KIP-1008: ParKa - the Marriage of Parquet and Kafka

Status

Current state: *"Under Discussion"*

Discussion thread: *here*

JIRA: *here [Change the link from KAFKA-1 to your own ticket]*

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

Note this is a joint work proposed by Xinli Shang Qichao Chu Zhifeng Chen @Yang Yang

Motivation

Kafka is typically used in conjunction with Avro, JSON, or Protobuf etc to serialize/deserialize data record by record. In the producer client, records are buffered as a segment (record batch), and compression is optionally applied to the segment. When the number of records in each segment is larger, columnar storage like Apache Parquet becomes more efficient in terms of compression ratio, as compression typically performs better on columnar storage. This results in benefits for reducing traffic throughput to brokers/consumers and saving disk space on brokers.

In the use case of using Kafka for data lake ingestion, if we can produce the segment with Parquet, which is the native format in a data lake, the consumer application (e.g., Spark jobs for ingestion) can directly dump the segments as raw byte buffer into the data lake without unwrapping each record individually and then writing to the Parquet file one by one with expensive steps of encoding and compression again. This would provide additional savings for both data freshness of ingestion and resource consumption, as the ingestion job would be lighter weight and finishes significantly faster.

Parquet has a built-in column encryption feature that would enable Kafka to have field encryption for free, offering several advantages: 1) It applies AES to the block of fields (i.e., Parquet page), minimizing overhead (5.7% for write and 3.7% for read). 2) It supports all data types of fields, including Boolean. 3) It is mature and deployed at scale e.g. Uber, Apple, and AWS. 4) It operates at the data format layer, requiring Kafka only to enable it.

Note: Although we propose adopting Parquet as the pilot, the design and implementation should not be limited to Parquet only. Other formats like ORC should be easily added later, although it is outside the scope of this KIP.

Public Interfaces

This KIP introduces the following additions to the public interfaces.

Client API changes

In the interface Producer<K,V>, we are going to add a new method as below

void setSchema(String topic, String schema);

No change is needed for Consumer<K, V> because KIP-712 already introduced 'fetch.raw.bytes' so that the ingestion consumer can fetch the byte buffer directly.

No broker changes are needed.

Proposed Changes

We propose adding Parquet as the encoder and optionally compressor in Kafka producer client. When this feature is enabled, Parquet is used to encode the batch records segment and optionally compress. Parquet has the encoding and compression in a columnar-oriented way.

We divide consumers into two categories. Minimum change is needed for messaging consumers for reading the Parquet segment. The way to divide into the two categories and these terms are used solely for the purpose of this proposal.

Messaging Consumer

In this scenario, the application expects one or more records with each poll. When the Kafka consumer client encounters the Parquet format, it invokes the Parquet reader library to unwrap the segment into records. The required change is similar to the producer side, as discussed above.

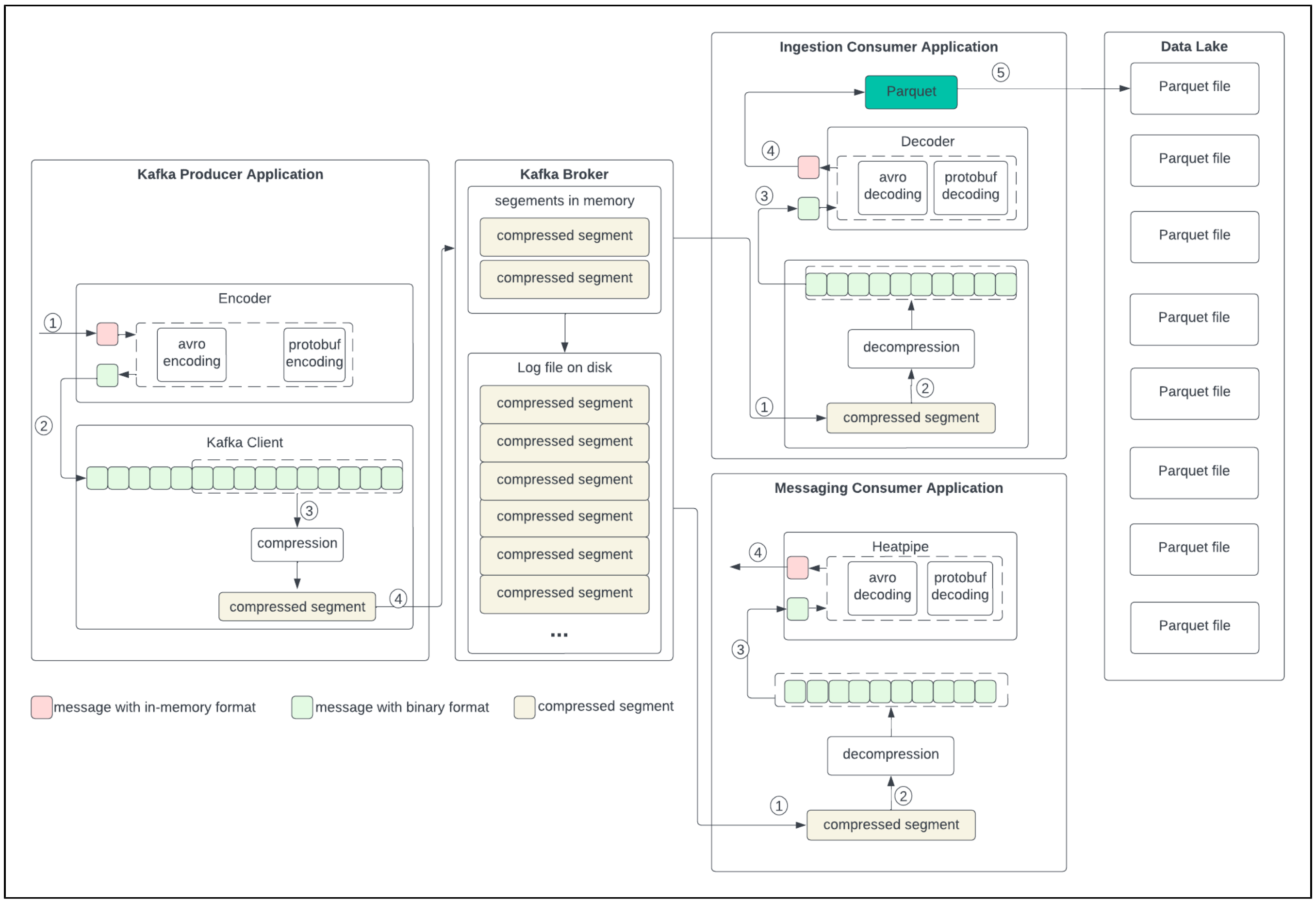
Ingestion Consumer

For applications expecting a batch of records or even the entire segment to write to the sink (data lake), the Kafka consumer client can simply return byte buffer of the entire segment to the application, allowing it to directly dump it into the sink.

We refer to the first type of use case as a messaging consumer and the second one as an ingestion consumer.

Current Data Format Transformation

To set up the context for discussing the changes in the next section, let's examine the current data formats in the producer, broker, and consumer, as well as the process of transformation outlined in the following diagram. We don't anticipate changes to the broker, so we will skip discussing its format.



Producer

The producer writes the in-memory data structures to an encoder to serialize them to binary and then sends them to the Kafka client.

1. The application uses an encoder, such as Avro or Protobuf encoder, to serialize the in-memory data structure into a binary record per record.

2. Each record in binary format is sent to the Kafka client, which has a buffer to form a batch, i.e., a segment.
3. Optionally, the segment is sent to a compressor to apply compression.
4. The compressed segment is then sent to the broker.

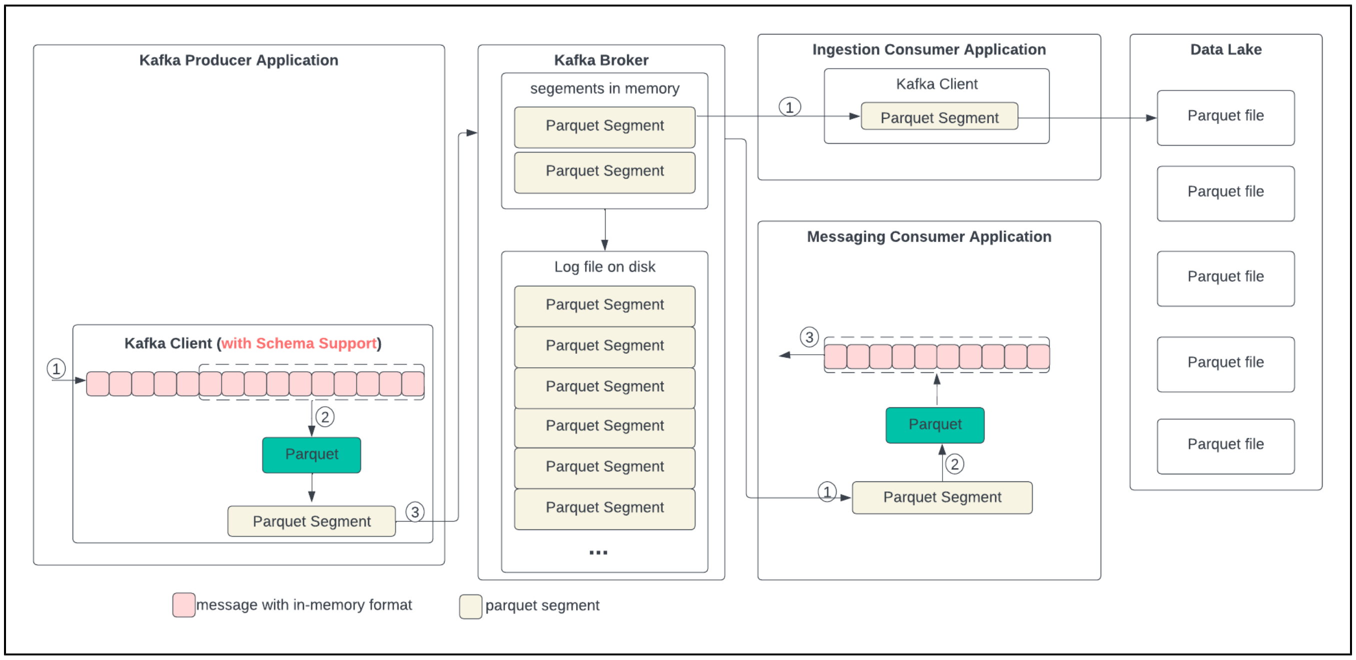
Consumer

The process of handling the data is the reverse of the producer. In both messaging and ingestion consumers, the process is the same, except that the ingestion consumer needs to write to the Parquet.

1. The consumer gets the segment from the broker.
2. If the segment is compressed, the compressor decompresses it.
3. The record in binary format is sent to the decoder.
4. The decoder decodes the record to the in-memory format and returns it to the application.
5. For the ingestion consumer, it writes to the Parquet format record per record and then applies Parquet encoding and compression.

Proposed Data Format Transformation

In the following diagram, we describe the proposed data format changes in each state and the process of the transformation. In short, we propose replacing compression with Parquet. Parquet combines encoding and compression at the segment level. The ingestion consumer is simplified by solely dumping the Parquet segment into the data lake.



Producer

The producer writes the in-memory data structures directly to the Kafka client and encodes and compresses all together.

1. The application writes in-memory data structures to the Kafka client, which has a buffer to form a batch, i.e., a segment.
2. The segment is encoded and optionally compressed using Parquet.
3. The segment with the Parquet format is then sent to the broker.

The needed changes in the producer Kafka client are: 1) Add Parquet as a new encoder, 2) Add a new method to pass in the schema in the Producer<K, V> for topics, which is needed for Parquet encoding,

Consumer

The process of handling the data differs between ingestion and messaging consumers.

Messaging Consumer

1. The consumer gets the segment from the broker.
2. If the segment is in Parquet format, it uses the Parquet library to decode and optionally decompress.
3. The record in the in-memory format is then sent to the application.

Ingestion Consumer

1. The consumer gets the segment with Parquet format from the broker and sends it directly to the data lake.

The needed changes are: 1) Add Parquet as the decoder. 2) Add an API to allow the application to get the record as a ByteBuffer for ingestion.

Configuration

The following two configurations are to be added.

columnar.encoding

The columnar encoding type for the data generated for a topic by the producer. The default is none (i.e., no columnar encoding). Valid values are Parquet for now and future to include ORC. When the value is not none, the key.serializer and value.serializer is disabled because the column encoder will serialize and deserialize the data. Column encoding is for full batches of data, so the efficacy of batching will also impact the efficiency of columnar encoding (more batching means better encoding).

Type:	string
Default:	none
Valid Values:	[none, parquet]
Importance:	low

columnar.encoding.compression

The columnar encoders, such as Parquet, have their own compression methods. This configuration specifies the compression method in the columnar encoding. The default is none (i.e., no compression). Valid values are snappy, gzip, zstd, and lz4. When the value is not none, the data in the columnar format will be compressed using the specified method, and the specified compression in compression.type will be disabled. If the value is none, the specified compression in compression.type will work as before. Column encoding is for full batches of data, so the efficacy of batching will also impact the efficiency of columnar compression (more batching means better compression).

Type:	string
Default:	none
Valid Values:	[none, gzip, snappy, lz4, zstd]
Importance:	low

Compatibility, Deprecation, and Migration Plan

When the configuration columnar.encoding is set to 'none', both the producer and consumer work as before.

When the configuration columnar.encoding is set to a value other than 'none,' the producer needs to change by calling the new API setSchema() to add a schema. The consumer, without the proposed change, should still be able to poll the records as before. Optionally, the consumer can call the new method pollBuffer() to get the segment in the columnar encoding format.

Test Plan

Functional tests

1. Regression test - The configuration ‘columnar.encoding’ is set to 'none’, run all the tests in Uber staging env which includes but is not limited to read/write with different scales.
2. Added feature - The configuration ‘columnar.encoding’ is set to 'parquet’.
    a. Verify the data is encoded as Parquet format
    b. The producer, broker, and consumer all work as before functionality-wide. No exceptions are expected.
    c. The consumer fetches the data as byte buffer with 'fetch.raw.bytes' set and the data is with Parquet format.

Performance tests

Run tests for different topics that should have different data types and scale

1. Benchmarking the data size when the number of rows is changed in the batch.
2. Benchmarking CPU utilization on producer and consumer

Compatibility tests

Test the compatibility among producer, consumer, and replicator with/without the proposed changes.

1. Both producer and consumer have the proposed changes
    a. The feature is turned off in configuration, all regression tests should work as before.
    b. When the producer turns on this feature, the consumer and replicator can consume as before.
2. Producer has the proposed changes, but the consumer doesn’t
    a. The feature is turned off in configuration, all regression tests should work as before.
    b. When the producer turns on this feature, the consumer and replicator throw an exception
3. Producer doesn’t have the proposed changes, but the consumer does
    a. All the regression tests should pass

Rejected Alternatives

The alternative is to apply columnar encoding and compression outside Kafka clients. The application can add a buffer to create a batch for records, apply columnar encoding and compression, and then put them into the (K, V) in the ProducerRecord. The benefit of doing this is to avoid changes in the Kafka client, but there are problems with this approach, as outlined below:

1. It changes the consumer commit semantics. The consumer application can only commit as a batch.
2. It causes duplication of effort in building buffering and batching, which is already present in the Kafka client today.
3. Kafka application engineers need to do all the work, losing the benefit that we proposed, which is that they can just enable it.