KIP-33 - Add a time based log index

- Status
- Motivation
- Public Interfaces
- Proposed Changes
    - Add a new time-based log index
        - Time index format
        - Build the time index
        - Broker startup
        - Log Truncation
    - Enforce time based log retention
    - Enforce time based log rolling
    - Search message by timestamp
- Use case discussion
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives
    - Add a timestamp field to log index entry
    - Option 1 - Time based index using LogAppendTime
        - Use a time index for each log segment to save the timestamp -> log offset at minute granularity
    - Option 2 - Time based index using CreateTime

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *KAFKA-3163*

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

Motivation

Kafka has a few timestamp based functions, including

1. Searching message by timestamp
2. Time based log rolling
3. Time based log retention.

Currently these operations depend on the create time / modification time of the log segment file. This has a few issues.

1. Searching offset by timestamp has very coarse granularity (log segment level), it also does not work well when replica is reassigned.
2. The time based log rolling and retention does not work well when replica is reassigned.

In this KIP we propose adding a time based log index using the timestamp of the messages introduced in KIP-32.

Public Interfaces

There will be some behavioral changes to time based log retention and log rolling.

1. The log retention will be changed to base on the time index of a log segment instead of basing on the last modification time of the log segment file.
2. The time based log rolling will have the following change: The log segment will roll out when log.roll.ms has elapsed since the largest timestamp of the messages in the log segment.

This KIP will introduce another index file for each log segment. So the number of open file handlers will increase by about 1/3.

Proposed Changes

Add a new time-based log index

Broker will build time index based on the timestamp of the messages. The log index works for both LogAppendTime and CreateTime.

Time index format

Create another index file for each log segment with name **SegmentBaseOffset.timeindex.** The density of the index is upper bounded by **index.interval.bytes** configuration.

The time index entry format is:

```
Time Index Entry => Timestamp Offset
  Timestamp => int64
  Offset => int32
```

- Timestamp - the biggest timestamp seen so far in this segment.
- Offset - the next offset when the time index entry is inserted.
- A time index entry (*T*, *offset*) means that in this segment any message whose timestamp is greater than *T* come after *offset.*

Build the time index

Based on the proposal in KIP-32, the broker will build the time index in the following way:

1. When a new log segment is created, the broker will create a time index file for the log segment.
2. The default initial / max size of the time index files is the same as the offset index files. (time index entry is 1.5x of the size of offset index entry, user should set the configuration accordingly).
3. Each log segment maintains the largest timestamp so far in that segment. The initial value of the largest timestamp is -1 for a newly created segment.
4. When broker receives a message, if the message is not rejected due to timestamp exceeds threshold, the message will be appended to the log. (The timestamp will either be LogAppendTime or CreateTime depending on the configuration)
5. When broker appends the message to the log segment, if an offset index entry is inserted, it will also insert a time index entry if the max timestamp so far is greater than the timestamp in the last time index entry.
    - For message format v0, the timestamp is always -1, so no time index entry will be inserted when message is appended.
6. When the current active segment is rolled out or closed. A time index entry will be inserted into the time index to ensure the last time index entry has the largest timestamp of the log segment.
    a. If largest timestamp in the segment is non-negative (at least one message has a timestamp), the entry will be **(largest_timestamp_in_the_segment -> base_offset_of_the_next_segment)**
    b. If largest timestamp in the segment is -1 (No message in the segment has a timestamp), the time index will be empty and the largest timestamp would be default to the segment last modification time.

The time index is not monotonically increasing for the segments of a partition. Instead, it is only monotonically increasing within each individual time index file. i.e. It is possible that the time index file for a later log segment contains smaller timestamp than some timestamp in the time index file of an earlier segment.

Broker startup

On broker startup, the latest timestamp is needed for the next log index append. The broker will find the largest timestamp by looking at the last time index entry. The last time index entry has the largest timestamp if the broker is shutdown normally before. Broker will only load the largest timestamp if **message.format.version** is on or after 0.10.0. Otherwise the broker will skip loading the largest timestamp. On a recovery after hard failure, the broker will scan the active log segment till the log end.

If there is no time index file for an inactive segment, the broker will create an empty time index file and append a time index entry **(last_modification_time_of_the_segment -> base_offset_of_the_next_segment).** For the active segment, the broker will create the time index file but leave it empty.

Log Truncation

When the log is truncated, because the offset in the time index is also monotonically increasing, we will also truncate the time index entries whose corresponding messages have been truncated. The active segment will reload the largest timestamp after truncation by scanning the messages from the offset of the last time index entry till the log end.

Enforce time based log retention

To enforce time based log retention, the broker will check from the oldest segment forward to the latest segment. For each segment, the broker checks the last time index entry of a log segment. The timestamp will be the latest timestamp of the messages in the log segment. So if that timestamp expires, the broker will delete the log segment. The broker will stop at the first segment which is not expired. i.e. the broker will not expire a segment even if it is expired, unless all the older segment has been expired.

Enforce time based log rolling

Currently time based log rolling is based on the creating time of the log segment. With this KIP, the time based rolling would be changed to only based on the message timestamp. More specifically, if the first message in the log segment has a timestamp, A new log segment will be rolled out if timestamp in the message about to be appended is greater than the timestamp of the first message in the segment + log.roll.ms. When message.timestamp.type=CreateTime, user should set max.message.time.difference.ms appropriately together with log.roll.ms to avoid frequent log segment roll out.

During the migration phase, if the first message in a segment does not have a timestamp, the log rolling will still be based on the (current time - create time of the segment).

Search message by timestamp

When searching by timestamp, broker will start from the earliest log segment and check the last time index entry. If the timestamp of the last time index entry is greater than the target timestamp, the broker will do binary search on that time index to find the closest index entry and scan the log from there. Otherwise it will move on to the next log segment.

Searching by timestamp will have better accuracy. The guarantees provided are:

- The messages whose timestamp are after the searched timestamp will be consumed.
- Some messages with earlier timestamp might also be consumed.

The OffsetRequest behaves almost the same as before. If timestamp **T** is set in the OffsetRequest, the first offset in the returned offset sequence means that if user want to consume from **T**, that is the offset to start with. The guarantee is that any message whose timestamp is greater than T has a bigger offset. i.e. Any message before this offset has a timestamp < **T**.

The time index granularity does not change the actual timestamp searching granularity. It only affects the time needed for searching.

Use case discussion

	Use case	Goal	Index based on LogAppendTime	Index based on CreateTime	Comparison
1	**Search by timestamp**	Not lose messages	If user want to search for a message with CreateTime **CT**. They can use **CT** to search in the LogAppendTime index. Because **LogAppendTime** > **CT** for the same message (assuming no skew clock). If the clock is skewed, people can search with **CT - X** where X is the max skew. If user want to search for a message with LogAppendTime **LAT**, they can just search with **LAT** and get a millisecond accuracy.	User can just search with **CT** and get a precise offset.	User may see duplicates when searching by CreateTime. Consider the following case: 1. A message m1 with CreateTime **CT** arrives broker at **LAT1**. 2. Some time later at **LAT2**, another message m2 with CreateTime **CT** arrives at broker. If user want to search with CT after they consumed m2, they will have to reconsume from m1. Depending on how big LAT2 - LAT1 is, the amount of messages to be reconsumed can be very big.
2	**Search by timestamp (bootstrap)**	1. Not lose messages 2. Consume less duplicate messages	In bootstrap case, all the LAT would be close. For example If user want to process the data in last 3 days and did the following: 1. Dump a big database into Kafka 2. Reprocess the message in last 3 days. In this case, LogAppendTime index does not help too much. That means user needs to filter out the data older than 3 days before dumping them into Kafka.	In bootstrap case, the CreateTime will not change, if user follow the same procedure started in LogAppendTime index section. Searching by timestamp will work.	LogAppendTime index needs further attention from user.
3	**Failover from cluster 1 to cluster 2**	1. Not lose messages 2. Consume less duplicate messages	Similar search by timestamp. User can choose to use **CT** or **LAT** of cluster 1 to search on cluster 2. In this case, searching with **CT - MaxLatencyOfCluster** will provide strong guarantee on not losing messages, but might have some duplicates depending on the difference in latency between cluster 1 and cluster 2.	User can use **CT** to search and get precise offset. Duplicates are still not avoidable. There can be some tricky cases here. Consider the following case **[1]**: - **m1** with **CT1** and **m2** with **CT2** are both produced to cluster 1 and cluster 2. - **m1** is created earlier than **m2**. i.e. **CT1 < CT2** - **m1** arrives cluster 1 at **LAT11** and arrives cluster 2 at **LAT12**, assuming **LAT11 < LAT12** - **m2** arrives cluster 2 at **LAT21** and arrives cluster 2 at **LAT22**, assuming **LAT12 > LAT22** In this case, **m1** is created before **m2**. Due to latency difference, **m1** arrives cluster 1 then **m2** does, **m2** arrives cluster 2 before **m1** does. If a consumer consumed **m2** in cluster 2 and fail over to cluster 1, simply search by **CT2** will miss **m1** because m1 has larger offset than **m2** in cluster 2 but smaller offset than **m2** in cluster 1. So the same trick or **CT - MaxLatencyOfCluster** is still needed.	In cross cluster fail over case, both solution can provide strong guarantee of not losing messages. But both needs to depend on the knowledge of **MaxLatencyOfCluster**.

4	**Get lag for consumers by time**	Know how long a consumer is lagging by time.	With LogAppendTime in the message, consumer can easily find out the lag by time and estimate how long it might need to reach the log end.	Not supported.	
5	**Broker side latency metric**	Let the broker to report latency of each topic. i.e. LAT - CT	The latency can be reported as LAT - CT.	The latency can be reported as System. currentTimeMillis - CT	The two solutions are the same. This latency information can be used for **MaxLatencyOfCluster** in use case 3.

From the use cases list above, generally having a LogAppendTime index is better than having a CreateTime based timestamp.

Compatibility, Deprecation, and Migration Plan

The change is backward compatible after KIP-31 and KIP-32 are checked in.

User may want to bump up the log.index.size.max.bytes to 1.5x because the time index may take up to 1.5x of the offset index. We will change the index size to appropriate value (3 MB for 1GB segment, 4KB index interval).

Broker will keep an in-memory **maxTimestampSoFar** variable, which is initialized to -1 and only gets updated when a message with a larger timestamp is appended to the log segment.

If maxTimestampSoFar is -1, log retention will still be based on last_modification_time. And log rolling will still be based on log create time.

Broker will do the following during migration:

1. The broker will create a time index for each segment if the segment does not have one.
    a. For the inactive log segments, the broker will append an entry **(last_modification_time_of_the_segment -> offset_of_the_first_message_in_the_segment)** to each empty time index.
    b. For the active log segments, the time index file will be left empty.
2. The broker will not rebuild the time index unless the broker had a hard failure during the previous shutdown.

After the entire cluster is migrated to use time based log index for log retention,

1. The broker will enforce log retention using time index. Given what we do in step 1, the behavior is
    a. For segments only having messages whose versions are before 0.10.0, the entry with last modification time in the time index will be used for retention.
    b. For segments having at least one message with version 0.10.0, the max timestamp of the messages will be used for log retention.
2. The broker will enforce the log rolling time when there is at least one message with format version 0.10.0 inserted, otherwise the log rolling is still based on segment create time.
3. Searching by timestamp will behave the same as before because if there is no message with timestamp, the time index will only contain one entry with last_modification_time. If there are messages with timestamp inserted, those messages will be indexed for search.

Rejected Alternatives

Add a timestamp field to log index entry

The most straight forward approach to have a time index is to let the log index files have a timestamp associate with each entry.

```
Log Index Entry => Offset Position Timestamp
  Offset => int32
  Position => int32
  Timestamp => int64
```

Because the index entry size become 16 bytes instead of 8 bytes. The index file size also needs to be doubled. As an example, one of the broker we have has ~3500 partitions. The index file took about 16GB memory. With this new format, the memory consumption would be 32GB.

Option 1 - Time based index using LogAppendTime

In order to enable timestamp based search at finer granularity, we need to add the timestamp to log indices as well. Broker will build time index based on LogAppendTime of messages.

Because all the index files are memory mapped files the main consideration here is to avoid significantly increasing the memory consumption.

The time index file needs to be built just like the log index file based on each log segment file.

Use a time index for each log segment to save the timestamp -> log offset at minute granularity

Create another index file for each log segment with name **SegmentBaseOffset.time.index** to have index at minute level. The time index entry format is:

```
Time Index Entry => Timestamp Offset
  Timestamp => int64
  Offset => int32
```

The time index granularity does not change the actual timestamp searching granularity. It only affects the time needed for searching. The way it works will be the same as offset search - find the closet timestamp and corresponding offset, then start the leaner scan over the log until find the target message. The reason we prefer minute level indexing is because timestamp based search is usually rare so it probably does not worth investing significant amount of memory in it.

The time index will be built based on the log index file. Every time when a new entry is inserted into log index file, we take a look at the timestamp of the message and if it falls into next minute, we insert an entry to the time index as well. The following table give the summary of memory consumption using different granularity. The number is calculated based on a broker with 3500 partitions.

second	86400	3.4 GB
Minute	1440	57 MB

Users don't typically need to look up offsets with seconds granularity.

Option 2 - Time based index using CreateTime

Another option is to build index based on CreateTime of messages. Similar to option 1, we are going to have one time index file per log segment.

The biggest challenge of indexing using CreateTime is that CreateTime can be out of order.

One solution is as below:

1. Each broker keeps in memory a timestamp index map - Map[TopicPartitionSegment, Map[TimestampByMinute, Offset]]
    a. The timestamp is on minute boundary
    b. The offset is the offset of the first message in the log segment that falls into a minute
2. Create a timestamp index file for each log segment. The entry in the file is in following format:

    ```
    Time Index Entry => Timestamp Offset
      Timestamp => int64
      Offset => int32
    ```

    So the timestamp index file will simply become a persistent copy of timestamp index map. Broker will load the timestamp map from the file on startup.
3. When a broker (regardless leader or follower) receives a message, it does the following:
    a. Find which minute MIN the message with offset OFFSET falls in
    b. Check if MIN has already been in the in memory timestamp map for current log segment. If the timestamp does not exist, then the broker add [MIN->OFFSET] to both the in memory timestamp index map and the timestamp index file.
4. When a log segment is deleted, the broker:
    a. Remove the TopicPartitionSegment key from in memory map
    b. Remove the log segment timestamp index file