January 26, 2020

kafka alpakka-kafka alpakka akka-streams

How Alpakka Uses Flow Control Optimizations In Apache Kafka 2.4

Sean Glover
Principal Engineer, Lightbend, Inc.

Lightbend Contributes Performance Enhancements To Apache Kafka 2.4

Alpakka Kafka is a convenient library you can include in your application to perform stream processing from Kafka with Akka Streams. Alpakka Kafka allows the user to take advantage of a rich set of features in Akka Streams to tackle nearly any kind of stream processing business requirement. If the Akka Streams high level DSLs can't get the job done, then you can drop down to lower level DSLs, build Custom Akka Streams Graph Stages, or integrate with your own Akka Actor system.

Akka Streams' most defining feature is back-pressure[1], which is implemented according to the Reactive Streams specification. Alpakka Kafka bridges the gap between the asynchronous nature of Reactive Streams and the synchronous polling of the Kafka Consumer (Consumer), but until recently the mechanisms to perform flow control in the Consumer that are necessary to enable back-pressure created significant performance problems.

> ***Apache Kafka 2.4.0 includes a fix contributed by Lightbend that solves these performance problems (KAFKA-7548). In our benchmarks we've observed a network traffic savings between the Broker and Consumer of 32%, and an improvement in Consumer throughput of 16%!***

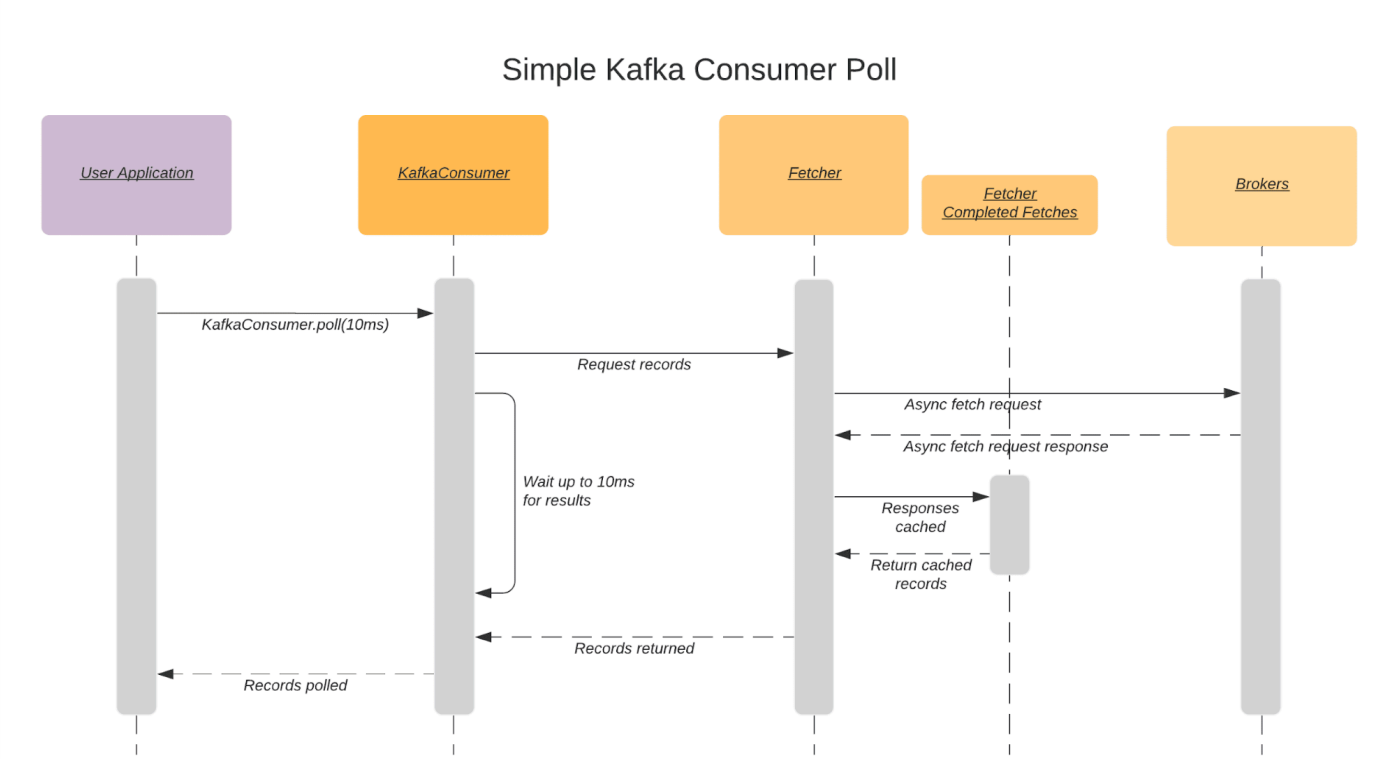
This blog post begins with details about how the Consumer and Alpakka Kafka internals are designed to facilitate asynchronous polling and back-pressure. We'll discuss how the Consumer's flow control mechanisms were improved to provide better performance. We conclude with some benchmarks that demonstrate the performance improvements before and after the new Consumer is used.

If you're not very familiar with Akka Streams or Alpakka Kafka then you may find it worthwhile to read the whole post to learn how both of these tools work and integrate together. If you're just interested in reading about the performance improvements then skip to the Flow Control Efficiency Improvements in Kafka 2.4.0 section.

Kafka Consumer Polling

The standard way to consume records from Kafka is to call the `poll` method of Consumer. Although this method is blocking (with an optional timeout), internally the Consumer requests and caches records asynchronously. Calling the `poll` method actually performs many different operations internally, but in simple terms, it will asynchronously create fetch requests to brokers to request Kafka records for assigned partitions.

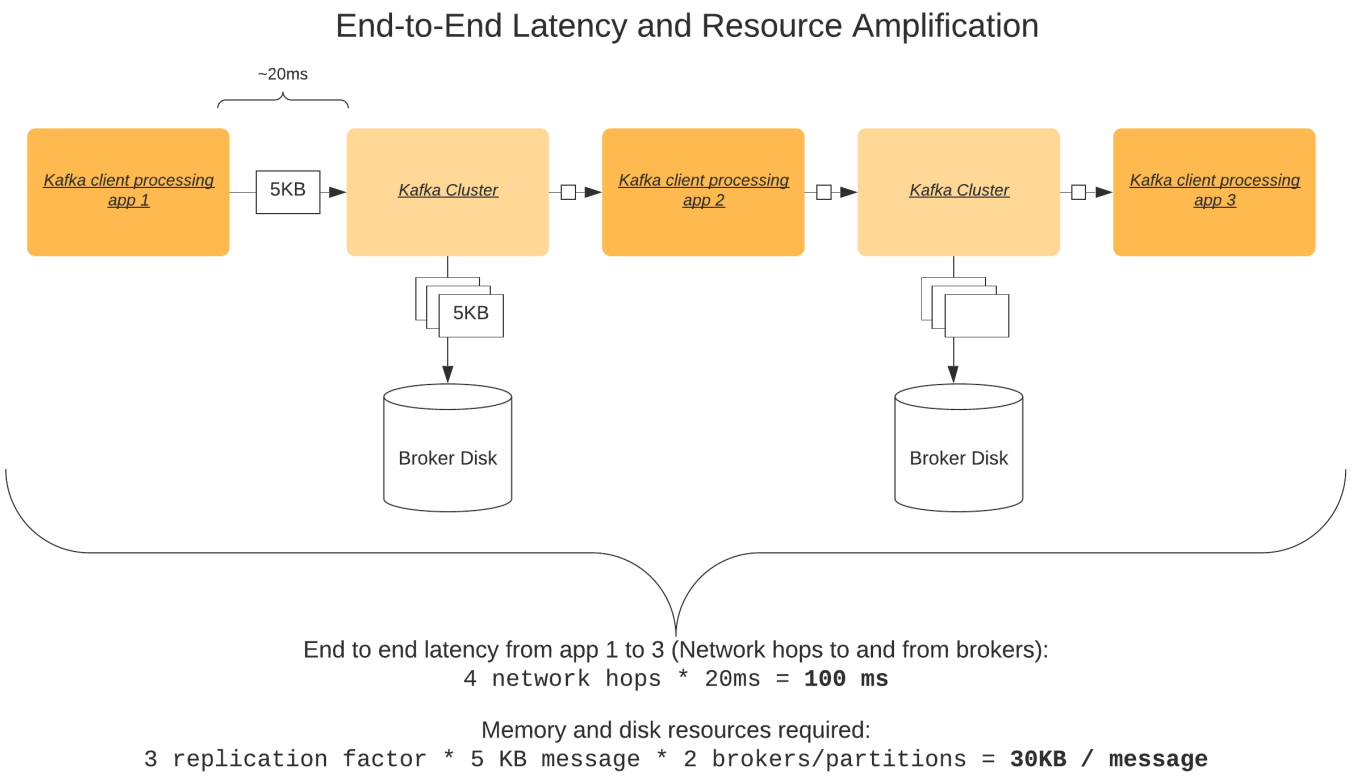
If there are records available to return (as a result of previous async fetch requests triggered by previous polls), then those records are returned immediately. If no records are available then the Consumer will wait up to the user-defined timeout for fetch request responses and then return those records. The sequence diagram below depicts an ideal scenario where the Consumer is polled and has records returned within the timeout provided (10ms).



The recommended model for flow control when using the Consumer is to poll and process in one synchronous loop so that you only ever retrieve more records when it's possible to process them. This works for simple consuming use cases that can be modeled as a single loop, but it's not sufficient when you want to distribute work across worker threads to better utilize local CPU and Memory resources.

Partitioning your topic and using consumer groups to distribute partitions across a set of Consumer is the simplest and recommended way to distribute your processing workload, but this model still requires you to poll and process messages in a synchronous manner.

If you want to subdivide that processing further then you could route messages by consuming them from a source partition, producing them to a different sink partition and then consuming them again downstream, but this model quickly bloats your end-to-end latency metrics as you consume from and produce to Kafka for each step in your data processing pipeline. It also impacts the resource requirements of your Kafka cluster as each additional message and replica takes up space and uses traffic on your network, broker disk, and broker memory. In high throughput streaming platforms this can quickly cause Kafka to become a bottleneck.



Instead of using Kafka's partitioning model every time we want to distribute work we could also asynchronously process records on separate threads (or the illusion of separate threads using Akka Actors), but to do this we must break with the synchronous polling model and use something else. Alpakka Kafka uses an asynchronous processing model by isolating the Consumer's synchronous polling and defer the responsibility of message processing to Akka Streams.

Decoupling Polling from Processing

There are times when it's ideal to asynchronously process messages from Kafka. The most common use case is when message processing takes a long time to complete and delays the next poll from occurring. Although the Consumer will asynchronously send heartbeats to the Consumer Group coordinator so that it remains alive, processing is effectively blocked because no progress is made in consuming more records. If we do not poll again within the defined `max.poll.interval.ms` then the Consumer is considered to be in a "live lock" and is removed from the consumer group.

One solution is to set a generous `max.poll.interval.ms` in the Consumer to increase the amount of time allowed between polls, or to decrease the `max.poll.records` so that less records are returned and processed in each poll. However, to get either of these solutions right requires you to have a thorough understanding of your application's specific processing semantics to get right, and still may be difficult to define for all situations. The documentation for the Consumer suggests that asynchronous polling is the best general solution to this problem:

> ***For use cases where message processing time varies unpredictably, neither of these options (`max.poll.interval.ms` and `max.poll.records`) may be sufficient. The recommended way to handle these cases is to move message processing to another thread, which allows the consumer to continue calling poll while the processor is still working.***
>
> https://kafka.apache.org/24/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html

In Complex Event Processing[2] systems it's not uncommon to have unpredictable increases in processing time if messages must be routed through complex business rules or require coordination across different systems to enrich data. By decoupling polling from processing we can process records from Kafka more swiftly and more efficiently handle other Consumer activities that occur during a poll, such as:

- Respond to consumer group rebalance events
- Asynchronously commit offsets for successfully processed records
- Process offset commit responses from commit requests

A Primer on Akka Streams and Back-pressure

A simple asynchronous processing model would have the Consumer in one thread and a pool of workers which each have their own thread. Akka Streams provides simple asynchronous APIs to scale out processing stages, which can provide us with a bounded pool of workers. Since Akka Streams is using Akka internally, we don't actually require 1 thread per worker, but for the sake of this post this detail is not important.

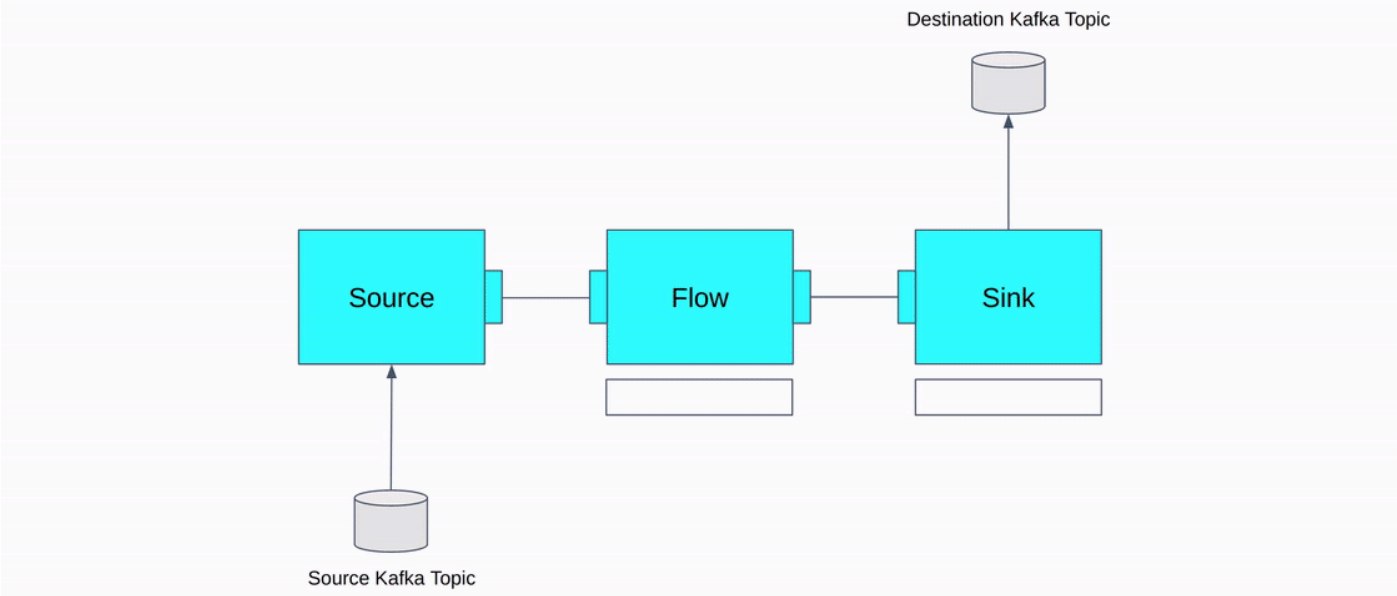
When you poll the Consumer asynchronously you must implement means to either signal the Consumer thread to only send records when they're needed (pull model), or to check if a thread is ready for records and send them (push model). Akka Streams uses a hybrid model called "dynamic push-pull" that provides the safety of a pull model with some of the performance benefits of a push model. The dynamic push-pull model is defined by the Reactive Streams specification and implemented in Akka Streams.

It works like this: when a downstream stage has room in its buffer it will create a demand request and send it to its immediate upstream stage (*pull*). The request includes the number of records to push to the downstream stage so that its buffer is full of messages to process. This allows us to only push the exact number of records a downstream stage can handle, and no more. By not sending a demand (*pull*) request for each record, we get an economy of scales by requiring less signalling overhead per record.

This property of the Reactive Streams specification is known as back-pressure[3]. Back-pressure is the key to keeping our asynchronous processing fully utilized while ensuring that it's not overloaded. Akka Streams implements all the coordination required to enable back-pressure so that the user doesn't need to be concerned with it.

The GIF below is a simple example of how back-pressure and the dynamic push-pull model works within Akka Streams. The stream contains 3 stages: a Source consuming from a Kafka source topic, a Flow to process the data, and a Sink that produces to a new Kafka destination topic.

Back-pressure Demo



1. When the stream is first run there are no messages flowing.
2. The Sink sends a demand request upstream to the Flow to signal that it has room in its mailbox. The Flow has no messages to provide, so it sends its own demand request upstream to the Source that it has room in its mailbox.
3. The Source receives the demand message from the Flow and retrieves messages (in this case it consumes messages using an Alpakka Kafka Source).
4. The Source satisfies the demand request of the Flow and sends downstream the number of messages it requested. The Flow processes messages and sends downstream the number of messages the Sink requested.
5. The Sink writes messages (in our case it produces messages using an Alpakka Kafka Sink).

How Alpakka Kafka uses Flow Control in the Kafka Consumer

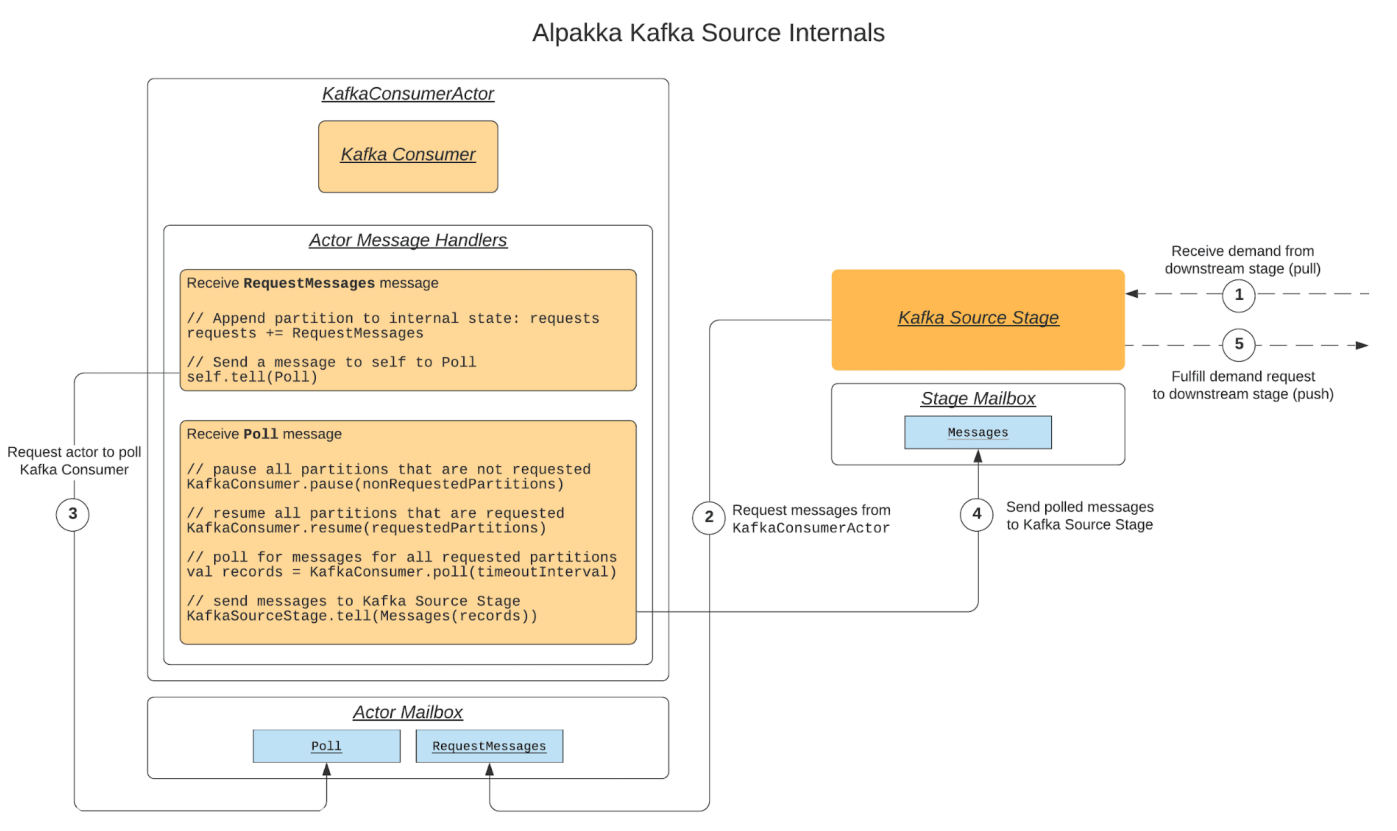
Alpakka Kafka encapsulates the Consumer in an Akka Actor called the `KafkaConsumerActor`. When an Alpakka Kafka Source stage (an Akka Streams Source) receives a demand request, it will asynchronously send a `Poll` message to the `KafkaConsumerActor`. The Alpakka Kafka Source stage will only send `Poll` messages when it receives demand requests from downstream. When the `KafkaConsumerActor` actor processes a `Poll` message it will trigger a Consumer `poll` and send any messages that were returned to the Alpakka Kafka Source stage that requested them.

At the same time, the `KafkaConsumerActor` will poll every `poll-interval` of 50 ms (by default) in order to handle other Consumer activities. If there are no requests to send messages to the Alpakka Kafka Source during a `Poll`, then we call the Consumer `pause` method for all partitions assigned to it. This instructs the Consumer to not send any fetch requests or return any records for paused partitions that may have already been fetched the next time the Consumer `poll` method is called.

You may ask why we poll if we don't intend to receive any messages. The reason is to perform other functions that occur during a poll, such as those that were discussed in Decoupling Polling from Processing. When there are requests to send messages then the Alpakka Kafka Actor will call the Consumer `resume` method for the partitions that we want to poll records for (some partitions may still remain paused if there are no requests for them, they will be re-evaluated every poll). If the poll returns messages then they are sent to the Alpakka Kafka Source stage.

By default an Alpakka Kafka Source will emit messages from all partitions assigned to it. An alternative is to use a "partitioned" Source. These sources will emit a `Source` per `TopicPartition` so that the user can process messages from Kafka per partition in its own stream. In this case there is still a single Kafka Consumer Actor (and a single Consumer), but the polled messages for each partition are routed to their own Alpakka Kafka Source.

In this situation, it's possible for one partition's stream to apply back-pressure and another one to have none. This situation drives the use case in the Kafka Consumer Actor where we only `pause` or `resume` specific partitions. Below is a diagram that shows each step of the process.

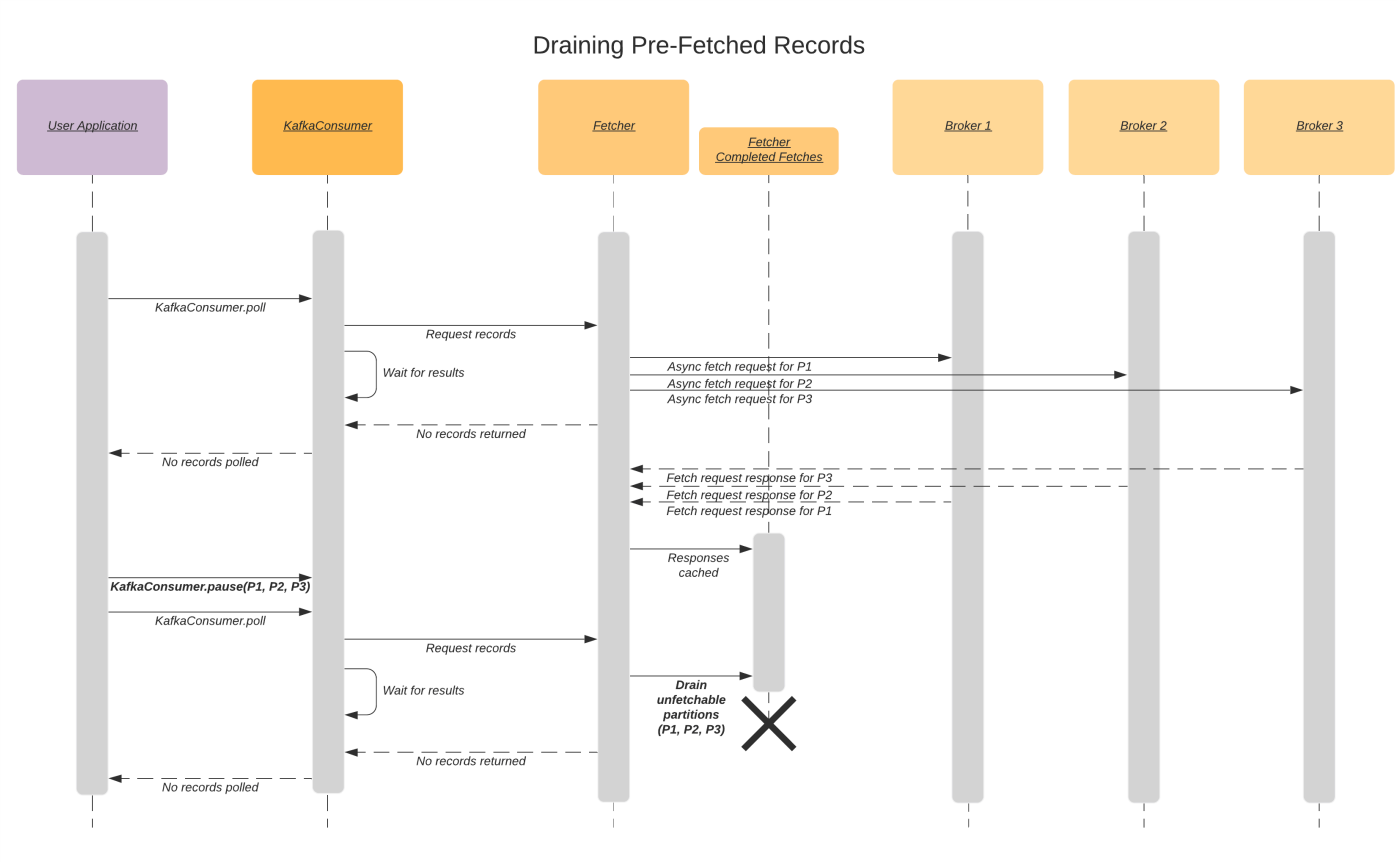


The wonderful thing about Akka Streams is that it takes care of most of the complexities involved with building asynchronous back-pressured systems. In a typical Alpakka Kafka application you have an asynchronous boundary between polling, processing, and optionally between an arbitrary set of stages within your stream[4]. Asynchronous systems can often process workloads faster and more efficiently because they can be more easily distributed across all the resources (i.e. CPU cores) of a system.

Flow Control Efficiency Improvements in Apache Kafka 2.4.0

Using `pause` and `resume` for flow control on the Consumer prior to Apache Kafka 2.4.0 could incur significant performance penalties. When a partition was paused the Consumer would treat it as any other “unfetchable” partition. Partitions can be unfetchable for a few reasons: the partition may no longer be assigned to the Consumer or is in the process of being reassigned; the current offset position is not valid according to the group coordinator; or the partition is paused. In any of these cases we do not want to fetch more records, but the major consequence is that any pre-fetched data in the Consumer which might already be ready to be polled will be “drained” (or in other words, deleted).

When pausing partitions for flow control the reasonable assumption is that it will eventually be resumed. If the pre-fetched data is deleted before the user resumes the partition, then that data must be re-fetched. The sequence diagram below illustrates how this problem can occur within the Consumer.



The implication of deleting pre-fetched data is obvious - it incurs significant performance penalties during normal streaming operations across the board. Some of the biggest penalties include:

- Traffic between the consumer and broker increases because records may be fetched more than once.
- Consumer throughput decreases.
- Fetch requests to the broker increase.
- Heap and off-heap memory allocation churn increases.

This issue is more pronounced the more often you pause and resume partitions, which is quite common in Alpakka Kafka when the downstream is in a back-pressuring state. Back-pressure will occur whenever you have a typical “fast producer slow consumer” situation. Other stream processors like Apache Samza and Kafka Streams also make use of pause and resume flow control, but to a lesser degree. Several Kafka Connect connectors will pause and resume as well.

The solution to this problem is to treat paused partitions differently from other unfetchable partitions. Instead of draining data that was fetched before the partition was paused, keep it in the completed fetches cache until the partition is resumed and returned to the end-user during either a poll, Consumer shutdown, or partition data invalidation.

This problem was originally reported by Mayuresh Gharat, a Kafka Developer at LinkedIn, in Apache Kafka Jira issue KAFKA-7548. LinkedIn was aware of the pre-fetch draining issue and decided to measure the performance implication of using `pause` and `resume` in stream processors like Apache Samza. They created a test which would randomly pause 9 out of 10 partitions before

each poll and observed a significant difference in consumer throughput before and after an interim fix was applied. The exact details of the implementation are not available, but the summary Mayuresh provided in KAFKA-7548 was compelling.

> ***We had a consumer subscribed to 10 partitions of a high volume topic and paused predefined number partitions for every poll call. The partitions to pause were chosen randomly for each poll() call.***
>
> - Time to run Benchmark = 60 seconds.
> - MaxPollRecords = 1
> - Number of TopicPartition subscribed = 10
>
> | Number Of Partitions Paused | Number of Records consumed (Before fix) | Number of Records consumed (After fix) |
> |---|---|---|
> | 9 | 2087 | 4884693 |
>
> https://issues.apache.org/jira/browse/KAFKA-7548

The Alpakka team was made aware of this issue after an issue with similar characteristics was submitted to the alpakka-kafka GitHub issue tracker. Adam Kafka, a Software Engineer at Tesla, reported #543 "Consumer fetches duplicate records when lagging, resulting in high network use, but low throughput". Several theories and workarounds were discussed before eventually the connection was made to the issue that Mayuresh had originally reported. Once we were aware of KAFKA-7548 we kept an eye on it, anticipating a fix would be available soon, but when progress stalled on remediation work in the PR #5844 we decided to lend a hand. We picked up where Mayuresh had left off and followed up with new PRs #6988 and #7228, which were eventually merged and released with Apache Kafka 2.4.0 (a big thank you to Jason Gustafson of Confluent who helped get us across the finish line).

Benchmarks

Once Apache Kafka 2.4.0 was released we were able to perform before and after benchmarks[5] to demonstrate the performance improvement.

Benchmarks were run with Alpakka Kafka 2.0.0-RC1 referencing Apache Kafka 2.3.1 (the latest release before the fix) and Apache Kafka 2.4.0 (the first release with the fix). Brokers were run as Docker containers using Confluent Platform 5.3.1 images, and the testcontainers-java project was used to orchestrate them during the test.

Metrics were collected every 100 ms until all records were consumed. Metrics were polled from the Consumer, broker (for the test topic), and Consumer's JVM. The test had the following configuration:

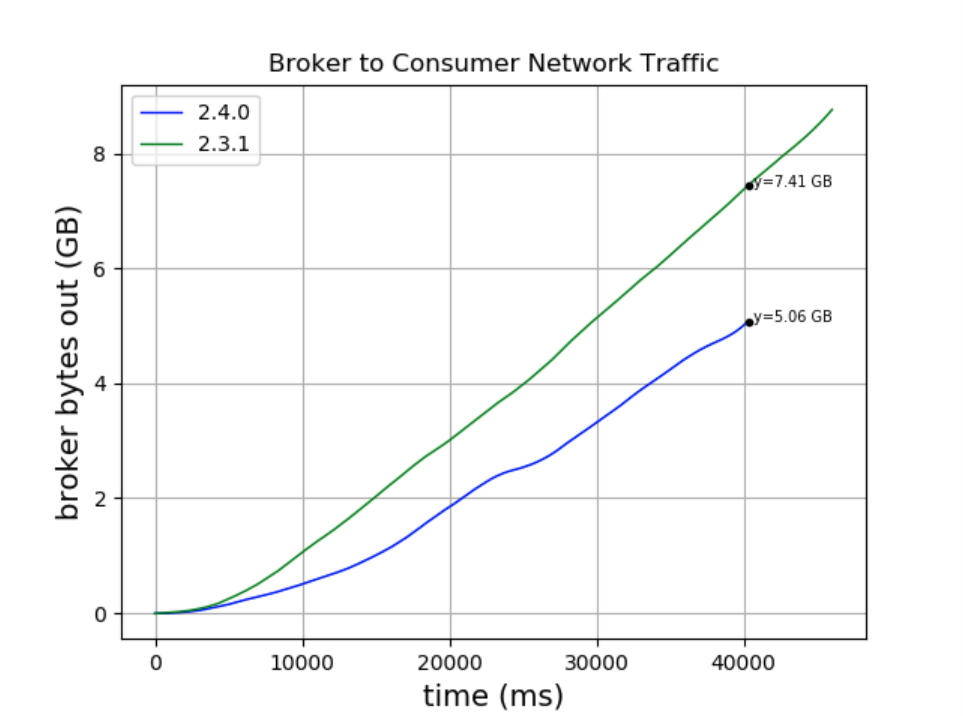
- Number of records: 1,000,000
- Test Topic Partitions: 100
- Test Topic Replication Factor: 3
- Message Size: 5KB
- Number of Brokers: 3
- All other Alpakka Kafka and Consumer configuration used defaults

Keep in mind that these benchmarks are only used to determine a rough estimate of relative performance difference between two versions of the Consumer. Actual performance savings could be more or less than the ratios presented in this post, but they should always be better than when using previous versions of Alpakka Kafka.

Broker to Consumer Network Traffic

The most significant savings is in the amount of network traffic observed between the Consumer and Kafka brokers. The amount of data sent from the broker to the consumer is significantly less when less pre-fetched partition data is thrown away.

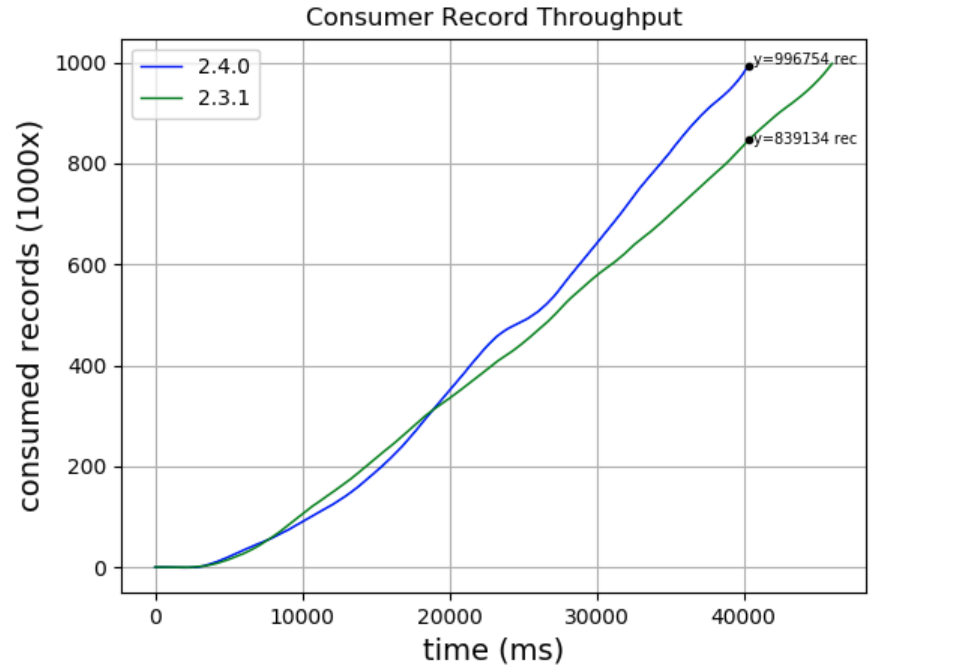
Estimated improvement: **32%**



Consumer Record Throughput

The record throughput increases because the consumer is able to poll cached records from previously paused partitions without waiting for another fetch request to retrieve the data again.

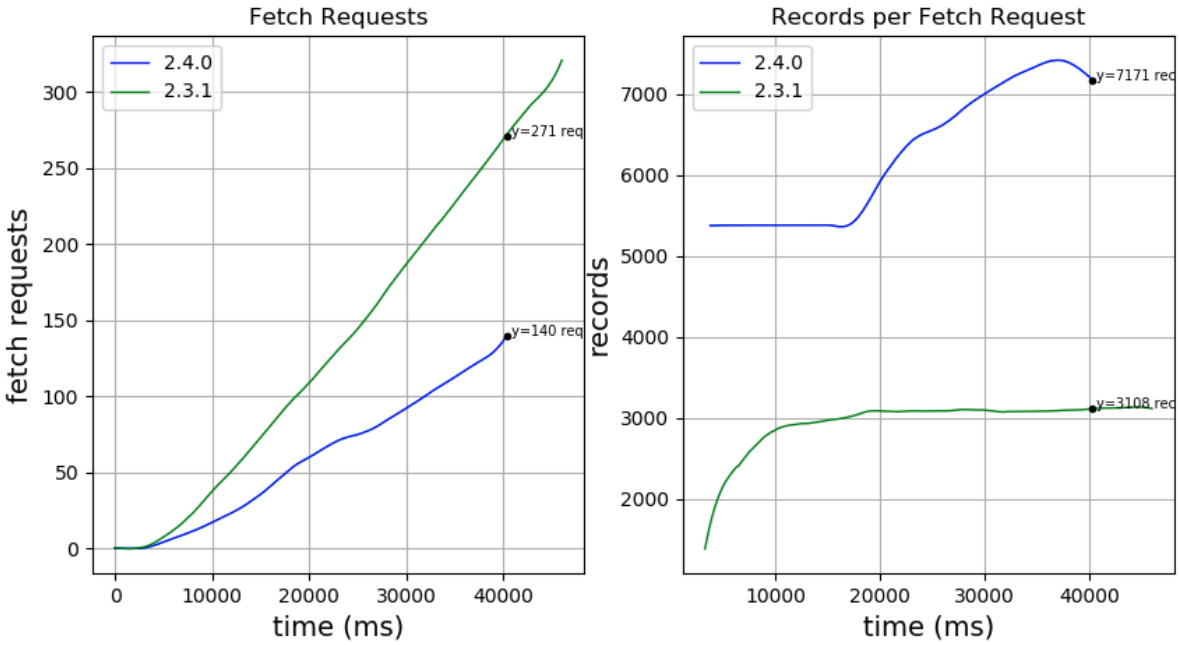
Estimated improvement: **16%**



Fewer Consumer to Broker Fetch Requests

The Consumer no longer has to re-fetch pre-fetched data that was previously drained. The first chart shows how many cumulative fetch requests are sent to Kafka brokers. The second chart indicates how many records were returned per fetch request that were actually processed. The efficiency indicated by this ratio correlates to the decrease in fetch requests issued by the Consumer.

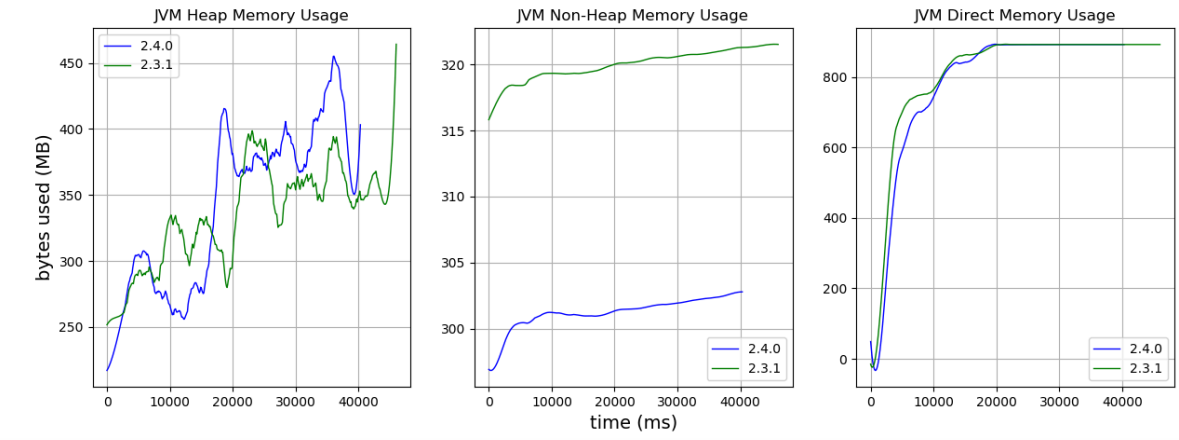
Fetch requests estimated improvement: **48%**
Records per fetch request estimated improvement: **57%**



Heap, Off-Heap, and Direct Memory Usage Churn is Reduced

The test results comparing JVM memory usage were inconclusive. JVM Heap Memory usage is highly variable, depending a lot on a test run's particular runtime characteristics and luck-factor with GC timing. The JVM Non-Heap Memory shows a small and consistent difference, but this could be explained by the differences of what's stored in resident memory when referencing either Kafka 2.3.1 or 2.4.0. For example, each version has different class metadata, or compiled code artifacts (loaded dependencies) of different sizes. Direct memory usage was nearly identical.

With the fix the JVM no longer has to deallocate and reallocate memory as often for pre-fetched partition data. GC metrics (not shown) did not indicate a significant difference in cumulative GC pause time or iterations. Some users of Alpakka Kafka have anecdotally noticed less memory-pressure (specifically direct memory pressure in constrained memory environments such as cgroups run by Kubernetes pods), but I was unable to reproduce these results.



Conclusion

The asynchronous stream processing model can offer a lot of advantages when building Complex Event Processing streaming platforms. Alpakka Kafka, Akka Streams, and the Akka toolkit in general provide a complete set of tools that can be used to build a streaming platform for nearly any set of requirements.

The optimized Consumer flow control implemented by Lightbend in KAFKA-7548 is available in Alpakka Kafka 2.0.0, released on January 15, 2020. Although this fix is only applicable to Alpakka Kafka Sources, there have been other performance improvements in Alpakka Kafka Flows and Sinks by combining producing and offset committing into a single Akka Streams stage.

This optimization allowed for more performant streams with at-least-once message delivery. These combined improvements provide a large performance savings in Alpakka Kafka stream processing across the board. If you are using an earlier version of Alpakka Kafka then you should upgrade today to take advantage! For a summary of other changes introduced in Alpakka Kafka 2.0.0 see our release post on the Akka blog.

Need Help With Kafka And Akka Streams?

If you need help building backpressure-enabled Alpakka Kafka and Akka Streams applications, consider Lightbend's newest product, Cloudflow. Cloudflow takes care of most of your development and infrastructure data engineering needs when building streaming data platforms. Cloudflow uses Alpakka Kafka to let users build Akka Streams runners, and provides deep metrics and observability for Lightbend subscribers using Lightbend Console and Telemetry. Request a demo to see Cloudflow in action!

GET A CLOUDFLOW DEMO

[1 & 3] Back-pressure educational resources. An excellent introduction to Akka Streams, the Reactive Streams specification, and back-pressure can be found in this Lightbend presentation by Konrad Malawski "Understanding Akka Streams, Back Pressure and Asynchronous Architecture". The Akka Streams documentation also has a detailed section called "Back pressure Explained".?1 ?3

[2] Complex Event Processing - As defined by the Databricks glossary: "Complex event processing (CEP) also known as event, stream or event stream processing is the use of technology for querying data before storing it within a database or, in some cases, without it ever being stored. Complex event processing is an organizational tool that helps to aggregate a lot of different information and that identifies and analyzes cause-and-effect relationships among events in real time. CEP matches continuously incoming events against a pattern and provides insight into what is happening. and allows you to proactively take effective actions." https://databricks.com/glossary/complex-event-processing. ?

[4] Asynchronous boundaries - For a good primer on how async boundaries can be used within an Akka Streams graph see the Asynchronous Boundaries section of Colin Breck's blog post "Maximizing Throughput for Akka Streams". ?

[5] Benchmark test: `"bench with normal messages and one hundred partitions with inflight metrics"`
Data analysis, charts, and diagrams used for this blog post: seglo/alpakka-kafka-flow-control-blog-post. ?

Author

Sean Glover
Principal Engineer, Lightbend, Inc.

Twitter: @seg1o GitHub: seglo

Sean is a Principal Engineer on the Akka team at Lightbend where he maintains the open source Alpakka and Alpakka Kafka projects. Sean enjoys building data streaming platforms, reactive distributed systems, and working within the open source community.