KIP-227: Introduce Incremental FetchRequests to Increase Partition Scalability

Status

Current state:Accepted

Discussion thread: https://www.mail-archive.com/dev@kafka.apache.org/msg83115.html

JIRA*:* KAFKA-6254

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

- Status
- Motivation
- Proposed Changes
    - Fetch Sessions
        - Fetch Session ID
        - Fetch Epoch
        - Cached data about each partition
        - Privileged bit
        - The time when the fetch session was last used
    - Fetch Session Caching
- Public Interface Changes
    - New Error Codes
    - FetchRequest Changes
        - Fetch Session ID
        - Fetch Session Epoch
        - FetchRequest Metadata meaning
        - Incremental Fetch Requests
        - Schema
    - FetchResponse Changes
        - Top-level error code
        - Fetch Session ID
        - FetchResponse Metadata meaning
        - Incremental Fetch Responses
        - Handling Partition Size Limits in Incremental Fetch Responses
        - Schema
- New Metrics
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives

Motivation

Apache Kafka Brokers make periodic *FetchRequests* to other brokers, in order to learn about updates to partitions they are following. These periodic *Fetch Requests* must enumerate all the partitions which the follower is interested in. The responses also enumerate all the partitions, plus metadata (and potentially data) about each one.

The frequency at which the leader responds to the follower's fetch requests is controlled by several configuration tunables, including *replica.fetch.wait.max.ms, replica.fetch.min.bytes, replica.fetch.max.bytes,* and *replica.fetch.response.max.bytes.* Broadly speaking, the system can be tuned for lower latency, by having more frequent, smaller fetch responses, or for reduced system load, having having fewer, larger fetch responses.

There are two major inefficiencies in the current *FetchRequest* paradigm. The first one is that the set of partitions which the follower is interested in changes only rarely. Yet each *FetchRequest* must enumerate the full set of partitions which the follower is interested in. The second inefficiency is that even when nothing has changed in a partition since the previous *FetchRequest*, we must still send back metadata about that partition.

These inefficiencies are linearly proportional to the number of extant partitions in the system. So, for example, imagine a Kafka installation with 100,000 partitions, most of which receive new messages only rarely. The brokers in that system will still send back and forth extremely large *FetchRequests* and *FetchResponses*, even though there is very little actual message data being added per second. As the number of partitions grows, Kafka uses more and more network bandwidth to pass back and forth these messages.

When Kafka is tuned for lower latency, these inefficiencies become worse. If we double the number of *FetchRequests* sent per second, we should expect there to be more partitions which haven't changed within the reduced polling interval. And we are less able to amortize the high fixed cost of sending metadata for each partition in every *FetchRequest* and *FetchResponse*. This again results in Kafka using more of the available network bandwidth.

Proposed Changes

We can solve the scalability and latency problems discussed above by creating "incremental" fetch requests and responses that only include information about what has changed. In order to do this, we need to introduce the concept of "fetch sessions."

Fetch Sessions

A fetch session encapsulates the state of an individual fetcher. This allows us to avoid resending this state as part of each fetch request.

The Fetch Session includes:

1. A randomly generated 32-bit session ID which is unique on the leader
2. The 32-bit fetch epoch
3. Cached data about each partition which the fetcher is interested in.
4. The privileged bit
5. The time when the fetch session was last used

Fetch Session ID

The fetch session ID is a randomly generated 32-bit session ID. It is a unique, immutable identifier for the fetch session. Note that the fetch session ID may not be globally unique (although it's very likely to be so.) It simply has to be unique on the leader.

Since the ID is randomly chosen, it cannot leak information to unprivileged clients. It is also very hard for a malicious client to guess the fetch session ID. (Of course, there are other defenses in place against malicious clients, but the randomness of the ID provides defense in depth.)

Fetch Epoch

The fetch epoch is a monotonically incrementing 32-bit counter. After processing request N, the broker expects to receive request N+1.

The sequence number is always greater than 0. After reaching MAX_INT, it wraps around to 1.

Cached data about each partition

If the fetch session supports incremental fetches, the FetchSession will maintain information about each partition in the incremental fetch.

For each partition, we maintain:

- The topic name
- The partition index
- The maximum number of bytes to fetch from this partition
- The fetch offset
- The high water mark
- The fetcher log start offset
- The leader log start offset

Topic name and partition index come from the TopicPartition.

maxBytes, fetchOffset, and fetcherLogStartOffset come from the latest FetchRequest in which the partition appeared.

highWatermark and localLogStartOffset come from the leader.

The leader uses this cached information to decide which partitions to include in the FetchResponse. Whenever any of these elements change, or if there is new data available for a partition, the partition will be included.

Privileged bit

The privileged bit is set if the fetch session was created by a follower. It is cleared if the fetch session was created by a regular consumer.

This is retained in order to prioritize followers over consumers, when resources are low. See the section on fetch session caching for details.

The time when the fetch session was last used

This is the time in wall-clock milliseconds when the fetch session was last used. This is used to expire fetch sessions after they have been inactive. See the section on fetch session caching for details.

Fetch Session Caching

Because fetch sessions use memory on the leader, we want to limit the amount of them that we have at any given time. Therefore, each broker will create only a limited number of incremental fetch sessions.

There is one new public configurations for fetch session caching:

- **max.incremental.fetch.session.cache.slots**, which set the number of incremental fetch session cache slots on the broker. Default value: 1,000

There is one new constant for fetch session caching:

- **min.incremental.fetch.session.eviction.ms**, which sets the minimum amount of time we will wait before evicting an incremental fetch session from the cache. Value: 120,000

When the server gets a new request to create an incremental fetch session, it will compare the proposed new session with its existing sessions. The new session will evict an existing session if and only if:

1. The new session belongs to a follower, and the existing session belongs to a regular consumer, OR
2. The existing session has been inactive for more than **min.incremental.fetch.session.eviction.ms**, OR
3. The existing session has existed for more than **min.incremental.fetch.session.eviction.ms**, AND the new session has more partitions

This accomplishes a few different goals:

- Followers get priority over consumers
- Inactive session get replaced over time
- Bigger requests, which benefit more from being incremental, are prioritized
- Cache thrashing is limited, avoiding expensive session re-establishment when there are more fetchers than cache slots.

Public Interface Changes

New Error Codes

FetchSessionIdNotFound: The server responds with this error code when the client request refers to a fetch session that the server does not know about. This may happen if there was a client error, or if the fetch session was evicted by the server.

InvalidFetchSessionEpochException. The server responds with this error code when the fetch session epoch of a request is different than what it expected.

FetchRequest Changes

There are several changes to the FetchRequest API.

Fetch Session ID

A 32-bit number which identifies the current fetch session. If this is set to 0, there is no current fetch session.

Fetch Session Epoch

A 32-bit number which identifies the current fetch session epoch. Valid session epochs are always positive-- they are never 0 or negative numbers.

The fetch session epoch is incremented by one for each fetch request that we send. Once it reaches MAX_INT, the next epoch is 1.

The fetch epoch keeps the state on the leader and the follower synchronized. It ensures that if a message is duplicated or lost, the server will always notice. It is also used to associate requests and responses in the logs. Other numbers, such as IP addresses and ports, or NetworkClient sequence numbers, can also be helpful for this purpose-- but they are less likely to be unique.

FetchRequest Metadata meaning

Request SessionId	**Request SessionEpoch**	**Meaning**
0	-1	Make a full FetchRequest that does not use or create a session. This is the session ID used by pre-KIP-227 FetchRequests.
0	0	Make a full FetchRequest. Create a new incremental fetch session if possible. If a new fetch session is created, it will start at epoch 1.
$ID	0	Close the incremental fetch session identified by $ID. Make a full FetchRequest. Create a new incremental fetch session if possible. If a new fetch session is created, it will start at epoch 1.
$ID	$EPOCH	If the ID and EPOCH are correct, make an incremental fetch request.
$ID	-1	Close the incremental fetch session identified by $ID. Make a full FetchRequest.

Incremental Fetch Requests

Incremental fetch requests have a positive fetch session ID.

A partition is only included in an incremental FetchRequest if:

- The client wants to notify the broker about a change to the partition's maxBytes, fetchOffset, or logStart
- The partition was not included in the incremental fetch session before, but the client wants to add it.
- The partition is in the incremental fetch session, but the client wants to remove it.

If the client doesn't want to change anything, the client does not need to include any partitions in the request at all.

If the client wants to remove a partition, the client will add the partition to the removed_partition list in the relevant removed_topics entry.

Schema

FetchRequest => max_wait_time replica_id min_bytes isolation_level fetch_session_id fetch_session_epoch [topic] [removed_topic]

max_wait_time => INT32

replica_id => INT32

min_bytes => INT32

isolation_level => INT8

fetch_session_id => INT32

fetch_session_epoch => INT32

topic => topic_name [partition]

topic_name => STRING

partition => partition_id fetch_offset start_offset max_bytes

partition_id => INT32

fetch_offset => INT64

start_offset => INT64

max_bytes => INT32

removed_topic => removed_topic_name [removed_partition_id]

removed_topic_name => STRING

removed_partition_id => INT32

FetchResponse Changes

Top-level error code

Per-partition error codes are no longer sufficient to handle all response errors. For example, when an incremental fetch session encounters an FetchSessionIdNotFoundException, we do not know which partitions the client expected to fetch. Therefore, the FetchResponse now contains a top-level error code. This error code is set to indicate that the request as a whole cannot be processed.

When the top-level error code is set, the caller should assume that all the partitions in the fetch received the given error.

Fetch Session ID

The FetchResponse now contains a 32-bit fetch session ID.

FetchResponse Metadata meaning

Request SessionId	**Meaning**
0	No fetch session was created.

$ID	The next request can be an incremental fetch request with the given $ID.

Incremental Fetch Responses

A partition is only included in an incremental FetchResponse if:

- The broker wants to notify the client about a change to the partition's highWatermark or broker logStartOffset
- There is new data available for a partition

If the broker has no new information to report, it does not need to include any partitions in the response at all.

The format of the partition data within FetchResponse is unchanged.

Handling Partition Size Limits in Incremental Fetch Responses

Sometimes, the per-fetch-request limit is too small to allow us to return information about every partition. In those cases, we will limit the number of partitions that we return information about, to avoid exceeding the per-request maximum. (As specified in KIP-74, the response will always return at least one message, though.)

If we always returned partition information in the same order, we might end up "starving" the partitions which came near the end of the order. With full fetch requests, the client can rotate the order in which it requests partitions in order to avoid this problem. However, incremental fetch requests need not explicitly specify all the partitions. Indeed, an incremental fetch request may contain no partitions at all.

In order to solve the starvation problem, the server must rotate the order in which it returns partition information. The server does this by maintaining a linked list of all partitions in the fetch session. When data is returned for a partition, that partition is moved to the end of the list. This ensures that we eventually return data about all partitions for which data is available.

Schema

```
FetchResponse => throttle_time_ms error_code error_string fetch_session_id [topic]

  throttle_time_ms => INT32

  error_code => INT16

  fetch_session_id => INT32

  topic => topic_name [partition]

  topic_name => STRING

  partition => partition_header records

  partition_header => partition_id error_code high_watermark last_stable_offset log_start_offset [aborted_transaction]

  partition_id => INT32

  error_code => INT16

  high_watermark => INT64

  last_stable_offset => INT64

  log_start_offset => INT64

  aborted_transaction => producer_key first_offset

  producer_key => INT64

  first_offset => INT64

  records => RECORDS
```

New Metrics

The following new metrics will be added to track cache consumption:

NumIncrementalFetchSessions: Tracks the number of incremental fetch sessions which exist.

NumIncrementalFetchPartitionsCached: Tracks the total number of partitions cached by incremental fetch sessions.

IncrementalFetchSessionEvictionsPerSec: Tracks the number of incremental fetch sessions that were evicted from the cache per second. This metric is not increased when a client closes its own incremental fetch session

Compatibility, Deprecation, and Migration Plan

Although this change adds the concept of incremental fetch requests, existing brokers can continue to use the old *FetchRequest* as before. Therefore, there is no compatibility impact.

Rejected Alternatives

We considered several other approaches to minimizing the inefficiency of *FetchRequests* and *FetchResponses*.

- Reduce the number of temporary objects created during serialization and deserialization
- Assign each topic a unique ID, so that topic name strings do not have to be sent over the wire

Changing the way Kafka does serialization and deserialization would be a huge change, requiring rewrites of all the Request and Response classes. We could potentially gain some efficiency by getting rid of the Builder classes for messages, but we would have to find a way to solve the problems which the Builder classes currently solve, such as the delayed binding between message content and message version and format. Micro-optimizing the serialization and deserialization path might also involve making tradeoffs that decrease perceived code quality. For example, we might have to sometimes give up immutability, or use primitive types instead of class types.

Assigning unique IDs to topics is also a very big project, requiring cluster-wide coordination, many RPC format changes, and potentially on-disk format changes as well.

More fundamentally, while both of these approaches improve the constant factors associated with *FetchRequest* and *FetchResponse*, they do not change the computational complexity. In both cases, the complexity would remain **O(num_extant_partitions)**. In contrast, the approach above makes the messages scale as **O(num_partition_changes)**. Adding *IncrementalFetchRequest* and *IncrementalFetchResponse* is a smaller and more effective change.