Message Metadata (aka Headers) in Kafka

TL;DR

Add (String, byte[]) pairs to the ProducerRecord and ConsumerRecord. Maybe put this into the underlying message format. Use ints as keys only if you need ultimate performance, but probably just don't use headers in that case.

Author: Sean McCauliff <smccauliff@linkedin.com>
Last Modification: 23 January 2016

Introduction

Terminology

Kafka clients (producers and consumers) have an abstraction of the data that are written or read from Kafka. These are known as ProducerRecord or ConsumerRecord. When speaking about user data we use the term "record" to mean both the ProducerRecord and the ConsumerRecord.

What are headers

Many different data storage and transport systems provide a way to augment to the data being stored or sent in order to provide a common system of augmentations that allows for suppliers of data to interact with the users of that data without changing the semantics of the data being stored or transported. In a storage context this additional data is often known as metadata in the networking context this additional data is often known as headers (although it may physically be implemented as footer, that is the headers bytes are physically located at the tail end of the data). Additionally when we talk about headers we mean use extensible headers as opposed to system headers, although there is no technical reason the system itself can not use them.

Examples of other systems implementing headers:

- Other message queuing APIs and protocols such as JMS and AMQP.
- Protocols such as HTTP and SMTP.
- Filesystem extended attributes.
- Media files such as MP3 and JPEG.
- Scientific file formats such as FITS.

We assert that headers will also be useful for Kafka for some of the same reasons that it is useful in these other case and in some cases that are more specific to Kafka. Headers would be added to each record rather than adding the headers to some collection of records; as Kafka does not have a user visible abstraction that embodies a collection of records although Kafka may at times operate over a collection of records.

Specifically this is talking about adding an API to add a number of (key, value) pairs to each record.

Header Implementation Trade-Offs

Headers in Container

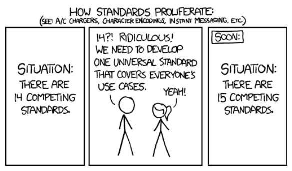
Put the headers in a container. This means users use KafkaProducer<K, ContainerWithHeaders<V>> and KafkaConsumer<K, ContainerWithHeaders<V>>. ContainerWithHeaders has the header CRUD. The user payload is now the V in ContainerWithHeaders.

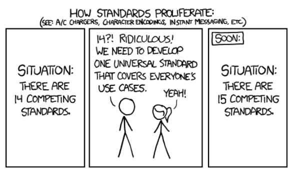
Headers in Value

Put the headers in the value but hide this from the end user. This means users still use KafkaConsumer<K, V> and KafkaProducer<K,V> but the underlying message that is sent to the broker never knows about the headers. ProduceRecord and ConsumeRecord have the CRUD methods associated with header key, value pairs. It's possible for users to implement this without it being part of the open source but at the cost of interoperability.

Headers as first class feature

As Headers in Value but the message format used internally by Kafka is now aware of the headers.

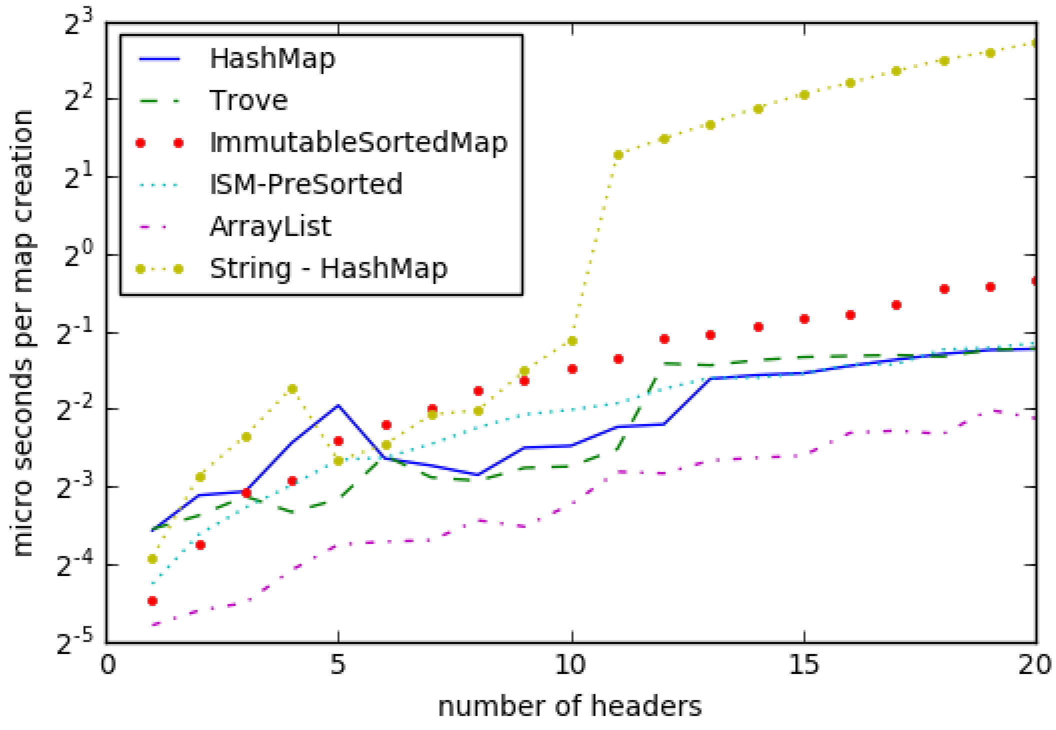
Issue	**Headers in Container (which gets stored in value)**	**Headers in Value (broker not aware)**	**Headers as first class (broker aware)**
Changes to user code not using headers	All user code needs to be changed. If user code needs to use their legacy serialization format it must now be packed into the value.	None.	None.
Long term code maintenance	 HOW STANDARDS PROLIFERATE: (SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.) SITUATION: THERE ARE 14 COMPETING STANDARDS. 14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES. YEAH! SOON: SITUATION: THERE ARE 15 COMPETING STANDARDS.	Having CRUD functionality for each header, separate from the constructor ,allows for the implementation details to be hidden from the the user. Interceptor functionality would need to be carefully considered each time the producer or consumer code is touched.	As Headers in Value and also: The internal message format used by Kafka needs to be considered. On the plus side many modifications to the internal message format can be implemented with this feature rather potentially saving substantial of long term maintenance.
Adoption	Most people that want this already have their own so there is no advantage to adopting a new container format. This is probably not sufficient reason for LinkedIn to move to a new container format.	LinkedIn and several other large users of Kafka have expressed interest in this feature. We believe this represents a large number of the active Kafka clusters.	LinkedIn and several other large users of Kafka have expressed interest in this feature. We believe this represents a large number of the active Kafka clusters.
Performance	User code may need to serialize twice.	Serializing to bytes twice means an extra copy.	Potentially no extra copies.
Online Upgrade	Only user code upgrade needs to be coordinated.	Consumers need to be upgraded and then producers.	Consumers and brokers need to be upgraded and then producers. Upgrading the

			brokers is definitely more complicated. If new features requiring changes to the message format are implemented using headers then future online upgrades for such features only require producer and consumer coordination.
Ability to implement outside of Kafka open source project.	Yes. LinkedIn already has something like this and we would rather move off of this solution.	Yes. This requires wrapping KafkaProducer<K, byte[]> and KafkaProducer<K, byte[]>. But this hampers interoperability between organizations.	Difficult. Only by maintaining a fork of Kafka which needs to be message compatible with the master. Probably no interoperability.
Interoperability between different organizations.	 HOW STANDARDS PROLIFERATE: (SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC) SITUATION: THERE ARE 14 COMPETING STANDARDS. 14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES. YEAH! SOON: SITUATION: THERE ARE 15 COMPETING STANDARDS. … so basically no.	Only if implemented in open source.	Only if implemented in open source.
Log compaction	Not with the current implementation.	Not with current implementation.	Broker can see zero length value and compact.

String vs Int Keys: Fight

Collections

This is a plot of different collection implementations for headers and the time it takes to populate one of them (y-axis) with the specified number of headers (x-axis). This is the mean time for 1M instantiations. A warmup of 100k iterations over each container is performed before benchmarking. The same value is always used: a byte array of length zero.



HashMap is HashMap<Integer, byte[]>.
Trove is the TIntObjectHashMap<byte[]> implementation from GNU Trove. It uses primitive int for keys and open addressing rather than chaining. This is to test the effects of a different hash table algorithm.
ImmutableSortedMap is the sorted map implementation from Guava. It backs the map with an array rather than a binary search tree. This is see if hashing is some kind of problem.
ISM-PreSorted is a guava ImmutabledSortedMap, but the keys are added in order so sorting is not needed.
ArrayList<Header<Integer, byte[]>>. A Header is just the key,value pair that is the header. If we allowed for duplicate header keys (i.e. they are not keys) and we don't care about lookup performance then this is probably close to the best we can do.
String - HashMap is HashMap<FakeString, byte[]>. A wrapper around char[] is used rather than an actual String since String caches the results of computing hashCode() and String(String) propagates the computed hashCode. FakeString uses the same hashCode() and equals() algorithm as String.

There is a jump at 10 headers for Map<String, byte[]> creation time. It’s not clear why this happens as the plot has the same shape even when the HashMap is preallocated to a capacity of 30 (not shown).

At the low end all the implementations and key representations are probably within the measurement error of this benchmark (not shown).

Going with other Map implementations only differ in CPU performance by a factor of two. They don't seem worth considering. Requiring ordering of headers so that an ordered map can used also does not seems like a worthwhile optimization.

Map<String, V> is probably ok with respect to map computation time, for small numbers of headers. When the number of headers is more than 10 it looks like there are performance issues.

Selecting a map implementation other than HashMap is probably not worth it with respect to header representations. Probably it's worth considering Radai's proposal of lazy parsing the headers in the ConsumerRecord so that it can be done in a different thread if needed.

machine mac laptop

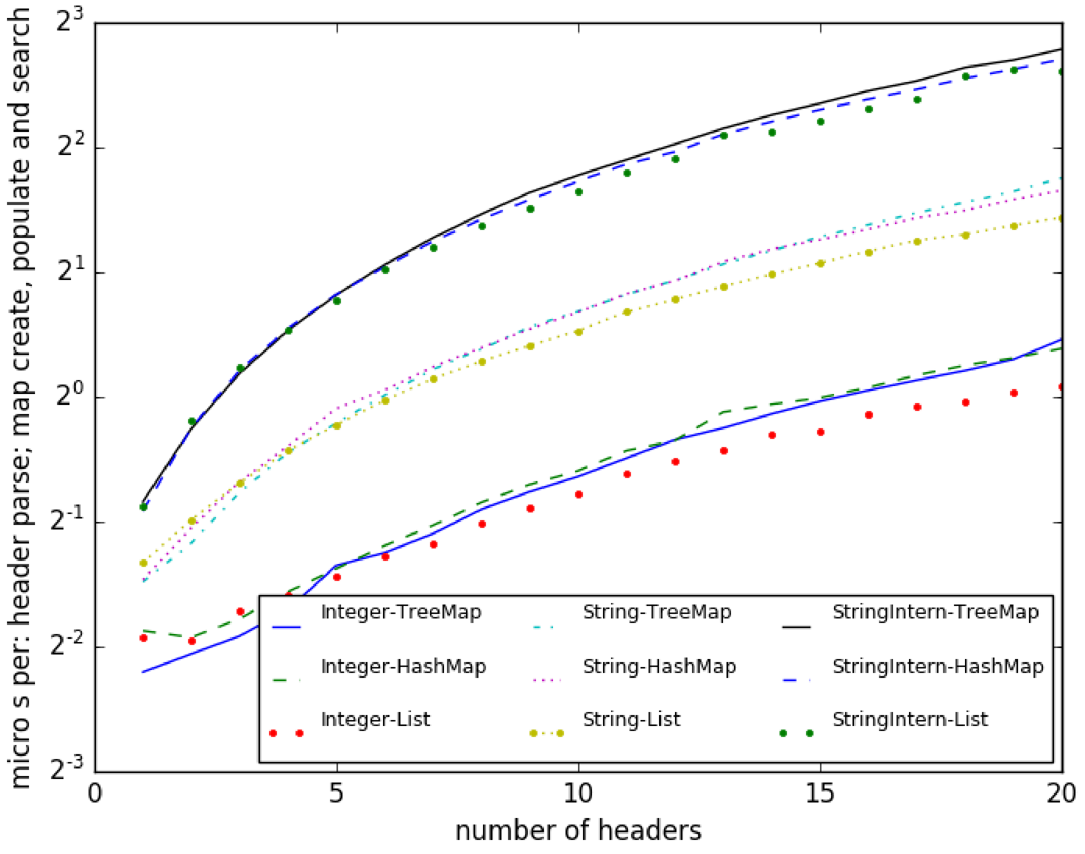
Parse Header, create collection, populate collection search collection

This benchmark measures:

1. create a map
2. deserializes a key, value pair
3. populates the map with the key value pairs
4. searches the map for the known key value pairs (20)

create a map: Uses HashMap, TreeMap or a ListMap. This last one is a map backed by an ArrayList.
deserializes: There is a fixed set of strings that look kind of like keys one might actually use. For some of the string tests String.intern() is called (this is Java's Flightweight implementation for Strings). Integers are just in some increasing range much larger than 255.
searches the map: There is a fixed table of random permutations that are generated and reused. These are used to generate searches generated map. This is done 20 times regardless of how many items were added to the map.

machine Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz which is probably more like a server than a mac laptop.



Conclusions.
* String as header keys are fine unless you need ultimate performance in which case you should probably not use headers which his still a viable option.
* String intern() is expensive and you only really need to do this if the ConsumerRecord is going to hang around for some time. So probably no.
 desktop machine Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz

Table of string keys used in these benchmarks:

```
static char[][] keys = new char[][] {
   "linkedin.consumer.class".toCharArray(),
     "kafka.feature1".toCharArray(),
     "kafka.feature2".toCharArray(),
     "LinkedIn.group.id".toCharArray(),
     "LinkedIn.avro.schema.id".toCharArray(),
     "something.something.darkside.something.something.complete".toCharArray(),
     "cats.don't.care.about.kafka".toCharArray(),
     "argle bargle".toCharArray(),
     "org.apache.kafka.something.something.too.long".toCharArray(),
     "LinkedIn.priority".toCharArray(),
     "LinkedIn.one-more-key".toCharArray(),
     "header-11".toCharArray(),
     "header-12a".toCharArray(),
    "header-13aa".toCharArray(),
    "header-14".toCharArray(),
    "header-15".toCharArray(),
    "header-16aaa".toCharArray(),
    "header-17".toCharArray(),
    "header-18".toCharArray(),
    "header-19".toCharArray()
};
```