KIP-387: Fair Message Consumption Across Partitions in KafkaConsumer

Status

Current state: *Under Discussion*

Discussion thread: *here [Change the link from the KIP proposal email archive to your own email thread]*

JIRA: *KAFKA-3932 [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).

Motivation

The KafkaConsumer API centers around the poll() API which is intended to be called in a loop. On every iteration of the loop, poll() returns a batch of records from the partitions this consumer can retrieve at that time. The size of returned records is determined by the *max.poll.records*, as described in KIP-41: KafkaConsumer Max Records. Currently the implementation will return available records starting from the last partition the last poll call retrieves records from. This leads to unfair patterns of record consumption from multiple partitions.

This proposal discusses a mechanism to mitigate that issue.

Public Interfaces

No public interface changes is proposed.

Proposed Changes

The issue stems from the greedy consumption of a partition in serving a poll call, as described in Ensuring Fair Consumption of KIP-41, to be used again in the next poll call, and so continue that greedy behavior against that previous partition in the next call.

The simplest solution is to pick another partition that has available records as the starting point for the next poll call. The current implementation keeps the partitions with received records in a queue called *completedFetches* inside the *consumer.internals.Fetcher* class, the main class to return records from the poll call. Partitions in that queue is ordered by when the *Fetcher* receives the partition messages. We can pick the next partition from that queue to serve the next poll call instead. To avoid parsing the partition messages repeatedly, we can save those parsed fetches to a list (*parsedFetches*) and maintain the next partition to get messages there.

The logic will use partitions from *completedFetches* to retrieve records in the original greedy fashion, and move them to the *parsedFetches* list after they are parsed. When *completedFetches* queue is empty, it will consume records in partitions in the *parsedFetches* list in round robin order. The partition with parsing errors will be moved to the end of the *completedFetches* queue to return records to the current and subsequent poll calls successfully.