KIP-110: Add Codec for ZStandard Compression

Status

Current state: *Accepted*

Discussion thread: *thread1 thread2*

JIRA: KAFKA-4514

Released: 2.1.0

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

On September 2016, Facebook announced a new compression implementation named ZStandard, designed to scale with modern data processing environment. With its great performance in both speed and compression rate, lots of popular big data processing frameworks are supporting ZStandard.

- Hadoop (3.0.0) - ⚠ Unable to render Jira issues macro, execution error.
- HBase (2.0.0) - ⚠ Unable to render Jira issues macro, execution error.
- Spark (2.3.0) - ⚠ Unable to render Jira issues macro, execution error.

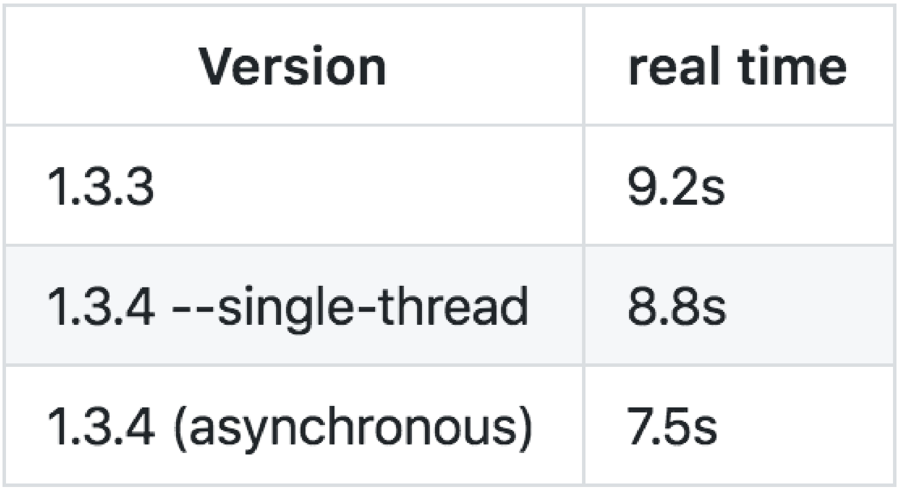
ZStandard also works well with Apache Kafka. Benchmarks with the draft version (with ZStandard 1.3.3, Java Binding 1.3.3-4) showed significant performance improvement. The following benchmark is based on Shopify's production environment (Thanks to @bobrik)

blocked URLblocked URL

(Above: Drop around 22:00 is zstd level 1, then at 23:30 zstd level 6.)

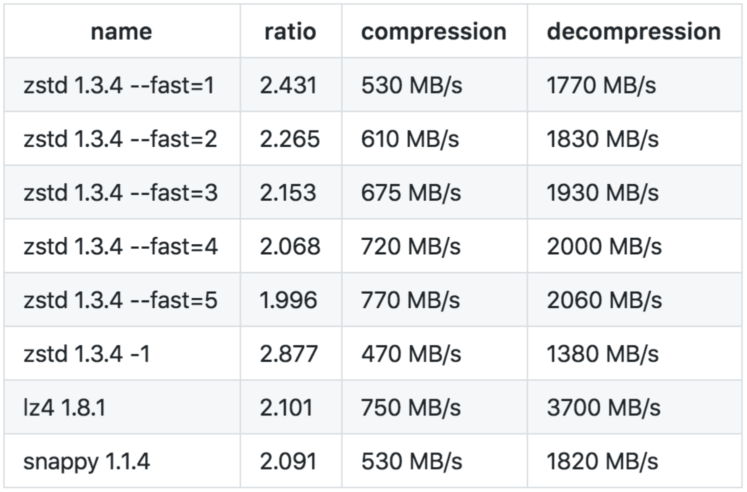
As You can see, ZStandard outperforms with a compression ratio of 4.28x; Snappy is just 2.5x and Gzip is not even close in terms of both of ratio and speed.

It is worth noting that **this outcome is based on ZStandard 1.3**. According to Facebook, ZStandard 1.3.4 improves throughput by 20-30%, depending on compression level and underlying I/O performance.



Version	real time
1.3.3	9.2s
1.3.4 --single-thread	8.8s
1.3.4 (asynchronous)	7.5s

(Above: Comparison between ZStandard 1.3.3. vs. ZStandard 1.3.4.)



name	ratio	compression	decompression
zstd 1.3.4 --fast=1	2.431	530 MB/s	1770 MB/s
zstd 1.3.4 --fast=2	2.265	610 MB/s	1830 MB/s
zstd 1.3.4 --fast=3	2.153	675 MB/s	1930 MB/s
zstd 1.3.4 --fast=4	2.068	720 MB/s	2000 MB/s
zstd 1.3.4 --fast=5	1.996	770 MB/s	2060 MB/s
zstd 1.3.4 -1	2.877	470 MB/s	1380 MB/s
lz4 1.8.1	2.101	750 MB/s	3700 MB/s
snappy 1.1.4	2.091	530 MB/s	1820 MB/s

(Above: Comparison between other compression codecs, supported by Kafka.)

As of September 2018, the draft implementation uses Java binding for ZStandard 1.3.5.

Public Interfaces

This feature introduces a new available option 'zstd' to the compression.type property, which is used in configuring Producer, Topic and Broker. It's id will be 4.

It also introduces a new error code, UNSUPPORTED_COMPRESSION_TYPE (74). For details on this error code, see 'Compatibility, Deprecation, and Migration Plan' section.

Proposed Changes

1. Add a new dependency on the Java bindings of ZStandard compression.
2. Add a new value on CompressionType enum type and define ZStdCompressionCodec on kafka.message package.
3. Add a new error type, 'UNSUPPORTED_COMPRESSION_TYPE'.
4. Implement the compression logic along with compatibility logic described below.

You can check the concept-proof implementation of this feature on this Pull Request.

Compatibility, Deprecation, and Migration Plan

We need to establish some backward-compatibility strategy for the case an old client subscribes a topic using ZStandard, explicitly or implicitly (i.e., 'compression.type' configuration of given topic is 'producer' and the producer compressed the records with ZStandard). After discussion, we decided to support zstd to the new clients only (i.e., uses v2 format) and return UNSUPPORTED_COMPRESSION_TYPE error for the old clients.

Here is the detailed strategy:

1. Zstd will only be allowed with magic = 2 format. That is,
    - Instantiating MemoryRecords with magic < 2 is disallowed.
    - Down-conversion of zstd-compressed records will not be supported. So if the requested partition uses 'producer' compression codec and the client requests magic < 2, the broker will down-convert the batch until before using zstd and return with a dummy oversized record in place of the zstd-compressed batch. When the client attempts to fetch from then on, it will receive a UNSUPPORTED_COMPRESSION_TYPE error.
2. Bump produce and fetch request versions. It will give the old clients a message to update their version.
3. Zstd will only be allowed for the bumped produce API. That is, for older version clients (=below KAFKA_2_1_IV0), we return UNSUPPORTED_COMPRESSION_TYPE regardless of the message format.
4. Zstd will only be allowed for the bumped fetch API. That is, if the requested partition uses zstd and the client version is below KAFKA_2_1_IV0, we return UNSUPPORTED_COMPRESSION_TYPE regardless of the message format.

The following section explains why we chose this strategy.

Rejected Alternatives

A. Support ZStandard to the old clients which can understand v0, v1 messages only.

This strategy necessarily requires the down-conversion of v2 message compressed with Zstandard into v0 or v1 messages, which means a considerable performance degradation. So we rejected this strategy.

B. Bump the API version and support only v2-available clients

With this approach, we can message the old clients that they are old and should be upgraded. However, there are still several options for the Error code.

B.1. INVALID_REQUEST (42)

This option gives the client so little information; the user can be confused about why the client worked correctly in the past suddenly encounters a problem. So we rejected this strategy.

B.2. CORRUPT_MESSAGE (2)

This option gives inaccurate information; the user can be surprised and misunderstand that the log files are broken in some way. So we rejected this strategy.

B.3 UNSUPPORTED_FOR_MESSAGE_FORMAT (43)

The advantage of this approach is that we don't need to define a new error code; we can reuse it and that's all. The disadvantage of this approach is that it is also a little bit vague; This error code is defined as a work for KIP-98 and now returned in the transaction error.

Since adding a new error type is not a big problem and a clear error message always helps, we decided to reject this strategy.

Related issues

This update introduces some related issues like following.

Whether to use existing library or not

There are two ways of adapting ZStandard to Kafka, each of which has its pros and cons.

1. Use existing bindings.
    - Pros
        - Fast to work.
        - The building task doesn't need ZStandard to be pre-installed to the environment.
    - Cons
        - Somebody has to keep the eyeballs on the updates of both of the binding library and ZStandard itself. If needed, he or she has to update the binding library to adapt them to Kafka.
2. Add existing JNI bindings directly.
    - Pros
        - Can concentrate on the updates of ZStandard only.
    - Cons
        - ZStandard has to be pre-installed before building Kafka.

        - A little bit cumbersome to work.

The draft implementation adopted the first approach, following its Snappy support. (In contrast, Hadoop follows the latter approach.) You can see the used JNI binding library at here. However, I thought it would be much better to discuss the alternatives, for I am a newbie to Kafka.

License

We can use zstd and its Java binding, zstd-jni without any problem, but we need to include their license to the project - BSD and BSD 2 Clause license, respectively. They are not listed in the list of prohibited licenses also.

What we need is attaching the licenses for the dependencies only. A recent update on Apache Spark shows how to approach this problem. They did:

- 'LICENSE' file: License of the project itself (i.e., Apache License) and the list of source dependencies and their licenses.
- 'LICENSE-binary' file: The list of binary dependencies and their licenses.
- 'license' directory: Contains the license files of source dependencies.
- 'license-binary' directory: Contains the license files of binary dependencies.