleneck would not be on the client-side.
3. ZooKeeper VMs of type t2.small. These are not critical because ZooKeeper is not stressed in any form during the benchmark execution.

We tested two configurations for Pulsar:

1. **Pulsar with Journaling (Default)**:
    - Uses a journal for strong durability (this exceeds the durability provided by Kafka).
    - Replicates and f-syncs data on disk before acknowledging producers.
2. **Pulsar without Journaling**:
    - Replicates data in memory on multiple nodes, before acknowledging producers, and then flushes to disk in the background.
    - Provides the same durability guarantees as Kafka.
    - Achieves higher throughput and lower latency when compared to the default Pulsar setup with journaling.
    - Provides a cost-effective alternative to the standard Pulsar setup, at the expense of strong durability. ("Strong durability" means that the data is flushed to disk before an acknowledgement is returned.)

We configured Apache Pulsar 2.9.1 to run with the 3/3/2 persistence policy, which writes entries to 3 storage nodes and waits for 2 confirmations. We are deploying 1 broker and 1 bookie for each of the 3 VMs we are using.

We used Apache Kafka 3.0.0 and the configuration recommended by Confluent in its fork of the OpenMessaging benchmark.

Details on the Kafka configurations include:

1. Uses in-memory replication (using the OS page-cache) but it's not guaranteed to be on disk when a producer is acknowledged.
2. Uses the recommended Confluent setup to increase the throughput compared to the defaults:
    - num.replica.fetchers=8
    - message.max.bytes=10485760
    - replica.fetch.max.bytes=10485760
    - num.network.threads=8
3. Uses Producers settings to ensure a minimum replication factor of 2:
    - acks=all
    - replicationFactor=3
    - min.insync.replicas=2

Note: For both Kafka and Pulsar, the clients were configured to use ZGC to get lower GC pause time.

III. Benchmark Tests & Results

A. Test #1: Maximum Throughput

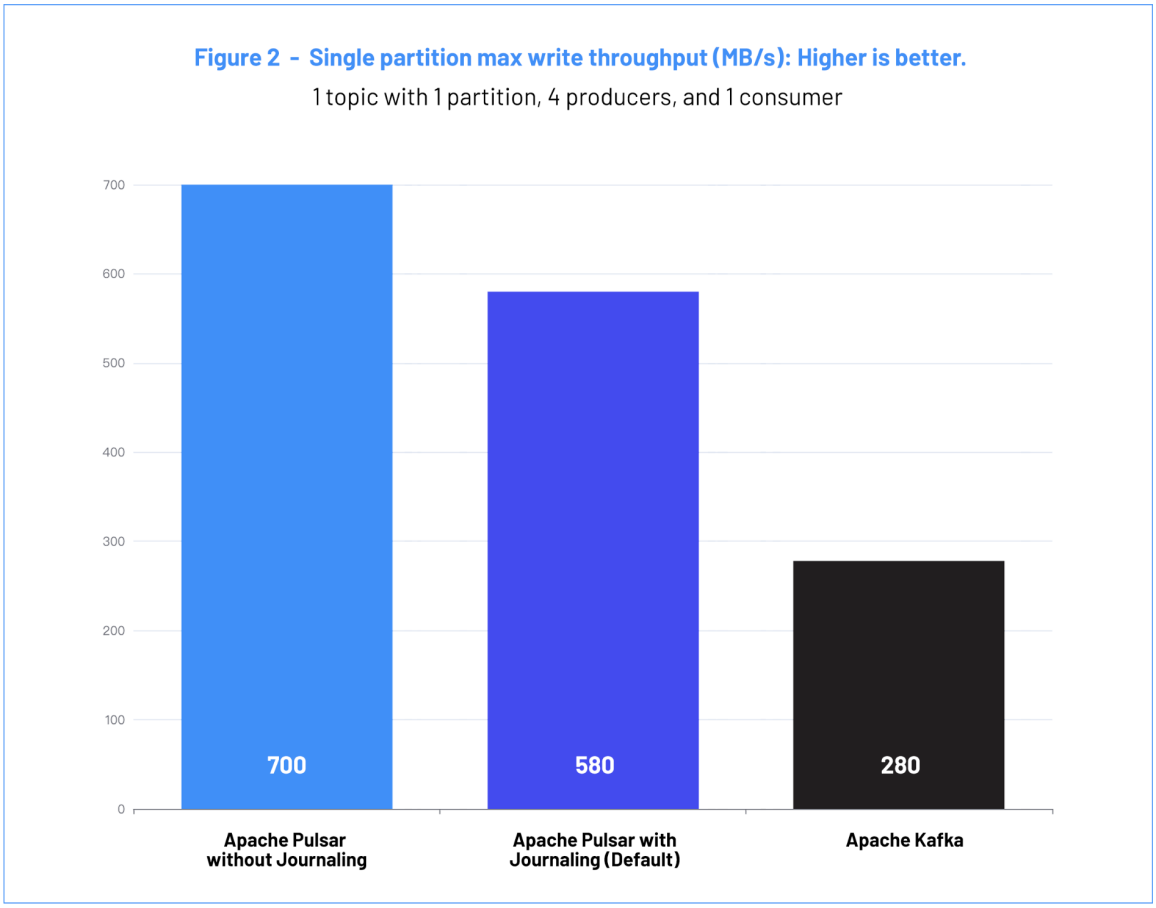
This test measures the maximum "sustainable throughput" reachable on a topic. Eg: The max throughput that is able to push from producers through consumers, without accumulating any backlog.

1. Test #1 / Case #1: Maximum Throughput with 1 Partition

This first test uses a topic with a single partition to establish the boundary for ingesting data in a totally ordered way. This is common in all the use case scenarios where a single history of all the events in a precise order is required, such as "change data capture" or event sourcing.

Driver files: pulsar.yaml, kafka-throughput.yaml
Workload file: max-rate-1-topic-1-partition-4p-1c-1kb.yaml

a. Case #1 Results: Maximum Throughput with 1 Partition



	Apache Pulsar without Journaling	Apache Pulsar with Journaling (Default)	Apache Kafka
Throughput (MB/s)	700	580	280

Figure 2: Single partition max write throughput (MB/s): Higher is better.

b. Case #1 Analysis

The difference in throughput between Pulsar and Kafka reflects how efficiently each system is able to "pipeline" data across the different components from producers to brokers, and then the data replication protocol of each system.

Pulsar achieves a throughput of 700 MB/s and 580 MB/s, respectively, on the single partitions, compared to Kafka's 280 MB/s. This is possible because the Pulsar client library combines messages into batches when sending them to the brokers. The brokers then pipeline data to the storage nodes.

In Kafka, two factors impose a bottleneck on the maximum achievable throughput: (1) the producer default limit of 5 maximum outstanding batches; and (2) the producer buffer size (batch.size=1048576) recommended by Confluent for high throughput.

Note: Increasing the batch.size setting has negative effects on the latency. This is not the case for Pulsar producers, where the batching latency is controlled by the `batchingMaxDelay()` setting, in addition to the batch max size.

With the increase in single topic throughput, Pulsar provides developers and architects more options in how they build their system. Teams can worry less about finding optimal partition keys and focus instead on mapping their data into streams.

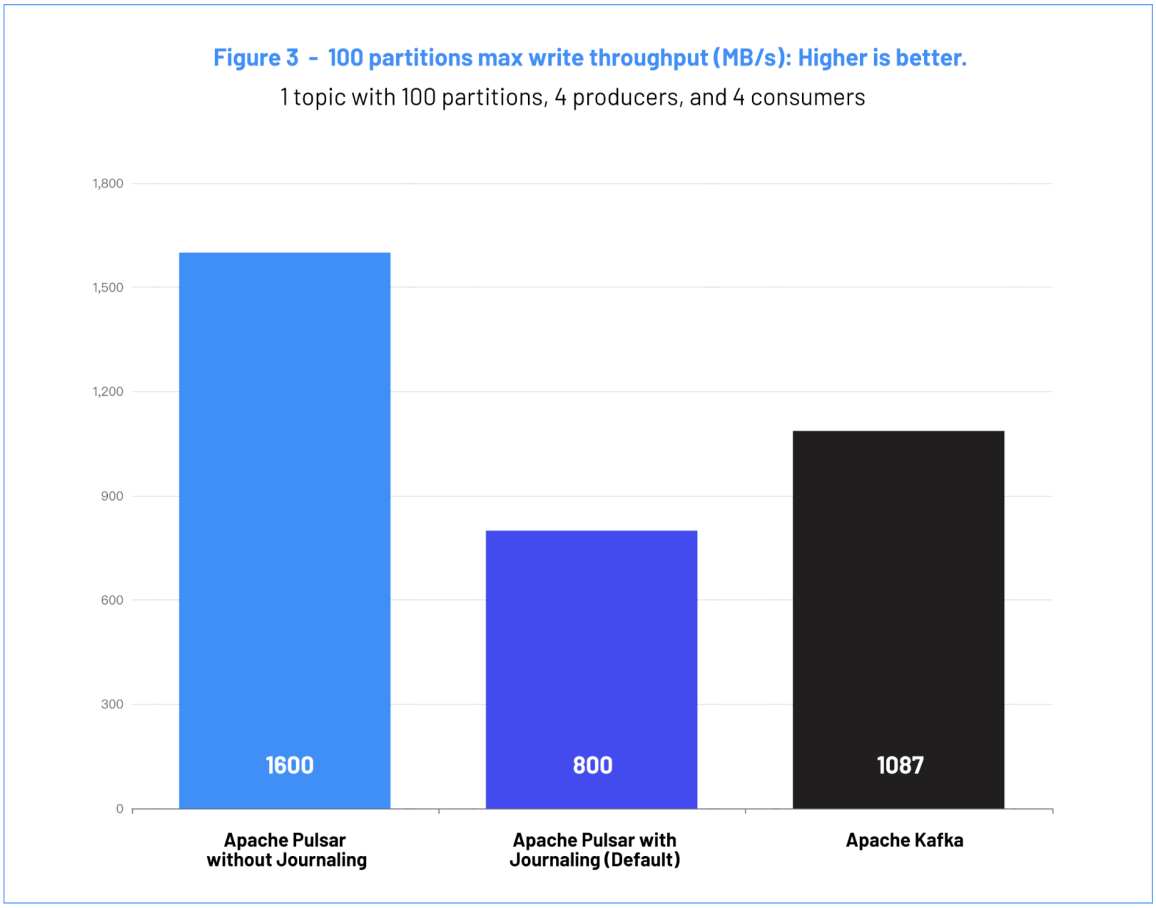
2. Test #1 / Case #2: Maximum Throughput with 100 Partitions

Most use cases that involve a significant amount of real-time data use partitioning to avoid the bottleneck of a single node. Partitioning is a way for messaging systems to divide a single topic into smaller chunks that can be assigned to different brokers.

Given that we tested on a 3-nodes cluster, we used 100 partitions to maximize the throughput of the system across the nodes. There is no advantage to using a higher number of partitions on this cluster because the partitions are handled independently and spread uniformly across the available brokers.

Driver file: pulsar.yaml, kafka-throughput.yaml
Workload file: 1-topic-100-partitions-1kb-4p-4c-2000k.yaml

a. Case #2 Results: Maximum Throughput with 100 Partitions



	Apache Pulsar without Journaling	Apache Pulsar with Journaling (Default)	Apache Kafka
Throughput (MB/s)	1600	800	1087

Figure 3: 100 partitions max write throughput (MB/s): Higher is better.

b. Case #2 Analysis

Pulsar without Journaling achieves a throughput of 1600 (MB/s), Kafka achieves a throughput of 1087 (MB/s) and Pulsar with Journaling (Default) achieves a throughput of 800 (MB/s). At equivalent durability guarantees Pulsar is able to outperform Kafka in terms of maximum write throughput. The difference in performance stems from how Kafka implements access to the disk. Kafka stores data for each partition in different directories and files, resulting in more files open for writing and scattering the IO operations across the disk. This increases the stress and contention on the OS page caching system that Kafka relies on.

When reading a file, the OS tries to cache blocks of data in the available system RAM. When the data is not available in the OS cache, the thread is blocked while the data is read from the disk and pulled in the cache.

The cost of pulling the blocked data into the cache is a significant delay (~100s of milliseconds) in serving write/read requests for other topics. This delay is observed in the benchmark results in the form of the publish latency experienced by the producers.

In the case of the default Pulsar deployment (with a journal for strong durability), the throughput is lower because 1 disk (out of 2 available in the VMs) is dedicated to the journal. Therefore we are capping the available IO bandwidth. In a production environment, this cap could be mitigated by having more disks to increase the IOPS/node capacity, but for this benchmark we used the same VM resources for each of the system configurations.

The difference in throughput can impact the cost of the solution. With parity of guarantees, this test shows that Pulsar would require 32% less hardware compared to Kafka for the same amount of traffic.

B. Test #2: Publish Latency

The purpose of this test is to measure the latency perceived by the producers at a steady state, with a fixed publish rate.

Messaging systems are often used in applications where data must efficiently and reliably be moved from a producing application to be durably stored in the messaging system. In high volume scenarios, even momentary increases in latency can result in memory resources being exhausted. In other situations, a human user may be “in-the-loop” and waiting on an operation which publishes a message - for example, a web page needs the

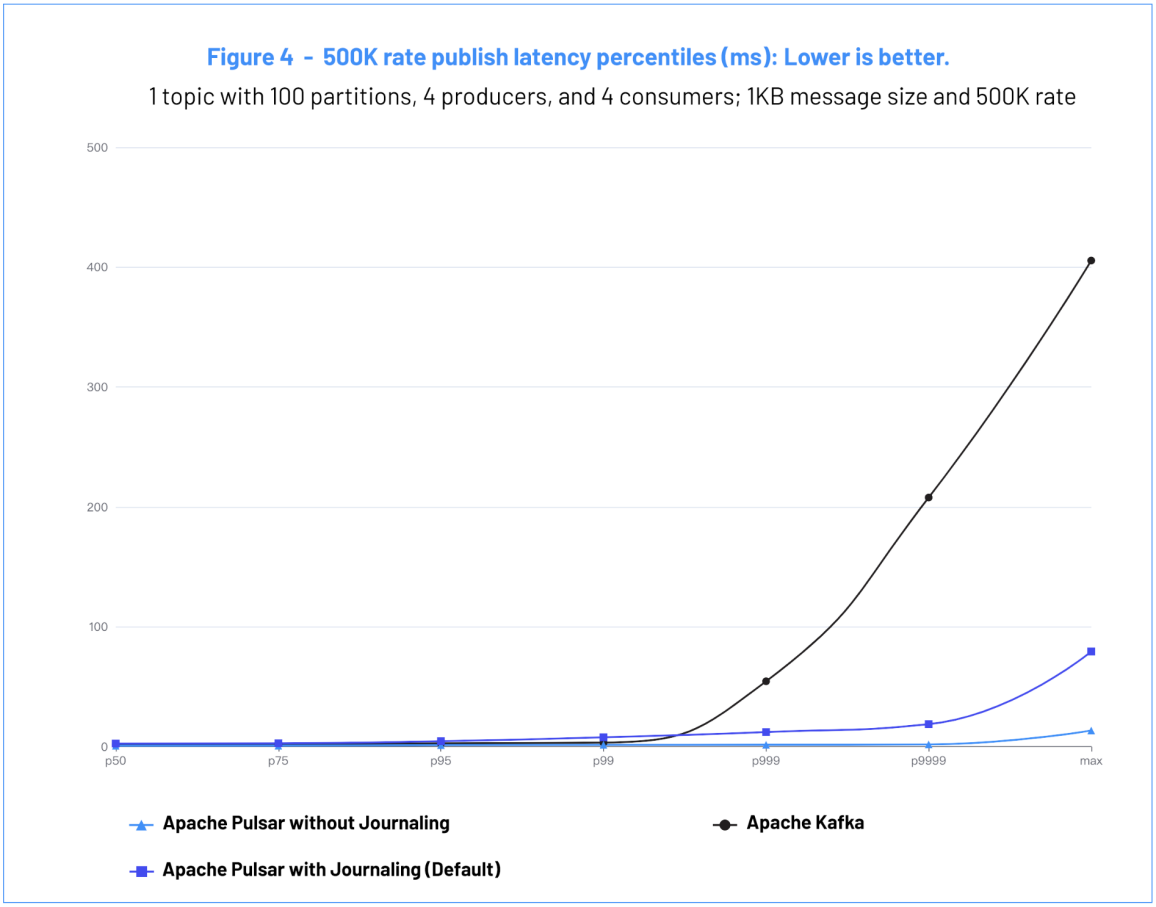
confirmation of the action before proceeding - and latency spikes can degrade the user experience. In these use cases, it is important to have a latency performance profile that is consistently within a given SLA (service-level agreement).

It is also important to consider that a high latency in the long tail (eg: 99.9 percentile and above) will still have an outsized impact over an SLA that can be offered by an application. In practical terms, a higher 99.9% latency in the producer will often result in a significantly higher 99% latency for the application request.

Because the messaging bus sits at the bottom of the stack, it needs to provide a low and consistent latency profile so that applications can provide their own latency SLAs.
This test is conducted by publishing and consuming at a fixed rate of 500 MB/s and comparing it to the publish latency seen by producers.

Driver file: pulsar.yaml, kafka-latency.yaml
Workload file: 1-topic-100-partitions-1kb-4p-4c-500k.yaml

a. Test #2 Results: Publish Latency



	Apache Pulsar without Journaling	Apache Pulsar with Journaling (Default)	Apache Kafka
P50 (ms)	0.77	2.64	1.75
P75 (ms)	0.85	2.86	2.09
P95 (ms)	1.36	4.62	2.86
P99 (ms)	1.58	7.89	3.46
P99.9 (ms)	1.68	12.24	54.56
P99.99 (ms)	1.96	18.82	207.83
Max	13.52	79.40	405.48

Figure 4: 500K rate publish latency percentiles (ms): Lower is better.

b. Test #2 Analysis

In this test, Pulsar is able to maintain a low publish latency while sustaining a high per-node utilization. Pulsar without Journaling is able to sustain 1.58 milliseconds latency at the 99 percentile and Pulsar with Journaling is able to sustain 7.89 milliseconds.

Kafka maintains a low publish latency up to the 99 percentile, where it is able to sustain 3.46 milliseconds in latency. But at 99.9%, Kafka's latency spikes to 54.56 ms.

Publishing at a fixed rate, below the max burst throughput, at 99.9% and above, Pulsar has lower latency than Kafka for both Pulsar with Journaling (default) and the Pulsar without Journaling.

The reasons for lower latency with Pulsar are:

1. When running Pulsar without Journaling, the critical data write path is decoupled from the disk access so it is not susceptible to the noise introduced by IO operations. The data is guaranteed to only be copied in memory, (unlike OS page cache which blocks under high load situations,) and then is flushed by background threads.
2. Pulsar with Journaling (Default) is able to maintain low latency because the BookKeeper replication protocol is able to ignore the slowest responding storage node. Due to the internal disk garbage collection mechanism, the performance profile of SSD and NVMe disks is characterized by good average write latency but with periodic latency spikes of up to 100 milliseconds. BookKeeper is able to smooth out the latency when used in 3/3/2 configuration, because it only waits for the two fast storage nodes for each entry.

By contrast, Kafka replication protocol is set to wait for all three of the brokers that are in the in-replica-set. Because of that, unless a broker crashes or is falling behind the leader for more than 30 seconds, each entry in Kafka needs to wait for all three brokers to have the entry.

C. Test #3: Catch-up Reads

In the consumer catch-up test, we build a backlog of data and then start the consumers. While the consumers catch-up, the writers continue publishing data at the same rate.

This is a common, real-life scenario for a messaging/streaming system. Below are a few common use cases:

1. Consumers come back online after a few hours of downtime and try to catch-up.
2. New consumers get bootstrapped and replay the data in the topic.
3. Periodic batch jobs that scan and process the historical data stored in the topic.

With this test, we can measure the following:

1. The catch-up speed.
    - Consuming applications want to be able to recover as fast as possible, draining all the pending backlog and catching up with the producers in the shortest time.
2. The ability to avoid performance degradation and isolate workloads.
    - Producing applications need to be decoupled and isolated from consuming applications and also from different, unrelated topics in the same cluster.

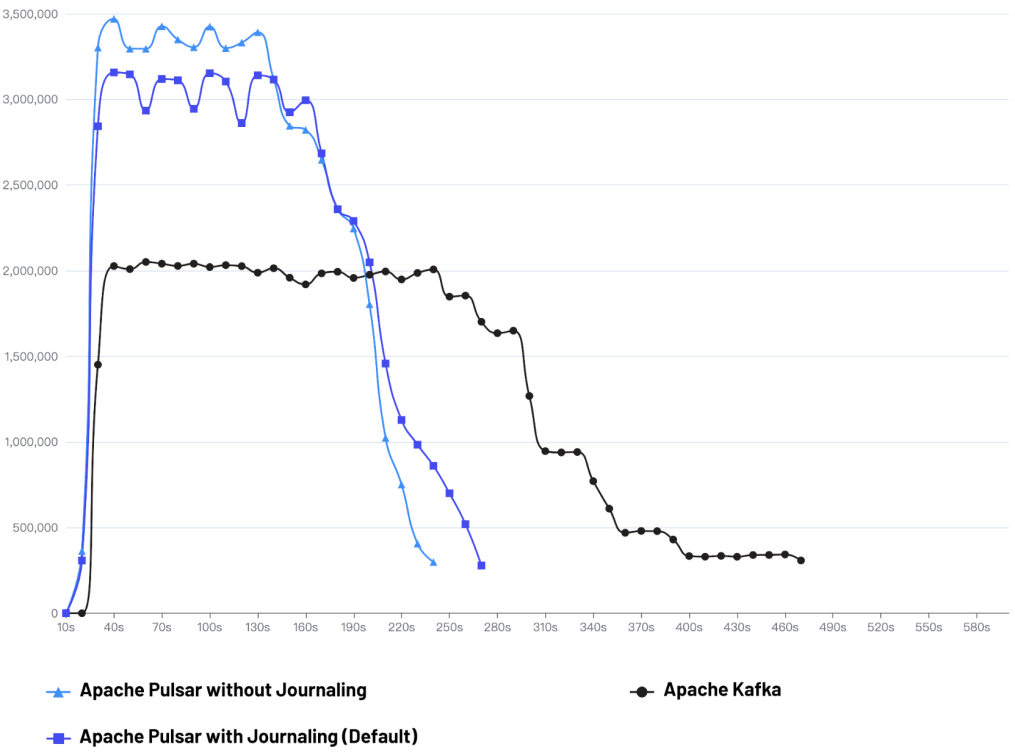
The size of the backlog is 512 GBs. It is larger than the RAM available in the nodes in order to simulate the case where the entire data does not fit in cache and the storage systems are forced to read from disk.

Driver file: pulsar.yaml, kafka-latency.yaml
Workload file: 1-topic-100-partitions-1kb-4p-4c-200k-backlog.yaml

a. Test #3 Results: Catch-up Reads

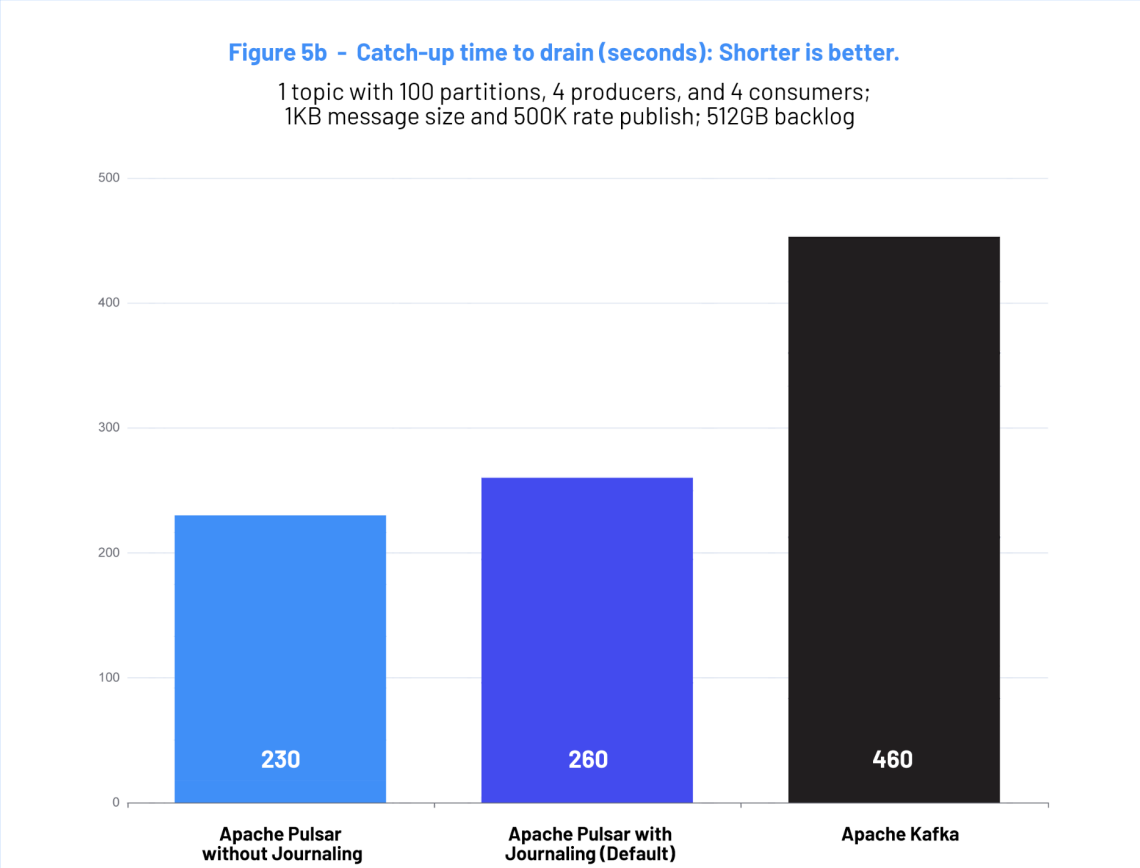
Figure 5a - Catch-up read throughput (msgs/s): Higher is better.

1 topic with 100 partitions, 4 producers, and 4 consumers; 1KB message size and 500K rate publish



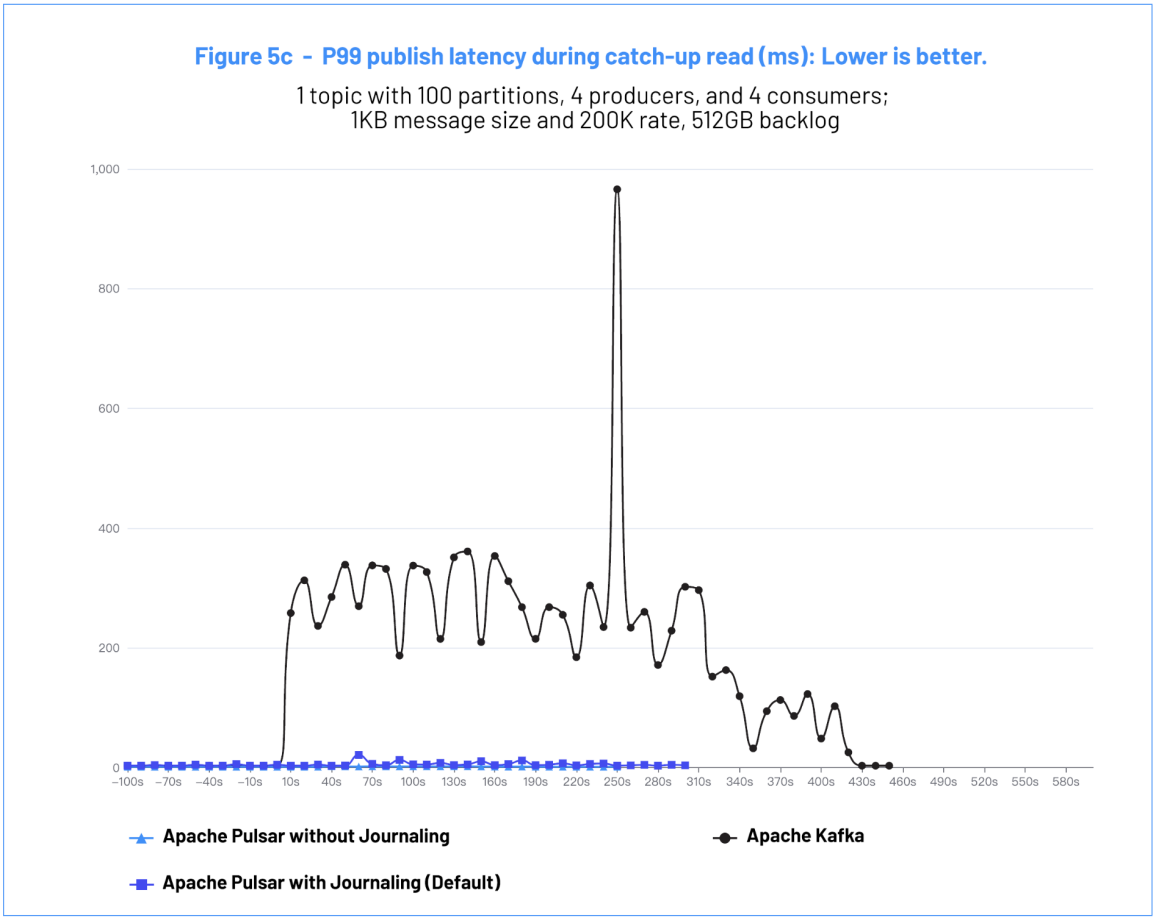
	Apache Pulsar without Journaling	Apache Pulsar with Journaling (Default)	Apache Kafka
Max Read Throughput (GB/s)	3.2GB/s	3.1GB/s	2GB/s

Figure 5a: Catch-up read throughput (msgs/s): Higher is better.



	Apache Pulsar without Journaling	Apache Pulsar Journaling (Default)	Apache Kafka
Chase time (s)	230	260	460

Figure 5b: Catch-up read chase time (seconds): Shorter is better.



	Apache Pulsar without Journaling	Apache Pulsar Journaling (Default)	Apache Kafka
P99 Publish Latency (ms)	Up to 2.5	Up to 21	Up to 380

Figure 5c: Impact publish latency during catchup read (ms): Lower is better.

b. Test #3 Analysis

The test shows that Pulsar consumers are able to drain the backlog of data ~2.5x faster than Kafka consumers, without impacting the performance of the connected producers.

With Kafka, the test showed that while the consumers are catching up, the producers are heavily impacted, with 99% latencies up to ~700 milliseconds and consequent throughput reductions.

The increase in latency is caused by the contention on the OS page cache used by Kafka. When the size of the backlog of data exceeds the RAM available in the Kafka broker, the OS will start to evict pages from the cache. This causes page cache misses that stop the Kafka threads. When there are enough producers and consumers in a broker, it becomes easy to end up in a "cache-thrashing" scenario, where time is spent paging data in from the disk and evicting it from the cache soon after.

In contrast, Pulsar with BookKeeper adopts a more sophisticated approach to write and read operations. Pulsar does not rely on the OS page cache because BookKeeper has its own set of write and read caches, for which the eviction and pre-fetching are specifically designed for streaming storage use cases.

This test demonstrates the degradation that consumers can cause in a Kafka cluster. This impacts the performance of the Kafka cluster and can lead to reliability problems.

Conclusion

The benchmark demonstrates Apache Pulsar's ability to provide high performance across a broad range of use cases. In particular, Pulsar provides better and more predictable performance, even for the use cases that are generally associated with Kafka, such as large volume streaming data over partitioned topics. Key highlights on the Pulsar versus Kafka performance comparison include:

1. Pulsar provides 99pct write latency **<1.6ms** without journal, and **<8ms** with journal for fixed 500MB/s write throughput. The latency profile does not degrade at the higher quantiles, while Kafka latency quickly spikes up to 100s of milliseconds.
2. Pulsar can prove up to **3.2 GB/s** historical data read throughput, **60% more** than Kafka which can only achieve 2.0 GB/s.
3. During historical data reading, Pulsar's I/O isolation provides a low and consistent publish latency, **2 orders of magnitude lower** than Kafka. This ensures that the real-time data stream will not be affected when reading historical data.

Pulsar: Unified Messaging & Streaming, and the Future

While Pulsar is often adopted for streaming use cases, it also provides a superset of features and is widely adopted for message queuing use cases and for use cases that require unified messaging and streaming capabilities. This benchmark did not cover the message queuing capabilities of Pulsar, but you can learn more in the *Pulsar Launches 2.8.0, Unified Messaging and Streaming* blog.

Beyond the development of Pulsar's capabilities, the Pulsar ecosystem continues to expand. Protocol handlers allow for Pulsar brokers to natively communicate via other protocols, such as Kafka and RabbitMQ, enabling teams to easily integrate existing applications with Pulsar. Integrations with Apache Pinot, Delta Lake, Apache Spark, and Apache Flink have allowed teams to make Pulsar the ideal choice to help teams use one technology across both the data and application tiers.

For more on Pulsar, check out the resources below.

Want to Learn More?

1. To learn more about Apache Pulsar use cases, check out this page.
2. Use StreamNative Free Cloud to spin up a Pulsar cluster in minutes. Get started today.
3. Sign up for the monthly StreamNative Newsletter for Apache Pulsar.

About StreamNative

Founded by the original creators of Apache Pulsar, the StreamNative team has more experience deploying and running Pulsar than any company in the world. StreamNative offers a cloud-native, scalable, resilient, and secure messaging and event streaming solution powered by Apache Pulsar. With StreamNative Cloud, you get a fully-managed Apache-Pulsar-as-a-Service offering available in our cloud or yours. Learn more at Streamnative.io.

References

[1] The Linux Foundation Open Messaging Benchmark suite:
http://openmessaging.cloud/docs/benchmarks/
[2] The Open Messaging Benchmark Github repo:
https://github.com/openmessaging/benchmark
[3] A More Accurate Perspective on Pulsar's Performance:
https://streamnative.io/blog/tech/2020-11-09-benchmark-pulsar-kafka-performance/