KIP-392: Allow consumers to fetch from closest replica

Status

Current state: *Accepted*

Discussion thread:

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

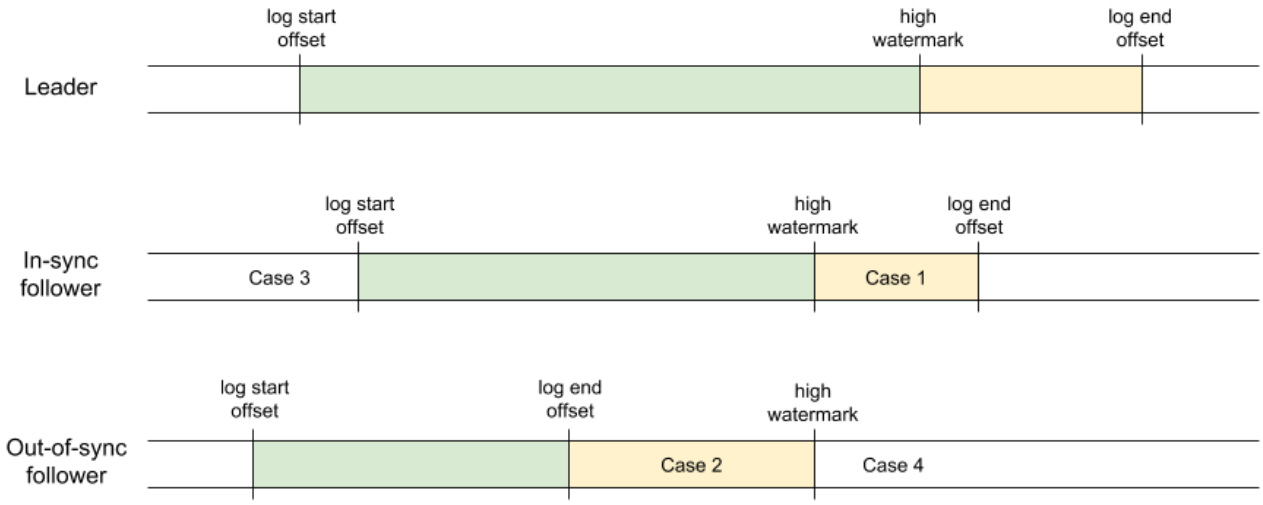
It is common to have a Kafka cluster spanning multiple datacenters. For example, a common deployment is within an AWS region in which each availability zone is treated as a datacenter. Currently, Kafka has some basic support for rack awareness which can be used in this scenario to control replica placement (by treating the availability zone as a rack). However, currently consumers are limited to fetching only from the leader, so there is no easy way to leverage locality in order to reduce expensive cross-dc traffic. To address this gap, we propose here to allow consumers to fetch from the closest replica.

Proposed Changes

In order to support fetching from the closest replica, we need 1) to add support in the Fetch protocol for follower fetching, and 2) a mechanism to find the "closest" replica to a given consumer.

Follower Fetching

We propose to extend the fetch protocol to allow consumer fetching from any replica. Just as when fetching from the leader, the replication protocol will guarantee that only committed data is returned to the consumer. This relies on propagation of the high watermark through the leader's fetch response. Due to this propagation delay, follower fetching may have higher latency. Additionally, follower fetching introduces the possibility of spurious out of range errors if a fetch is received by an out-of-sync replica. The diagram below illustrates the different cases that are possible.



In this diagram, the green portion of the log represents the records that are both available locally on the replica and known to be committed. These records can be safely returned to consumers at any time. The yellow portion represents the records which are either not available locally or not known to be committed (i.e. they have not been successfully replicated to all members of the ISR). These records cannot be returned until the high watermark is known to have advanced.

The leader always sees high watermark updates before any of the replicas. An in-sync replica may therefore have some committed data which is not yet available for consumption. In the case of an out-of-sync replica, the high watermark is typically above its log end offset, so there may be some committed data which is not yet locally available.

Note that the log start offsets are not consistently maintained between the leader and followers (even those in the ISR). This is due to the fact that retention is enforced independently by each replica. We will discuss this point in more detail below.

High watermark propagation

Some extra latency is expected when fetching from followers, but we should ensure that there are no unnecessary delays in the propagation of the high watermark. Following the improvements in KIP-227, the leader can use the fetch session to track the high watermark of the follower. Currently, updating the follower's high watermark could be delayed by as long as `replica.fetch.max.wait.ms` if there is no new data written to the partition.

We propose to change the fetch satisfaction logic to take high watermark updates into account. The leader will respond to a fetch immediately if the follower has a stale high watermark. In particular, the first request in a fetch session will not block on the leader in order to ensure that the follower has the latest high watermark for requested partitions. A side benefit of this change is that it reduces the window following a leader election in which the high watermark must be withheld from clients in order to preserve its monotonicity (see KIP-207).

Out of range handling

The main challenge for follower fetching is that a consumer may observe a committed offset on the leader before it is available on a follower. We need to define the expected behavior if the consumer attempts to fetch a valid offset from a follower which is relying on stale log metadata.

There are four cases to consider (each identified in the diagram above):

Case 1 (uncommitted offset): The offset is available locally on the replica receiving the fetch, but is not known to be committed. In fact, the broker already has logic to handle this case. When a replica is elected leader, it does not know what the true value of the high watermark should be until it receives fetches from the current ISR. If a consumer fetches at an offset between the last updated high watermark value and the log end offset, then the broker currently returns an empty record set. This ensures that no records are returned until they are known to be committed.

A similar problem was addressed in KIP-207. Prior to this, a leader which was just elected might return a stale value of the high watermark. With the adoption of KIP-207, the broker instead returns the OFFSET_NOT_AVAILABLE error code until the true high watermark can be determined. This is slightly inconsistent with the fetch behavior. The unintended side effect of returning an empty record set (and no error) is that it might expose an offset before it is known to be committed.

One of the improvements we make in this KIP is to make the behavior of these two APIs consistent. When the broker receives a fetch for an offset between the local high watermark and the log end offset, whether it's on the leader or follower, we will return the OFFSET_NOT_AVAILABLE error code. The consumer will handle this by retrying.

Case 2 (unavailable offset): The offset is not available locally, but is known to be committed. This can occur if a fetch is received by an out-of-sync replica. Even out-of-sync replicas will continue fetching from the leader and they may have received a high watermark value which is larger than its log end offset. So a fetch at an offset between the log end offset and the high watermark received from the leader is known to be committed though the replica does not have the data.

Ideally, consumers should fetch only from the ISR, but it is not possible to prevent fetching from outside the ISR because it takes time for the ISR state to propagate to each replica. We propose here to also return OFFSET_NOT_AVAILABLE for this case, which will cause the consumer to retry. In the common case, we expect the out-of-sync replica to return to the ISR after a short time. On the other hand, if the replica remains out-of-sync for a longer duration, the consumer will have a chance to find a new replica after some time. We discuss this point in more detail below in the section on how the consumer finds its preferred replica.

Case 3 (offset too small): A valid offset on one replica is smaller than the log start offset on another replica. To detect this case, we propose a minor change to the fetch response. When returning an OUT_OF_RANGE error, the broker will provide its current log start offset and high watermark. The consumer can compare these values with the fetch offset to determine whether the offset was too small or too large.

Reasoning about this case on the client is difficult due to the absence of guarantees on the consistency of the log start offset. However, we expect the log start offset of replicas to be relatively close, so the benefit of any additional handling seems marginal. Here we propose to skip offset reconciliation as outlined in KIP-320 when handling this error and simply resort to the offset reset policy.

Note that when the offset reset policy is set to "earliest," we need to find the log start offset of the current replica we are fetching from. The log start offset for the leader is not guaranteed to be present on the follower we are fetching from and we do not want the consumer to get stuck in a reset loop while we wait for convergence. So to simplify handling, the consumer will use the earliest offset on whatever replica it is fetching from.

In summary, to handle this case, the consumer will take the following steps:

1. If the reset policy is "earliest," fetch the log start offset of the current replica that raised the out of range error.
2. If the reset policy is "latest," fetch the log end offset from the leader.
3. If the reset policy is "none," raise an exception.

Case 4 (offset too large): The offset is not available locally on the replica and is not known to be committed. This can happen as in case 2 if an out-of-sync replica receives a fetch at an offset which is larger than the last high watermark value received from the leader. The replica has no choice in this case but to return OFFSET_OUT_OF_RANGE. In general, when fetching from followers, out of range errors cannot be taken as definitive. Before raising an error, the consumer must validate whether the offset is correct.

In fact, the logic to handle this case is already outlined in KIP-320. The consumer will reconcile its current offset with the leader and then will resume fetching. This gives us a chance to detect any ISR changes before we resume fetching from the same replica.

It may not be immediately obvious that the OffsetsForLeaderEpoch can be used to determine if an offset is still valid. The consumer will request from the leader the end offset for the epoch corresponding to its current fetch offset. If the fetch offset has been truncated in an unclean leader election, then the OffsetsForLeaderEpoch response will return either 1) the same epoch with an end offset smaller than the fetch offset, or 2) a smaller epoch and a smaller fetch offset. In either case, the returned end offset would be smaller than the fetch offset. Otherwise, if the offset exists, then the response will include the same epoch and an end offset which is larger than the fetch offset.

So to handle the OFFSET_OUT_OF_RANGE error code, the consumer will take the following steps:

1. Use the OffsetForLeaderEpoch API to verify the current position with the leader.
    a. If the fetch offset is still valid, refresh metadata and continue fetching
    b. If truncation was detected, follow the steps in KIP-320 to either reset the offset or raise the truncation error
    c. Otherwise, follow the same steps above as in case 3.

If we do not have an epoch corresponding to the current fetch offset, then we will skip the truncation check and use the same handling that we used for the FETCH_OFFSET_TOO_SMALL case (basically we resort to the reset policy). In practice, out of range errors due to ISR propagation delays should be extremely rare because it requires a consumer to see a committed offset before a follower that is still considered in the ISR.

TLDR: The only changes here are the following:

1. The broker will return OFFSET_NOT_AVAILABLE if an offset is known to exist, but is either not available or not known to be committed.
2. For out of range errors, we will skip the offset truncation check from KIP-320 if the fetch offset is smaller than the log start offset.
3. When resetting to the earliest offset, we need to query the replica that we will be fetching from.

Finding the preferred follower

The problem for a consumer is figuring out which replica is preferred. The two options are to either let the consumer find the preferred replica itself using metadata from the brokers (i.e. rackId, host information), or to let the broker decide the preferred replica based on information from the client. We propose here to do the latter.

The benefit of letting the broker decide which replica is preferred for a client is that it can take load into account. For example, the broker can round robin between nearby replicas in order to spread the load. There are many such considerations that are possible, so we we propose to allow users to provide a ReplicaSelector plugin to the broker in order to handle the logic. This will be exposed to the broker with the `replica.selector.class` configuration. In order to make use of this plugin, we will extend the Fetch API so that clients can provide their own location information. In the response, the broker will indicate a preferred replica to fetch from.

Public Interfaces

Consumer API

We use of the "broker.rack" configuration to identify the location of the broker, so we will add a similar configuration to the consumer. This will be used in Fetch requests as documented below.

In this proposal, the consumer will always fetch from the preferred replica returned by the Metadata request regardless whether a `rack.id` is provided.

Configuration Name	Valid Values	Default Value
client.rack	<nullable string>	null

Broker API

We will expose a new plugin interface configured through the "replica.selector.class" configuration that allows users to provide custom logic to determine the preferred replica to fetch from.

Configuration Name	Valid Values	Default Value
replica.selector.class	class name of ReplicaSelector implementation	LeaderSelector

By default, Kafka will use an implementation which always returns the current partition leader (if one exists). This is for backwards compatibility.

The ReplicaSelector interface is provided below. We will pass client metadata include the rackId passed through the Metadata API and other connection-related information.

```
interface ClientMetadata {
  String rackId();
  String clientId();
  InetAddress clientAddress();
  KafkaPrincipal principal();
  String listenerName();
}

interface ReplicaView {
  Node endpoint();
  long logEndOffset();
  long timeSinceLastCaughtUpMs();
}

interface PartitionView {
  Set<ReplicaView> replicas();
  Optional<ReplicaView> leader();
}

interface ReplicaSelector extends Configurable, Closeable {
    /**
     * Select the preferred replica a client should use for fetching.
     * If no replica is available, this method should return an empty optional.
         */
        Optional<ReplicaView> select(TopicPartition topicPartition, ClientMetadata metadata, PartitionView
partitionView);

    /**
     * Optional custom configuration
     */
    default void configure(Map<String, ?> configs) {}

    /**
     * Optional shutdown hook
     */
        default void close() {}
}
```

We will provide two implementations out of the box. One of these is the LeaderSelector mentioned above.

```
class LeaderSelector implements ReplicaSelector {

  Optional<ReplicaView> select(TopicPartition topicPartition, ClientMetadata metadata, PartitionView
partitionView) {
    return partitionView.leader();
  }
}
```

The second uses the rackId provided by the clients and implements exact matching on the rackId from replicas.

```
class RackAwareReplicaSelector implements ReplicaSelector {

  Optional<ReplicaView> select(TopicPartition topicPartition, ClientMetadata metadata, PartitionView
partitionView) {
    // if rackId is not null, iterate through the online replicas
    // if one or more exists with matching rackId, choose the most caught-up replica from among them
    // otherwise return the current leader
  }
}
```

Protocol Changes

We will extend the Fetch API in order to enable selection of the preferred replica. We use of the "client.rack" configuration in the consumer that was mentioned above to identify the location of the broker, so we will add a similar configuration to the consumer.

```
{
  "validVersions": "0-11",
  "fields": [
    {
      "name": "ReplicaId",
      "type": "int32",
      "versions": "0+",
      "about": "The broker ID of the follower, of -1 if this request is from a consumer."
    },
    {
      "name": "MaxWait",
      "type": "int32",
      "versions": "0+",
      "about": "The maximum time in milliseconds to wait for the response."
    },
    {
      "name": "MinBytes",
      "type": "int32",
      "versions": "0+",
      "about": "The minimum bytes to accumulate in the response."
    },
    {
      "name": "MaxBytes",
      "type": "int32",
      "versions": "3+",
      "default": "0x7fffffff",
      "ignorable": true,
      "about": "The maximum bytes to fetch. See KIP-74 for cases where this limit may not be honored."
    },
    {
      "name": "IsolationLevel",
      "type": "int8",
      "versions": "4+",
      "default": "0",
      "ignorable": false,
      "about": "This setting controls the visibility of transactional records. Using READ_UNCOMMITTED
(isolation_level = 0) makes all records visible. With READ_COMMITTED (isolation_level = 1), non-transactional
```

```
and COMMITTED transactional records are visible. To be more concrete, READ_COMMITTED returns all data from
offsets smaller than the current LSO (last stable offset), and enables the inclusion of the list of aborted
transactions in the result, which allows consumers to discard ABORTED transactional records"
    },
    {
      "name": "SessionId",
      "type": "int32",
      "versions": "7+",
      "default": "0",
      "ignorable": false,
      "about": "The fetch session ID."
    },
    {
      "name": "Epoch",
      "type": "int32",
      "versions": "7+",
      "default": "-1",
      "ignorable": false,
      "about": "The fetch session ID."
    },
    {
      "name": "Topics",
      "type": "[]FetchableTopic",
      "versions": "0+",
      "about": "The topics to fetch.",
      "fields": [
        {
          "name": "Name",
          "type": "string",
          "versions": "0+",
          "entityType": "topicName",
          "about": "The name of the topic to fetch."
        },
        {
          "name": "FetchPartitions",
          "type": "[]FetchPartition",
          "versions": "0+",
          "about": "The partitions to fetch.",
          "fields": [
            {
              "name": "PartitionIndex",
              "type": "int32",
              "versions": "0+",
              "about": "The partition index."
            },
            {
              "name": "CurrentLeaderEpoch",
              "type": "int32",
              "versions": "9+",
              "default": "-1",
              "ignorable": true,
              "about": "The current leader epoch of the partition."
            },
            {
              "name": "FetchOffset",
              "type": "int64",
              "versions": "0+",
              "about": "The message offset."
            },
            {
              "name": "LogStartOffset",
              "type": "int64",
              "versions": "5+",
              "default": "-1",
              "ignorable": false,
              "about": "The earliest available offset of the follower replica. The field is only used when the
request is sent by the follower."
            },
            {
              "name": "MaxBytes",
              "type": "int32",
```

```
              "versions": "0+",
              "about": "The maximum bytes to fetch from this partition. See KIP-74 for cases where this limit
may not be honored."
            }
          ]
        }
      ]
    },
    {
      "name": "Forgotten",
      "type": "[]ForgottenTopic",
      "versions": "7+",
      "ignorable": false,
      "about": "In an incremental fetch request, the partitions to remove.",
      "fields": [
        {
          "name": "Name",
          "type": "string",
          "versions": "7+",
          "entityType": "topicName",
          "about": "The partition name."
        },
        {
          "name": "ForgottenPartitionIndexes",
          "type": "[]int32",
          "versions": "7+",
          "about": "The partitions indexes to forget."
        }
      ]
    },
    {
      "name": "RackId",
      "type": "string",
      "versions": "11+",
      "default": "",
      "ignorable": true,
      "about": "Rack ID of the consumer making this request"
    }
  ]
}
```

The fetch response will indicate the preferred replica.

```
{
  "apiKey": 1,
  "type": "response",
  "name": "FetchResponse",
  "validVersions": "0-11",
  "fields": [
    {
      "name": "ThrottleTimeMs",
      "type": "int32",
      "versions": "1+",
      "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota."
    },
    {
      "name": "ErrorCode",
      "type": "int16",
      "versions": "7+",
      "ignorable": false,
      "about": "The top level response error code."
    },
    {
      "name": "SessionId",
      "type": "int32",
      "versions": "7+",
```

```
      "default": "0",
      "ignorable": false,
      "about": "The fetch session ID, or 0 if this is not part of a fetch session."
    },
    {
      "name": "Topics",
      "type": "[]FetchableTopicResponse",
      "versions": "0+",
      "about": "The response topics.",
      "fields": [
        {
          "name": "Name",
          "type": "string",
          "versions": "0+",
          "entityType": "topicName",
          "about": "The topic name."
        },
        {
          "name": "Partitions",
          "type": "[]FetchablePartitionResponse",
          "versions": "0+",
          "about": "The topic partitions.",
          "fields": [
            {
              "name": "PartitionIndex",
              "type": "int32",
              "versions": "0+",
              "about": "The partiiton index."
            },
            {
              "name": "ErrorCode",
              "type": "int16",
              "versions": "0+",
              "about": "The error code, or 0 if there was no fetch error."
            },
            {
              "name": "HighWatermark",
              "type": "int64",
              "versions": "0+",
              "about": "The current high water mark."
            },
            {
              "name": "LastStableOffset",
              "type": "int64",
              "versions": "4+",
              "default": "-1",
              "ignorable": true,
              "about": "The last stable offset (or LSO) of the partition. This is the last offset such that the
state of all transactional records prior to this offset have been decided (ABORTED or COMMITTED)"
            },
            {
              "name": "LogStartOffset",
              "type": "int64",
              "versions": "5+",
              "default": "-1",
              "ignorable": true,
              "about": "The current log start offset."
            },
            {
              "name": "Aborted",
              "type": "[]AbortedTransaction",
              "versions": "4+",
              "nullableVersions": "4+",
              "ignorable": false,
              "about": "The aborted transactions.",
              "fields": [
                {
                  "name": "ProducerId",
                  "type": "int64",
                  "versions": "4+",
                  "entityType": "producerId",
```

```
                "about": "The producer id associated with the aborted transaction."
              },
              {
                "name": "FirstOffset",
                "type": "int64",
                "versions": "4+",
                "about": "The first offset in the aborted transaction."
              }
            ]
          },
          {
            "name": "PreferredReadReplica",
            "type": "int32",
            "versions": "11+",
            "ignorable": true,
            "about": "The preferred read replica for the consumer to use on its next fetch request"
          },
          {
            "name": "Records",
            "type": "bytes",
            "versions": "0+",
            "nullableVersions": "0+",
            "about": "The record data."
          }
        ]
      }
    ]
  }
]
}
```

The FetchRequest schema has field for the replica id. Consumers typically use the sentinel -1, which indicates that fetching is only allowed from the leader. A lesser known sentinel is -2, which was originally intended to be used for debugging and allows fetching from followers. We propose to let the consumer use this to indicate the intent to allow fetching from a follower. Similarly, when we need to send a ListOffsets request to a follower in order to find the log start offset, we will use the same sentinel for the replica id field.

It is still necessary, however, to bump the version of the Fetch request because of the use of the OFFSET_NOT_AVAILABLE error code mentioned above. The request and response schemas will not change. We will also modify the behavior of all fetch versions so that the current log start offset and high watermark are returned if the fetch offset is out of range. This allows the broker to distinguish out of range errors.

One point that was missed in KIP-320 is the fact that the OffsetsForLeaderEpoch API exposes the log end offset. For consumers, the log end offset should be the high watermark, so we need a way for this protocol to distinguish replica and consumer requests. We propose here to add the same `replica_id` field that is used in the Fetch and ListOffsets APIs. The new schema is provided below:

```
OffsetsForLeaderEpochRequest => [Topic]
  ReplicaId => INT32              // New (-1 means consumer)
  Topic => TopicName [Partition]
    TopicName => STRING
    Partition => PartitionId CurrentLeaderEpoch LeaderEpoch
      PartitionId => INT32
      CurrentLeaderEpoch => INT32
      LeaderEpoch => INT32
```

When an OffsetsForLeaderEpoch request is received from a consumer, the returned offset will be limited to the high watermark. As with the Fetch API, when a leader has just been elected, the true high watermark is not known for a short time. If an OffsetsForLeaderEpoch request is received with the latest epoch during this window, the leader will respond with the OFFSET_NOT_AVAILABLE error code. This will cause the consumer to retry. Note as well that only leaders are allowed to handle OffsetsForLeaderEpoch queries.

A new JMX metric will be added to the Java client when it has a preferred read replica. Under the "kafka.consumer:type=consumer-fetch-manager-metrics, client-id=*,topic=*,partition=*" object, a new attribute "preferred-read-replica" will be added. This metric will have a value equal to the broker ID which the consumer is currently fetching from. If the attribute is missing or set to -1, it means the consumer is fetching from the leader.

Compatibility, Deprecation, and Migration Plan

This change is backwards compatible with previous versions. If the broker does not support follower fetching, the consumer will revert to the old behavior of fetching from the leader.

Rejected Alternatives

- **Consumer Replica Selection**: The first iteration of this KIP proposed to let the consumer decide which replica to fetch from using an interface similar to the one we have above. Ultimately we felt it was easier to reason about the system in a multi-tenant environment when replica selection logic could be controlled by the broker. It is much more difficult to have common replica selection logic through a client plugin since it involves coordinating dependencies and configuration across many applications. For the same reason, we also rejected the option to specify whether the preferred replica should be used by the client.
- **Handling Old Fetch Versions**: We have considered allowing the consumer to use older versions of the Fetch API. The only complication is that older versions will not allow us to use the more specific out of range error codes. This complicates out of range handling because we do not know whether we should expect the out of range offset to exist on the leader or not. If the offset is too small on the follower, it may still be in range on the leader. One option is to always do offset reconciliation on the replica that the consumer is fetching from. The downside of this is that we cannot detect situations when the consumer has seen a later offset than a replica due to stale propagation of the ISR state.
- **Improved Log Start Offset Handling**: We have mentioned the difficulty of reasoning about the consistency of the log start offset due to the fact that the retention is enforced independently by all replicas. One option we have considered is to only allow the leader to enforce retention and to rely only on the propagation of the log start offset in the Fetch response for follower deletion. In principle, this would give us the guarantee that the log start offset of a follower should always be less than or equal to that of the leader. However, unless we make log start offset acknowledgement a criteria for ISR membership, we cannot actually guarantee this. In the case of multiple leader elections occurring in a short time period, it is possible for this guarantee to be violated. Nevertheless, we think it may still be a good idea to rely on log start offset propagation for follower deletion, but we will consider this separately.