KIP-868 Metadata Transactions

Status

Current state: Accepted

Discussion thread: https://lists.apache.org/thread/895pgb85l08g2l63k99cw5dt2qpjkxb9

JIRA: ⚠ Unable to render Jira issues macro, execution error.

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

Motivation

In KRaft, we use record batches as a mechanism for atomicity. When performing a metadata update that generates a number of related records, we will commit them atomically as a batch to the KRaft quorum. This ensures that the entire batch will be appended to the KRaft layer, or none of it will. This atomicity is very important when it comes to controller failover. If a controller commits a partial set of records and then crashes, the uncommitted records will be lost.

One limitation of the current approach is that the maximum batch size is limited by the maximum fetch size in the Raft layer. Currently, this value is hard coded to 8kb. In extreme cases, the controller may generate a set of atomic records that exceed this size.

A practical use case is creating a topic with a huge number of partitions. We either want all the records (TopicRecord + PartitionRecords) to be processed by the controller and brokers, or none of them to be processed. The only way we can achieve this atomicity today is by using an atomic Raft batch. With this KIP, we can atomically create topics with an arbitrary number of partitions.

This KIP details an approach to allow the controller to generate atomic transactions of records that can exceed the maximum batch size.

Public Interfaces

Three simple marker records will be introduced to the KRaft metadata layer. These records will represent the beginning and end of a transaction.

```
{
  "apiKey": TBD
  "type": "metadata",
  "name": "BeginTransactionRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    {"name": "Name", "type": "string", "versions": "0+",
     "nullableVersions": "0+", "taggedVersions": "0+", "tag": 0,
     "about": "An optional textual description of this transaction. Should not exceed 255 bytes"}
  ]
}
```

```
{
  "apiKey": TBD
  "type": "metadata",
  "name": "EndTransactionRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": []
}
```

```
{
  "apiKey": TBD
  "type": "metadata",
  "name": "AbortTransactionRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    {"name": "Reason", "type": "string", "versions": "0+",
     "nullableVersions": "0+", "taggedVersions": "0+", "tag": 0,
     "about": "An optional textual explanation of why this transaction was aborted. Should not exceed 255
bytes"}
  ]
}
```

A new *metadata.version* will be introduced to gate the usage of these new records.

Proposed Changes

Transactions in the KRaft Controller

We define a new concept in the controller called a metadata transaction. A transaction consists of a BeginTransaction, a number of records, and either an EndTransaction or AbortTransaction. These records can be committed atomically to the Raft layer in multiple batches. By allowing a set of records to span across several record batches, we overcome the current batch size limitation and further decouple the controller from Raft layer implementation details.

We can define a few invariants to help narrow the design:

- Only a single transaction may exist at a time
- A transaction cannot be interleaved with records outside the transaction
- All writes to Raft for a transaction must be atomic, including the two marker records

By implementing transactions in the "application layer" (as opposed to the storage layer, i.e., Raft), we can be efficient about handling the incomplete records. For example, instead of actually buffering the incomplete records in-memory, we can apply them to the controller's timeline data structures optimistically. If the transaction is aborted, we can easily reset the state of these data structures.

An example topic creation using a transaction with two atomic batches:

Record	Record in transaction	Record in batch
BeginTransactionRecord	1	1
TopicRecord	2	2
PartitionRecord	3	3
PartitionRecord	4	4
PartitionRecord	5	1
PartitionRecord	6	2

ConfigRecord	7	3
EndTransactionRecord	8	4

Record Visibility

As the controller sends records to Raft, it will optimistically apply the record to its in-memory state. The record is not considered durable (i.e., "committed") until the high watermark of the Raft log increases beyond the offset of the record. Once a record is durable, it is made visible outside the controller to external clients. Since we are defining transactions that may span across Raft commits, we must extend our criteria for visibility to include whole transactions. When the controller sees that a partial transaction has been committed, it must not expose the records until the entire transaction is committed. This is how the controller will provide atomicity for a transaction.

For the broker, we must also prevent partial and aborted transactions from becoming visible to broker components. During normal operation, the records fetched by the brokers will include everything up to the Raft high watermark. This will include partial transactions. The broker must buffer these records and not make them visible to its internal components. This buffering can be achieved using the existing metadata infrastructure on the broker. That is, we will apply records to MetadataDelta, but not publish a new MetadataImage until an EndTransaction or AbortTransaction is seen.

The metadata shell must also avoid showing partially committed metadata transactions.

Controller Failover

Another aspect of atomicity for transactions is dealing with controller failover. If a transaction has not been completed when a controller crashes, there will be a partial transaction committed to the log. The remaining records for this partial transaction are forever lost, and so we must abort the entire transaction. This will be done by the newly elected controller inserting an AbortTransaction.

When the controller aborts a transaction, it will reset its in-memory state to the point before the transaction began. This effectively rolls back the applied records from the partially committed transaction. Because of the visibility guarantees mentioned in the previous section, there is no risk of these applied records being exposed to external clients.

Since we only allow for a single transaction at a time, detecting a partial transaction during failover is easily determined. An incomplete transaction would only occur at the end of the log. Using the above example, an incomplete transaction might look like:

- BeginTransactionRecord
- TopicRecord
- PartitionRecord
- PartitionRecord

There will be control records in the Raft log after this partial transaction (e.g., records pertaining to Raft election), but these are not exposed to the controller directly. To abort this transaction, the controller would write a AbortTransactionRecord to the log and revert its in-memory state.

Snapshots

Since snapshots are already atomic as a whole, there are no need for transaction records. On both the broker and controller, we generate snapshots based on in-memory state using arbitrary batch sizes. This design does not call for any special in-memory treatment of metadata transactions – they are simply a mechanism of atomicity to use within the log itself. Records applied to memory will not know if they came from a non-atomic batch, atomic batch, or transaction. Snapshot generation is unchanged by this KIP.

Compatibility, Deprecation, and Migration Plan

These new records will be incompatible with older versions of the broker and controller, and so we will gate their usage by introducing a new *metadata. version*. If the metadata log is downgraded to a version that does not support metadata transactions, we will simply exclude them from the downgrade snapshot that is produced. Note that lossy downgrades of the metadata log are detailed in KIP-778 and not yet fully implemented as of Kafka 3.3.

Test Plan

The most interesting aspect of this design to test at a system level is controller failover. We can use a system test to continuously create a topics with a large number of partitions during a failover event. This will allow us to observe and verify the recovery behavior on the new controller.

We will need good unit test coverage within the QuorumController since this design will be the first time we are rolling back committed records. Normally, the only time the controller resets its state is when we are handling a snapshot.

Rejected Alternatives

Raft Transactions

An alternative approach would be to put transactions in the Raft layer. This would be possible, but likely a much larger effort with a more complex implementation. The Raft layer in Kafka does not enforce the single-writer semantics that we have on the controller, so Raft transaction support would need to deal with concurrent transactions (much like we do with EOS). This would also likely involve changing the RaftClient#Listener interface. Since the controller is the single-writer of the metadata log, we can simplify the design quite a bit by managing a single transaction in that layer.

Another disadvantage of buffering transactions in Raft is that we would have to be very "dumb" about the buffering. Since Raft has no context of the contents of the records, it cannot do anything besides simply buffering the un-finished transaction records in memory. By doing transactinos in the controller /broker, we can be much smarter and more efficient about how we buffer things.