Apache Kafka for Beam Python SDK

Chamikara Jayalath
April 2018

Apache Kafka is one of the heavily used streaming software platforms available today. Apache Beam currently has a Kafka connector (a source and a sink) for Java SDK. Current Kafka source for Java SDK is developed using the Java *UnboundedSource* API while Kafka sink is developed primarily using Beam *ParDo* transforms.

Until recently, Beam Python SDK did not offer an API for defining unbounded sources. But we recently added support for Splittable *DoFn* API which serves this purpose. Additionally, we added support for executing Splittable *DoFn*s using *DirectRunner*. Support for other runners over Fn API is currently under active development.

Many users of Beam Python SDK have requested support for Kafka. Given that we have necessary API support for adding a new unbounded source and a sink I believe this will be a useful addition to Python SDK. In addition to this, I believe that a new Splittable *DoFn* based Kafka source will serve as a good example for future developers who wish to add new unbounded sources based on the same API.

Cross-language transforms

There are currently discussions on supporting cross-language transforms in Beam using the portability framework which will in the future allow, among other things, the ability to use existing Java IO transforms in Python SDK. The exact scope and limitations of this feature are yet to be determined. Even with this planned feature, I believe Beam community will significantly benefit from having a Python SDK version of Kafka connector. I have listed some of the benefits below.

- Users might find it useful to have at least one unbounded source and sink combination implemented in Python SDK and Kafka is the streaming system that makes most sense to support if we just want to add support for only one such system in Python SDK.
- Not all runner deployments might support cross-language IO. Also some user/runner/deployment combinations might require an unbounded source/sink implemented in Python SDK. An example might be a constrained deployment where a single Java runner harness instance connects to a cluster of Python SDK harness instances.
- We recently added Splittable *DoFn* support to Python SDK. It will be good to have at least one production quality Splittable *DoFn* that will server as a good example for any users who wish to implement new Splittable *DoFn* implementations on top of Beam Python SDK.

- Cross-language transform feature is currently is in the initial discussion phase and it could be some time before we can offer existing Java implementation of Kafka for Python SDK users.
- Cross-language IO might take even longer to reach the point where it's fully equivalent in expressive power to a transform written in the host language - e.g. supporting host-language lambdas as part of the transform configuration is likely to take a lot longer than "first-order" cross-language IO. KafkaIO in Java uses lambdas as part of transform configuration, e.g. timestamp functions.

Another alternative is to represent the transform in the form of a well-known URN and a payload and implement/override the transform natively in each runner and not maintain a Beam implementation at all. But having a Beam implementations of an IO connector such as Kafka has many benefits. For example,

- IO connector will offer same behavior and feature set across various runners/SDKs.
- Beam community will be able to view/modify/improve the IO connector.
- existing IO connectors will serve as examples for users who wish to develop new IO connectors.
- More runners will be able to execute the users pipelines that use the IO connector.

Features of Java Kafka Connector

Existing Java connector supports various advanced features on top of basic support for reading from and writing to Kafka. Below I have given the features of Java Kafka source and sink. We will be implementing a subset of these features in the first version of the Python Kafka connector. Eventually we can add additional features based on the user demand.

Advanced Features of Java Kafka source

- Multiple ways to handle timestamp (log append time, processing time, create time)
- Support for reporting backlog
- Reading batches of messages in the background through extra threads
- Caching the reader object
- Exactly once delivery
- Support for multiple versions of Kafka through an additional abstraction layer
- Committing finalized offsets to Kafka to improve performance when restarting jobs.

Advanced Features of Java Kafka sink

- Exactly once semantics when writing.
- Support for custom timestamp functions

Client libraries

Currently there are three main Kafka client libraries available for Python.

- kafka-python [1]
- pykafka [2]
- confluent-kafka [3]

Below I compare various aspects of these client libraries.

Ease of deployment

kafka-python is a pure Python implementation. Confluent kafka is backed by a C library librdkafka [4]. pykafka has a pure Python implementation but can also be customized to utilize the librdkafka library. Pure Python implementations will be easier to use when it comes to deployment. C libraries will require a more advanced deployment setup and not all runners may support that. pykafka has the additional advantage of supporting both pure-Python and C library-backed versions without having to significantly to update the user of the library.

Performance

Not surprisingly, C based libraries (confluent-kafka and pykafka+librdkafka) offer the highest performance for both message publication and consumption. But pure Python libraries offer very acceptable performance which should be sufficient for most users. I refer to a previous article [5] that compares performance of various Kafka client libraries for performance of pykafka backed by librdkafka and for performance of confluent-kafka. I ran additional experiments to compare the performance of two pure Python client libraries.

All experiments were run on a MacBook pro with a 2.2 GHz Intel Core I7 processor and 16GB of memory. Kafka deployment was a single node Kafka broker and a ZooKeeper deployment in the same machine. Each data point is the average of five runs where each run produced or consumed 1000000 messages. I used latest available versions of kafka-python (1.4.2) and pykafka (2.7.0). Kafka deployment was a single node broker of version 0.9.0.0.

Producer performance

Message size = 10 bytes

Message size (Bytes)	# of messages/sec	MBs/sec
kafka-python (msgs/sec)	11064	0.11

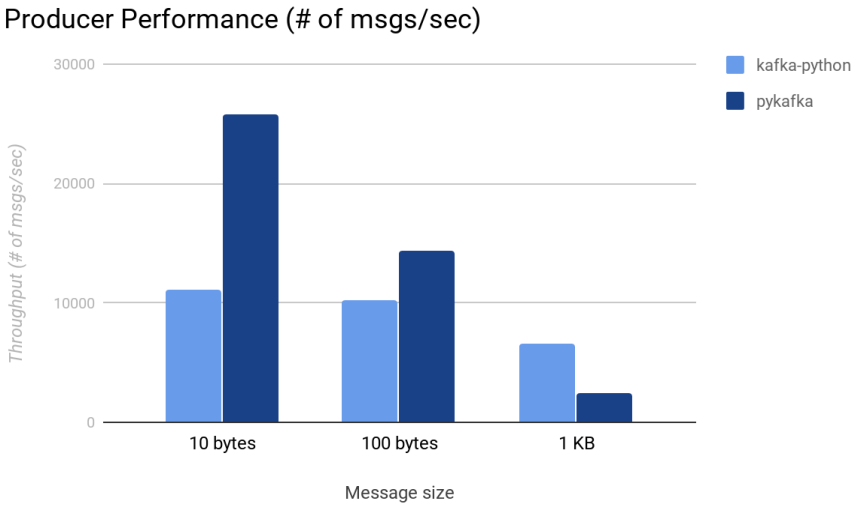
pykafka (msgs/sec)	25760	0.25

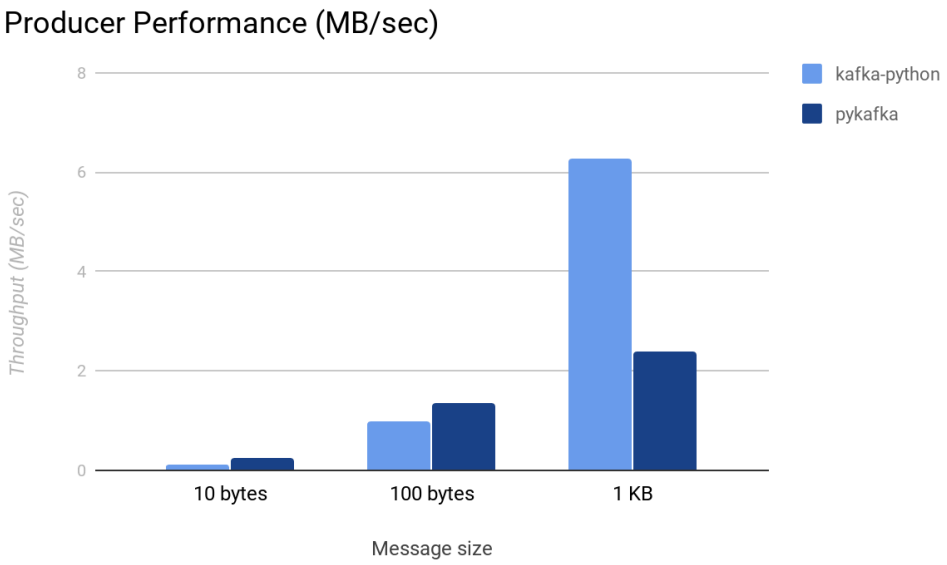
Message size = 100 bytes

Message size (Bytes)	# of messages/sec	MBs/sec
kafka-python (msgs/sec)	10202	0.97
pykafka (msgs/sec)	14340	1.37

Message size = 1KB

Message size (Bytes)	# of messages/sec	MBs/sec
kafka-python (msgs/sec)	6539	6.39
pykafka (msgs/sec)	2459	2.4





Consumer performance

Message size = 10 bytes

Message size (Bytes)	# of messages/sec	MBs/sec
kafka-python (msgs/sec)	42390	0.40
pykafka (msgs/sec)	19893	0.19

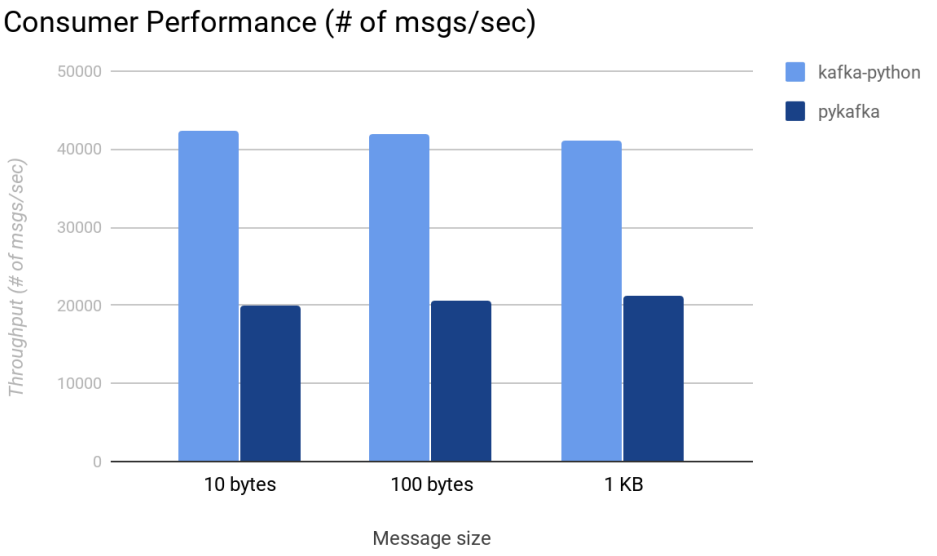
Message size = 100 bytes

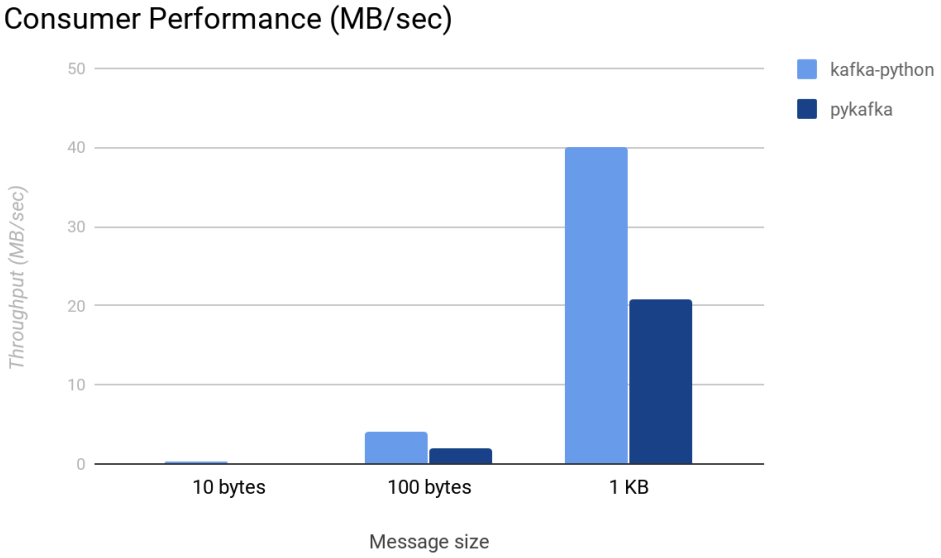
Message size (Bytes)	# of messages/sec	MBs/sec
kafka-python (msgs/sec)	41981	4
pykafka (msgs/sec)	20637	1.97

Message size = 1KB

Message size (Bytes)	# of messages/sec	MBs/sec

kafka-python (msgs/sec)	41071	40.11
pykafka (msgs/sec)	21208	20.71





In general, kafka-python exhibited better performance compared to pure-Python pykafka. One outlier was publishing really small (100 bytes or less) messages where pykafka performed better. kafka-python consumer was approximately 2 times faster compared to pykafka. So if we compare pure-Python implementations kafka-python seems to be better. If we use C-based deployment though, pykafka seems to be performing much better. See [5] for benchmarks

related to this. It is also to be noted that the performance exhibited by pure-Python Kafka libraries may be enough for most use-cases.

Maintainability

All three libraries are well maintained and have had active commits from many contributors in the recent past. confluent-kafka and pykafka are backed by companies Confluent and parse.ly respectively.

API and Feature set

Confluent Kafka is more recent and tries to maintain an API similar to the Java client library from the same company. I read many comments about this API being hard to use for Python developers and not being well documented. kafka-python also offers a Java-centric API but is better documented and easier to use. pykafka offers the most pythonic API and is well documented.

All three libraries offer core support for reading from and writing to a Kafka deployment. For developing an efficient Kafka source, we need to ability to read a Kafka partition from a given offset. All three client libraries support this.

Library of choice

I propose choosing kafka-python for implementing Kafka connector for Python SDK.

We previously discussed choosing between pure-Python and C based client libraries for Python IO [6] and decided to choose a pure Python version due to ease of deployment. I think this argument applies to Kafka client libraries as well. Even though C based libraries significantly outperforms pure-Python implementations, deployment of C based libraries might be complicated for certain runners and deployments. Moreover I believe that performance exhibited by pure-Python libraries will enough for most of the Python Kafka users.

Out of the two pure-Python libraries kafka-python is about twice as performant across all message sizes considered when it comes to message consumption and kafka-python outperformed pykafka for message publication for all but extremely small message sizes.

Poposed API

I created a PoC [7] that illustrates the proposed API of the connector. I included a set of features that I believe will be useful to include in the first version of the connector. Based on user feedback we can add to this feature-set in the future development iterations.

[1] https://pypi.org/project/kafka-python/

[2] https://pypi.org/project/pykafka/
[3] https://pypi.org/project/confluent-kafka/
[4] https://github.com/edenhill/librdkafka
[5] http://activisiongamescience.github.io/2016/06/15/Kafka-Client-Benchmarking/
[6] https://docs.google.com/document/d/1-uzKf4VPlGrkBMXM00sxxf3K01Ss3ZzXeju0w5L0LY0/edit?usp=sharing
[7] https://github.com/chamikaramj/beam/commit/2a126d26ddf637fe0f8d270877b3aec64e920192