Chapter 14: Data Retention

Given that infinite storage is yet be invented, what happens to all those records stored in Kafka?

Chapter 3: Architecture and Core Concepts introduced the concept of low and high-water marks and how these shift as records are written to the log and as records are eventually purged. The focus on this chapter will be the latter; specifically, the conditions under which data is removed and the precise behaviour of Kafka in that regard.

Kafka storage internals

Given the option, most operators prefer to treat Kafka as a black box when configuring its data retention behaviour, and many other aspects, for that matter. By all means, this is understandable — being able to quickly consult the official documentation for a handful of configuration properties, apply them as stated in the instructions and move on to a less mundane task — almost sounds too reasonable to be true. And unfortunately, it is; Kafka is brimming with nuanced behaviour that is strongly coupled to the underlying implementation. Without the necessary insights, one may easily lose many days trying to explain the reasons that Kafka's behaves the way it does, seemingly in contradiction to the way it was configured.

Organisation of log data

Before exploring how Kafka's data retention mechanisms operate, it is instructive to gain a basic understanding of its log structure. As it has been stated numerous times, Kafka is a distributed, append-only log. The *distributed* aspect was covered in Chapter 13: Replication and Acknowledgements. Turning to the local persistence of a partition — which, as we know, occurs at each replica — the log is organised on disk as a series of data and index files.

The log files for each replicated partition are stored in a dedicated subdirectory of `log.dirs`, which points to `/tmp/kafka-logs` by default. The subdirectory is named by concatenating the topic name with the partition index, delimited by the `-` (hyphen) character. For example, partition `0` of the `getting-started` topic can be found at `/tmp/kafka-logs/getting-started-0`. Listing the contents of the directory with the `tree` command, we can see the following:

```
.
├── 00000000000000000000.index
├── 00000000000000000000.log
├── 00000000000000000000.timeindex
└── leader-epoch-checkpoint
```

The `leader-epoch-checkpoint` file contains the leader epoch — the number of times the leadership of a partition has been reassigned by the cluster controller. The replicas use the leader epoch as a means of verifying the current leader, fencing a prior leader if their leadership status has been superseded. If a displaced leader attempts to syndicate data, the replicas will sense that it is operating under an outdated epoch number and will ignore its commands.

Indexes

The files named with a large, zero-padded integer are *log segments*. Kafka does not store its partition log in a single, contiguous 'jumbo' file. Instead, logs are broken down into discrete chunks. Each segment is named in accordance with the offset of the first records that it contains. This is called the *base offset*.

Records may be of arbitrary size, and therefore occupy a variable amount of space in their respective `.log` file. To further complicate matters, records are persisted in batches (which might be compressed). Variable record size and batching mean that a record cannot be trivially located by its offset using the `.log` file alone. This is where the `.index` file comes in. Essentially, an index is a sorted map of record offsets to the physical locations of the corresponding records in the `.log` file. Where a record is encompassed within a batch, the index file points to the beginning of the batch entry. For compactness, the index file uses a 32-bit integer to store offsets, having subtracted the base offset from the record's offset beforehand. The physical locations pointed to by the index are also stored as 32-bit integers, for a combined entry size of 8 bytes. Indexes are memory-mapped files; the storage space for an index is preallocated when the index file is created. The amount of preallocated space is prescribed by the `log.index.size.max.bytes` property, which defaults to `10485760` (10 MiB) and acts as a default value for all topics. To control the extent of the preallocation for a specific topic, set the `segment.index.bytes` property. When a log file is rolled over, the remaining unused space in its index files is returned to the filesystem.

Kafka uses different names for properties that affect the default behaviour versus those that target individual topics. There is no particular naming convention to these properties; for example, the `log.roll.ms` cluster-wide property corresponds to the `segment.ms` per-topic property.

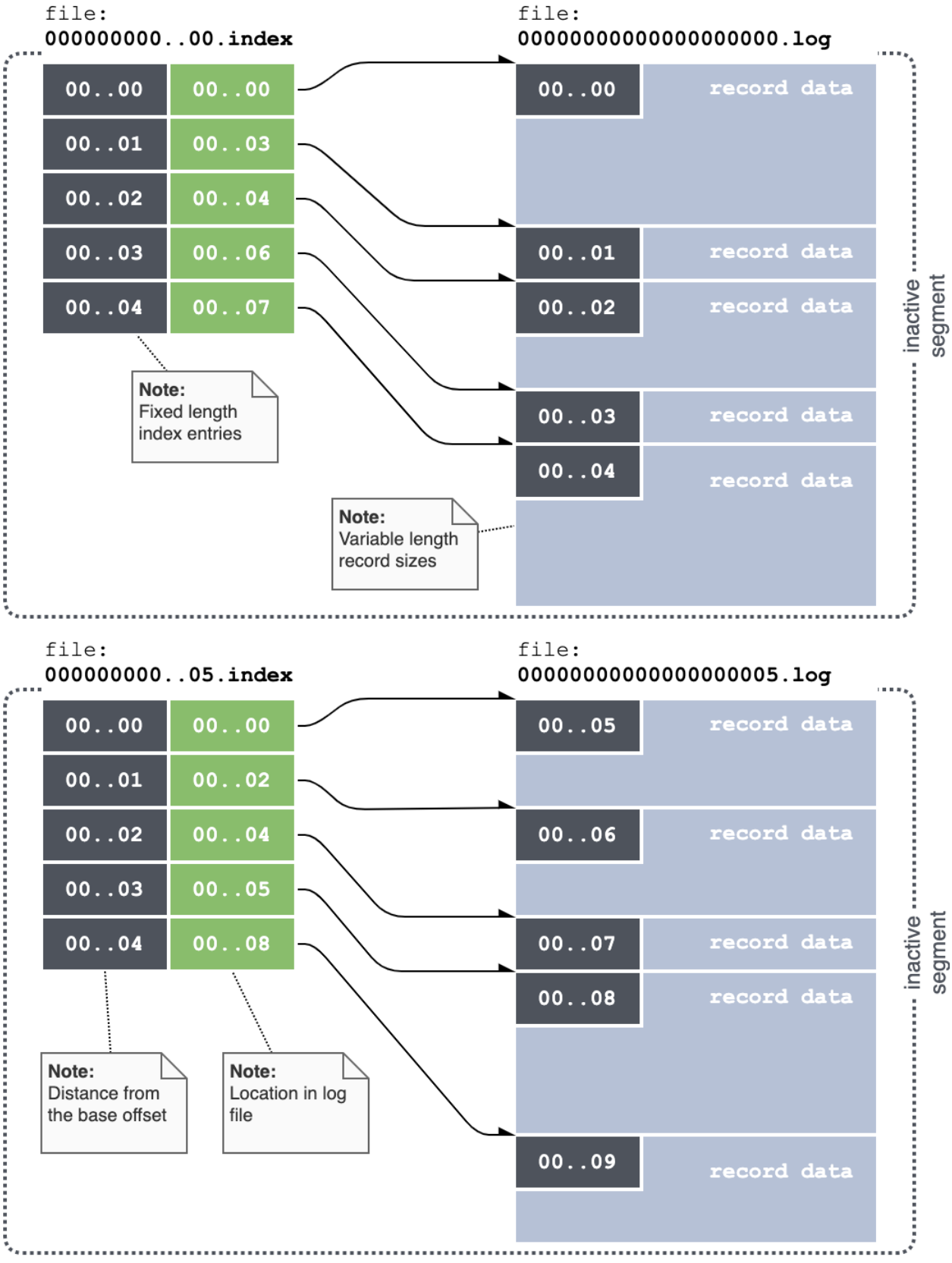
Not every record in the log has a corresponding entry in the index. For starters, records may be part of a batch, and only the location of the batch is indexed. In addition, to constrain the size of the index files and to minimise the amount of additional disk I/O, records are indexed at an interval

specified by the `log.index.interval.bytes`, which is `4096` by default. To override this setting on a per-topic basis, the `index.interval.bytes` property can be assigned on the topic entity.

With the aid of an index file, it is possible to locate a record in *O(1)* complexity. Given a record's offset — by subtracting the base offset, dividing by the index interval, and seeking to the resulting location in the index file, the physical location of the record is revealed. Where the record in question might not be indexed, the previously indexed offset is employed, requiring a short scan to find the record.

The `.timeindex` file is a close cousin of the `.index` file, mapping the millisecond-precise UNIX epoch timestamp of each record to its location in the `.log` file. The presence of `.timeindex` files allows Kafka to locate records based on time, rather than an offset.

The diagram below illustrates the segmented log structure and the use of indexes to map record offsets to physical locations in the log files.


file:
000000000..00.index
00..00
00..00
00..01
00..03
00..02
00..04
00..03
00..06
00..04
00..07
Note:
Fixed length
index entries
file:
00000000000000000000.log
00..00
record data
00..01
record data
00..02
record data
00..03
record data
00..04
record data
Note:
Variable length
record sizes
inactive
segment
file:
000000000..05.index
00..00
00..00
00..01
00..02
00..02
00..04
00..03
00..05
00..04
00..08
Note:
Distance from
the base offset
Note:
Location in log
file
file:
00000000000000000005.log
00..05
record data
00..06
record data
00..07
record data
00..08
record data
00..09
record data
inactive
segment
file:
000000000..10.index
00..00
00..00
00..01
00..01
file:
00000000000000000010.log
00..10
record data
00..11
record data
active
segment

Rotation of log segments

With the basics covered, we arrive at the most crucial part of our preamble — the rotation of log segments. Specifically, it is the log rotation behaviour that has the greatest potential to cause confusion in relation to Kafka's data retention mechanisms that will be discussed shortly.

A replica maintains a single *active* log segment for every partition that it hosts. The active segment corresponds to the *head end* of the log. All writes performed on behalf of producers are appended to the active log segment, while consumer reads may occur from any segment in the log. For a given active segment, no entity other than a producer is permitted to modify the segment and its corresponding index files; this constraint is positively intuitive in relation to the append-only nature of a log in Kafka.

At some point, the active segment file will approach a set of predefined constraints that, when breached, will trigger a roll-over. These constraints are defined by the following broker properties:

- `log.segment.bytes` — the maximum size of a single log file. Defaults to `1073741824` (1 GiB).
- `log.roll.hours` — the maximum amount of time (in hours) that an active segment is allowed to exist before being closed. The default value is `168` (one week). The age of a segment is derived by subtracting the timestamp of the first record from the current time. The size-based and time-based properties are complementary — tripping either of these thresholds is sufficient to initiate a roll-over.
- `log.roll.ms` — where the use of hour multiples to specify the roll-over threshold does not provide sufficient granularity, this property may be used to specify the threshold with millisecond precision. When set, this property supersedes `log.roll.hours`.

Replicas typically host multiple log files, particularly when dealing with 'wide' topics (i.e. topics with lots of partitions). It is likely that when a log segment reaches its roll-over threshold, other log segments would have reached theirs, or are just about to. To prevent a stampede of I/O requests, a broker will introduce a random amount of *jitter* — an artificial delay from the time a roll-over was triggered to the time of enacting the roll-over — thereby spreading the correlated roll-over of multiple log segments over a larger time period. The jitter values apply to the `log.roll.hours` and `log.roll.ms` settings, and are named `log.roll.jitter.hours` and `log.roll.jitter.ms`, respectively. No jitter is applied by default.

The properties listed above represent the baseline configuration applicable to all topics, and may be updated statically or dynamically, but cannot be used to target individual topics. There are equivalent per-topic properties that can be set dynamically:

- `segment.bytes` — in place of `log.segment.bytes`.
- `segment.ms` — in place of `log.roll.ms`.
- `segment.jitter.ms` — in place of `log.roll.jitter.ms`.

While index files are preallocated by default, log segments are not — owing to the log segment being rather large (1 GiB by default). Also, Linux-based filesystems are generally performant with respect to append operations. The lack of preallocation has been known to cause performance issues when Kafka is run on NTFS and some older Linux filesystems. To enable preallocation of log files, set the `log.preallocate` property to `true` for all topics, or the `preallocate` property when targetting individual topics.

Viewing log contents

Both log files and index files are encoded in binary form for compactness. Reading the data directly will not translate to anything particularly meaningful to the user, with the possible exception of the key or value portions of a record if these happen to be strings. To assist in diagnosing issues with logs and indexes, Kafka provides a CLI tool — `kafka-dump-log.sh`. This utility must be run locally on the broker housing the partition data, and can operate on either file type (including time indexes). The example below shows the output of `kafka-dump-log.sh` over a segment of the `getting-started-0` log.

```
LOG_DIR=/tmp/kafka-logs &&
$KAFKA_HOME/bin/kafka-dump-log.sh \
    --files $LOG_DIR/getting-started-0/00000000000000000000.log \
    --print-data-log
```

```
Dumping /tmp/kafka-logs/getting-started-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 □
    lastSequence: -1 producerId: -1 producerEpoch: -1 □
    partitionLeaderEpoch: 0 isTransactional: false □
    isControl: false position: 0 CreateTime: 1577930118584 □
    size: 73 magic: 2 compresscodec: NONE crc: 2333002560 □
    isvalid: true
| offset: 0 CreateTime: 1577930118584 keysize: -1 valuesize: 5 □
    sequence: -1 headerKeys: [] payload: alpha
```

Deletion

Reverting to the original question, what happens when we accumulate too much data?

Kafka offers two independent but related strategies — called *cleanup policies* — for dealing with data retention. The first and most basic strategy is to simply *delete* old data. The second strategy is more elaborate — *compact* prior data in such a manner as to preserve the most amount of information.

Both strategies can co-exist. This section will focus on the delete strategy; the following section will address compaction.

The cleanup policy can be assigned via the `log.cleanup.policy` broker property, which takes a comma-separated list of policy values `delete` and `compact`. The default is `delete`. This setting can be assigned statically or dynamically, acting as a cross-topic default. A topic-specific policy can be assigned via the `cleanup.policy` property — overriding the default `log.cleanup.policy` for the topic in question.

The `delete` policy operates at the granularity of log segments. A background process, operating within a replica, looks at each inactive log segment to determine whether a segment is eligible for deletion. The thresholds constraining the retention of log segments are:

- `log.retention.bytes` — the maximum allowable size of the log before its segments may be pruned, starting with the tail segment. This value is not set by default as most retention requirements are specified in terms of time, rather than size.
- `log.retention.hours` — the number of hours to keep a log segment before deleting it. The default value is `168` (one week). The size-based and time-based properties are complementary; the breach of any of the two constraints is sufficient to trigger segment deletion.
- `log.retention.minutes` — specifies the retention in minutes, where hour multiples fail to provide sufficient granularity. If set, this property overrides the `log.retention.hours` property.
- `log.retention.ms` — specifies the retention in milliseconds, overriding the `log.retention.minutes` property.

The background process checks files at an interval specified by `log.retention.check.interval.ms`, defaulting to `300000` (five minutes). This is sufficient for most retention settings where the lifetime of a log is measured in hours, days or weeks. When testing the deletion policy over smaller time frames, it may be necessary to wind this setting down to decrease the time between successive checks.

As it was stated in Chapter 3: Architecture and Core Concepts, the deletion of records results in the advancement of the *beginning offsets* (also known as the low-water mark) to the point of the first preserved record. This happens automatically as log segments are deleted.

Because the granularity of the `delete` policy is a log segment, it may be necessary to adjust the maximum size of a segment depending on one's expectations around the responsiveness of the deletion process. Suppose there was a requirement that a record should not be kept longer than seven days, but at least five days of retention is required. Setting `log.retention.hours` to `168` (seven days) would satisfy the lower bound of five days, but depending on how long it takes for the active log segment to roll over, the upper bound might not be satisfied. Recall, only producer-initiated writes are allowed on the active segment; all other manipulations — deletion and compaction — may only occur on the inactive segments. With the default settings in place, log files are not rotated until they either reach a maturity of seven days or grow to over 1 GiB in size. To satisfy the upper bound, we must constrain the lifetime of a log segment, so that it forcibly rolls over within a more suitable time frame. For example, we might set `log.retention.hours` to `132` (five days and twelve hours) and `log.roll.hours` to `24`, thereby forcing a collection after six-and-a-half days.

The `log.roll.hours` setting (and its more fine-grained equivalents) are an effective measure for the predictable rotation of log files in low-throughput topics, where there is insufficient write pressure to cause a roll-over due to file size alone. However, even the time-based trigger requires at least one write to the active segment for the broker to realise that its time is up, so to speak. If a topic is experiencing no writes for whatever reason, Kafka will not proactively rotate the active log segment, even if the latter has outlived its expected lifespan. So in the example above, we can only satisfy the upper bound requirement if there is a constant trickle of records to the topic in question.

Log deletion applies not just to user-defined topics, but also internal topics such as `__consumer_offsets`. For more information on how the deletion of log segments may affect consumer groups, see Chapter 5: Getting Started.

Compaction

Use cases behind compaction

Overwhelmingly, Kafka is used as a replicated log for notifying downstream parties of events of certain relevance, where the latter typically correlate to changes in an upstream system of record. Often, this system is backed by a database or some other persistent data store that offers an interface for querying the current state of the data, should the need for a holistic view of the state snapshot arise. Consumers may build their bespoke projections of the upstream state by faithfully replicating all relevant changes, provided these are emitted as events. At either rate, Kafka is used as an intermediate transport of sorts — essential middleware that underpins a broader, event-driven system. And as essential as it may appear in this context, in the majority of its uses, Kafka does not entirely replace a system of record.

Although as one might argue, Kafka comes remarkably close to fulfilling the role of a data store — particularly as applications grow more dependent on a replicated log. In some event-driven systems, the notion of querying the master data store has been virtually eliminated or, more often, relegated to the *hydration* scenario, where first-time consumers may query the master data in order to prime their internal projection, and switch to Kafka-based replication thereafter. For the reader's reference, such systems are called *bimodal* — in that they employ one mode for the initial *hydration* and another for the subsequent *replication*. While they are mostly straightforward — in that both modes can be easily reasoned about — there is the added complexity of maintaining code that deals with a one-off event — a contingency that typically occurs once in the lifetime of an application in its target operating environment.

What if Kafka was going to be used for event sourcing? Specifically, in the role of a definitive event store that any consumer could use to reconstitute the state of an entire domain from first principles — by replaying all updates from the point of conception. Such an arrangement would have a clear benefit — *unimodality*; specifically, using one mode of operation to accommodate both the *hydration* scenario and the subsequent *replication* of event data. Simpler code; easier to test; easier to maintain.

An event store recording all changes to every entity of interest over its entire lifetime? That sounds like a scalability nightmare. Where would one find the space to store all this data? And how long would the initial hydration take — having to painstakingly replay all records from day dot?

Even with infinite retention, the naive approach of storing everything would hardly be practical — especially for fast-moving data where any given entity might be characterised by a considerable number of updates over its lifetime. The replay time would grow beyond comprehension; we would have ourselves an event store that no one would dare read from. On the flip side, discarding the oldest records, while binding the replay time, would result in the loss of data — forcing us back down the bimodal path.

Kafka accommodates the classical unimodal event sourcing scenario using its log compaction feature. Where deletion operates at the segment level, compaction provides a finer-grained, per-record culling strategy — as opposed to the coarser-grained time-based retention. The idea is to selectively prune a related set of records, where the latter represents updates to some long-lived entity, and where a more recent (and therefore, more relevant) update record exists for the same key. In doing so, the log is guaranteed to have at least one record that represents the most recent snapshot of an entity — sacrificing the timeline of updates in order to maximally preserve the current state.

Another way of looking at compaction is that it turns a stream into a snapshot, where only the most recent information is retained. This is akin to a row in a database, where the latter disregards the timeline of updates — keeping only the last value. For this to work, the granularity of an update must align with the granularity of the underlying entity; in other words, *updates must be fully self-contained*. Compaction is intractable when updates contain *deltas* — piecemeal information that can be used to incrementally build the state of an entity. If a more recent update supersedes its logical predecessor, *there cannot be any information contained within the preceding record that cannot be obtained by inspecting the successor.*

Overwriting and deleting records

Compaction is activated by adding `compaction` to the list of policies in `log.cleanup.policy` (or `cleanup.policy` for the per-topic setting). Apart from adding the policy, compaction requires the *cleaner* process, which is enabled by default and controlled by the `log.cleaner.enable` property.

Once compaction is activated, there is nothing more required from the producer to overwrite an older record – one simply publishes an update with the same key. With compaction in force, the number of records approaches the number of entities that are being described. When the useful lifespan of an entity nears its conclusion, any previous updates for that entity may be deleted by publishing a *tombstone* — a record for the same key as the prior updates but with a `null` value. Tombstones will trigger the purging of like records, but will themselves be retained for some time before eventually being purged. (The reason for this will be discussed shortly.) There is nothing special about tombstones from a consumer's perspective — it will see records with `null` values — exactly as they were written by the producer.

Once a topic has been fully compacted (old records removed and tombstones purged), the amount of time a consumer will take to traverse the topic end-to-end will be proportional to the number of

retained unique keys. This is precisely what is needed to sustainably support the majority of event sourcing scenarios.

Behind the scenes

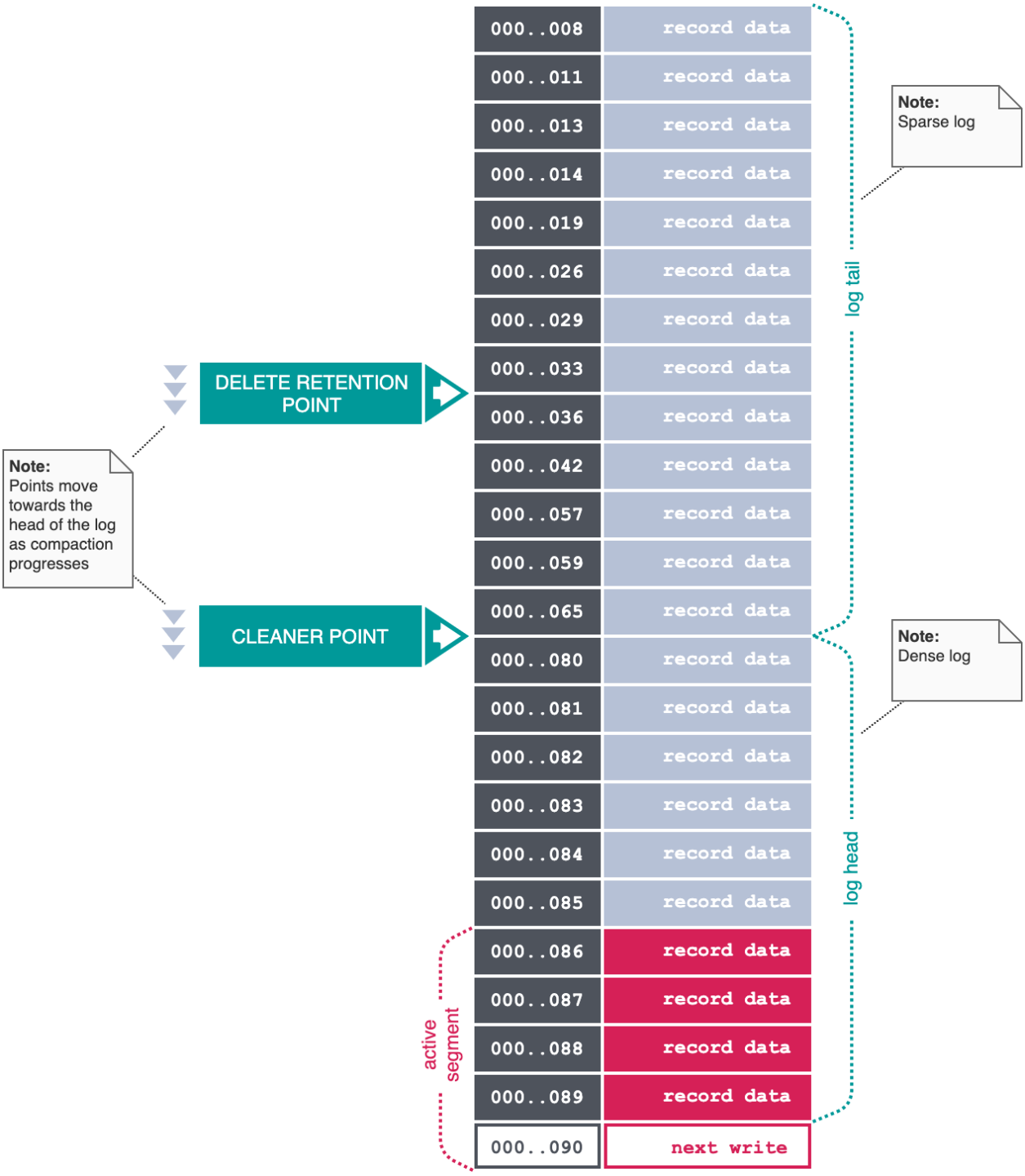
Compaction is fundamentally asynchronous, operating in the background by trailing the inactive log segments. Under the hood, log compaction recruits dedicated threads on each replica — the exact number of threads is given by the `log.cleaner.threads` broker property (defaulting to `1`). Compaction is driven by four main properties:

- `log.cleaner.min.cleanable.ratio` — the minimum ratio of the *dirty* log size to the total log size for a log to be eligible for cleaning. The default value is `0.5`. More on the dirty ratio in a moment.
- `log.cleaner.min.compaction.lag.ms` — prevents records newer than a minimum age from being subjected to compaction. The default value is `0`, implying there is no minimum time that a record will persist before it may be compacted, subject to the constraint of the active log segment. (Recall, only records in inactive segments are subject to manipulation.)
- `log.cleaner.max.compaction.lag.ms` — the upper bound on the compaction lag. Typically used with low-throughput topics, where logs might not exceed the `log.cleaner.min.cleanable.ratio` and, as such, will remain ineligible for compaction for an unbounded duration. The default setting is `Long.MAX_VALUE`, implying that compaction is driven by the `log.cleaner.min.cleanable.ratio`. (Note, the upper bound is not a hard deadline, being subject to the availability of cleaner threads and the actual compaction time.)
- `log.cleaner.delete.retention.ms` — the duration that the tombstone records are persisted before being purged from the log. More on tombstone retention in a moment.

The settings above act as cluster-wide defaults. Their per-topic equivalents are:

- `min.cleanable.dirty.ratio` — in place of `log.cleaner.min.cleanable.ratio`.
- `min.compaction.lag.ms` — in place of `log.cleaner.min.compaction.lag.ms`.
- `max.compaction.lag.ms` — in place of `log.cleaner.max.compaction.lag.ms`.
- `delete.retention.ms` — in place of `log.cleaner.delete.retention.ms`.

For every log being compacted, the cleaner maintains two logical points — the *cleaner point* and the *delete retention point*. The cleaner point is the location in the log corresponding to the progress of the cleaner through the log, dividing the log into two parts. The part older than the cleaner point is called the *log tail* and corresponds to those segments that have already been compacted by the cleaner. Conversely, the part of the log that is newer than the cleaner point is called the *log head* and corresponds to those segments that have yet to be compacted, including the active segment. The diagram below illustrates the location of these points in relation to the overall log.



Log compaction

The cleaner thread prioritises the log with the highest ratio of the head size to the overall length of the log — known as the *dirty ratio*. For a log to be selected for compaction, its dirty ratio must exceed the `log.cleaner.min.cleanable.ratio` or, alternatively the oldest uncompacted record must exceed the `log.cleaner.max.compaction.lag.ms`.

To perform the compaction, the cleaner scans the segments in the head portion, including the active segment, populating an in-memory hash table of unique record keys mapped to their most recent offsets in the log. All inactive log segments and their indexes are recopied (including those that have been previously compacted), purging records that have been superseded – except for tombstones, which are treated differently to conventional records. As the log cleaner works through the log segments, new segments get swapped into the log, replacing older segments. This way compaction does not require double the space of the entire partition; only the free disk space for one additional log segment is required.

Compaction preserves the order of the surviving records and their original offset; it will never reorder records or manipulate them in a way that might confuse a prospective consumer or contradict the producer's original intent. In effect, compaction converts a densely populated log into a sparse one, with 'holes' in offsets that should be disregarded by consumers. Reverting to what was stated in Chapter 3: Architecture and Core Concepts, a consumer should not interpret the offsets literally — using offsets purely as a means of inferring relative order.

The delete retention point lags behind the cleaner, prolonging the lifetime of tombstone records by an additional `log.cleaner.delete.retention.ms` on top of the `log.cleaner.min.compaction.lag.ms` accorded to conventional records. The `log.cleaner.delete.retention.ms` property applies equally to all topics, defaulting to `86400000` (24 hours); its per-topic equivalent is `delete.retention.ms`.

Tombstone records are preserved longer by design. The motivation for preserving tombstones is revealed in the following scenario. A publisher emits records for an entity, characterised by `Create` and `Update` events. Eventually, when the entity is deleted, the producer will follow with a tombstone — indicating to the compactor and the consumer ecosystem that the entity in question should be completely purged from the topic and any projections. This makes sense for a consumer that has not seen any traces of the entity. If the entity has already been dropped, then processing `Create`, `Update` and tombstone records for it is a waste of the consumer's time. It is more than a trivial waste of resources; in fact, it runs contrary to the requirements of a pure event store — the consumer should only be exposed to events that are legitimately required to reconstitute the snapshot of the current state, having been initialised with a clean slate.

The issue with deleting tombstones becomes apparent when considering a consumer operating with minimal lag that is part-way through handling entity updates for a soon-to-be-deleted entity. Suppose the producer has published `Create` and `Update` events for some entity, which the consumer has caught up with. The consumer then begins to accrue lag (perhaps it has gone offline momentarily). Meanwhile, the producer writes a tombstone record for the entity in question. If the compactor kicks in before the consumer resumes processing, the removal of all records for the key referenced by the tombstone would create an inconsistency on the consumer — leaving it in a state where the entity is retained in perpetuity. The handling of tombstones creates a contradiction between the requirements of hydration and subsequent replication — the former requires the tombstones to be deleted, while the latter needs tombstones in place to maintain consistency.

By allowing an additional grace period for tombstone records, Kafka supports both use cases. The tombstone retention time should be chosen such that it covers the maximum lag that a consumer might reasonably accumulate. The default setting of 24 hours may not be sufficient where consumers

are intentionally taken offline for long periods; for example, non-production environments. Also, it might not be sufficient for systems that need to tolerate longer periods of downtime; for example, to account for operational contingencies.

Compaction is a resource-intensive operation, requiring a full scan of the log each time it is run. The purpose of the `log.cleaner.min.cleanable.ratio` property (and its per-topic equivalent) is to throttle compaction, preventing it from running continuously, in detriment to the broker's normal operation. In addition to the cleanable ratio, the compactor can further be throttled by applying the `log.cleaner.backoff.ms` property, controlling the amount of time the cleaner thread will sleep in between compaction runs, defaulting to `15000` (15 seconds). On top of this, the `log.cleaner.io.max.bytes.per.second` property can be used to apply an I/O quota to log cleaner thread, such that the sum of its read and write disk bandwidth does not exceed the stated value. The default setting is `Double.MAX_VALUE`, which effectively disables the quota.

An example

To demonstrate the Kafka compactor in action, we are going to create a topic named `prices` with aggressive compaction settings:

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --topic prices --replication-factor 1 --partitions 1 \
    --config "cleanup.policy=compact" \
    --config "delete.retention.ms=100"  \
    --config "segment.ms=1" \
    --config "min.cleanable.dirty.ratio=0.01"
```

Normally, setting such small segment sizes and aggressive cleanable ratios is strongly discouraged, as it creates an obscenely large number of files and leads to heavy resource utilisation — every new record will effectively mark the partition as dirty, instigating compaction. This is done purely for show, as with conservative settings the compactor will simply not kick in for our trivial example. (The records will be held in the active segment, where they will remain out of the compactor's reach.)

Next, run a console producer:

```
$KAFKA_HOME/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 --topic prices \
    --property parse.key=true --property key.separator=:
```

Copy the following line by line, allowing for a second or two in between each successive write for the log files to rotate. Don't copy-paste the entire block, as this will likely group the records into a batch on the producer, placing them all into the active segment.

```
AAPL:279.74
AAPL:280.03
MSFT:157.14
MSFT:156.01
AAPL:284.90
IBM:100.50
```

Press `CTRL-D` when done.

Now run the consumer. Assuming that the compactor has had a chance to run, the output should be constrained to the unique record keys and their most recent values.

```
$KAFKA_HOME/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 \
    --topic prices --from-beginning \
    --property print.key=true
```

```
MSFT  156.01
AAPL  284.90
IBM   100.50
```

Also, we can see the effects of compaction on the log files:

```
tree -s /tmp/kafka-logs/prices-0
```

```
/tmp/kafka-logs/prices-0
├── [          0]  00000000000000000000.index
├── [        156]  00000000000000000000.log
├── [         12]  00000000000000000000.timeindex
├── [         10]  00000000000000000001.snapshot
├── [         10]  00000000000000000002.snapshot
├── [         10]  00000000000000000003.snapshot
├── [         10]  00000000000000000004.snapshot
├── [   10485760]  00000000000000000005.index
├── [         77]  00000000000000000005.log
├── [         10]  00000000000000000005.snapshot
├── [   10485756]  00000000000000000005.timeindex
└── [          8]  leader-epoch-checkpoint

0 directories, 12 files
```

The active log segment is `00000000000000000005.log`. All prior segments have been compacted, leaving hollow snapshots in their place; the remaining unique records were coalesced into `00000000000000000000.log`. Out of interest, we can view its contents using the `kafka-dump-logs.sh` utility:

```
$KAFKA_HOME/bin/kafka-dump-log.sh --print-data-log \
    --files /tmp/kafka-logs/prices-0/00000000000000000000.log


Dumping /tmp/kafka-logs/prices-0/00000000000000000000.log
Starting offset: 0
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 □
    lastSequence: -1 producerId: -1 producerEpoch: -1 □
    partitionLeaderEpoch: 0 isTransactional: false □
    isControl: false position: 0 CreateTime: 1577409425248 □
    size: 78 magic: 2 compresscodec: NONE crc: 2399134455 □
    isvalid: true
| offset: 3 CreateTime: 1577409425248 keysize: 4 valuesize: 6 □
    sequence: -1 headerKeys: [] key: MSFT payload: 156.01
baseOffset: 4 lastOffset: 4 count: 1 baseSequence: -1 □
    lastSequence: -1 producerId: -1 producerEpoch: -1 □
    partitionLeaderEpoch: 0 isTransactional: false □
    isControl: false position: 78 CreateTime: 1577409434843 □
    size: 78 magic: 2 compresscodec: NONE crc: 2711186080 □
    isvalid: true
| offset: 4 CreateTime: 1577409434843 keysize: 4 valuesize: 6 □
    sequence: -1 headerKeys: [] key: AAPL payload: 284.90
```

This is showing us that the `MSFT` and `AAPL` records have been bunched up together, although originally they would have appeared in different segments. The `IBM` record, having been published last, will be in the active segment.

The existence of the active segment is probably the most confusing aspect with respect to compaction, catching out its fair share of practitioners. When a topic appears to be partially compacted, in most cases the reason is that the uncompacted records lie in the active segment, and are therefore out of scope.

Combining compaction with deletion

As it was previously stated, it is possible to assign both `delete` and `compact` policies to the `log.cleanup.policy` property (or `cleanup.policy`, as the case may be). The question is: does it make sense to do so?

When a topic is configured for bounded retention, the typical use cases that are being serviced are bimodal replication of events or conventional event stream processing — where a consumer ecosystem or a series of processing stages react to events from an upstream emitter. Unbounded retention is typically a surplus to requirements; it is sufficient for the retention time to cover the maximum forecast consumer lag.

Conversely, when a topic is compacted, one naturally assumes that it is being utilised as the source of truth — where the preservation of state is an essential attribute — more so than the preservation of discrete changes. Under this model, there is either a characteristic absence of a queryable data source, or it may be intractable for the consumer to issue queries. This model enables unimodal processing, which is its dominant advantage.

Combining the two strategies might appear counterintuitive at first. It has been said: a compacted topic can be logically equated to a database (or a K-V store) — the most recent updates corresponding to rows in a table of finite size. Slapping a deletion policy on top feels like we are building a database that silently drops records when it gets to a certain size (or age). This configuration is peculiar, at the very least.

Nonetheless, a size-bound, compacted topic is useful in limited cases of change data capture and processing. This can be seen as an extension of the conventional, event-driven replication model. In the presence of fast-moving data, where updates are self-contained and the time-value of data is low, the use of compaction can dramatically accelerate the processing of data. More often than not, entity deletion is a non-issue, and tombstones are excluded from the model (although they don't have to be). Here, the intention is not to conserve space or to enable unimodal processing; rather, it is to reduce resource wastage on processing events that rapidly lose their significance. Because the size of the topic is inherently constrained (by way of deletion), one can afford to run a more aggressive compactor, using a much lower `log.cleaner.min.cleanable.ratio` in conjunction with smaller segment sizes. The more aggressive the compaction, the lower the likelihood that an obsolete record is observed by a consumer.

An example of where this 'hybrid' model is used is the internal `__consumer_offsets` topic, used to manage state information for consumer groups. This topic's segments are subjected to both compaction and deletion. Compaction allows for the rapid reconstruction of consumer state on the group coordinator, while deletion binds the size of the topic and acts as a natural 'garbage collector' of consumer groups that have not been utilised for some time.

This chapter looked at how Kafka deals with data retention. We started with a tour of its storage internals — understanding how topic-partition data on the replicas maps to the underlying filesystem. Among the key takeaways is the notion of a segmented log, whereby the partition is broken up into a series of chunks — each accompanied by a set of indexes — permitting *O(1)* lookups of records by their offset or timestamp. Of the log segments, the most recent is the *active* segment. Only a producer may write to the active segment — a fundamental design decision that influences the remainder of Kafka's data retention machinery. And also, one that frequently causes confusion.

Kafka offers two independent but related cleanup policies — *deletion* and *compaction*. Deletion is relatively straightforward — truncating log segments when the log reaches a certain size or where the oldest record matures to a certain age. As a cleanup policy, deletion is suitable where consumers

are expected to keep up with the data syndicated over a topic, or where an alternate mechanism for retrieving any lapsed data exists.

Compaction is the more elaborate cleanup policy, providing a fine-grained, per-record retention strategy. The idea is to selectively remove records where the latter represent entity updates, and where a more recent update record exists for the same key. Compaction leaves at least one record that represents the most recent snapshot of an entity — turning an event stream into a continuously updating data store.

Crucially, compaction is a *best-effort* endeavour — it ensures that at least one record for a given key is preserved. There is no *exactly-one* guarantee when it comes to compaction. Depending on various factors — the contents of the active segment and the dirty ratio — compaction might not run, or having run, it might appear to have a partial effect. Also, compaction is not bound by a deadline — being subject to the availability of cleaner threads, the backlog of competing partitions in need of compaction, and the actual compaction time.

The deletion and compaction policies may be used in concert, which is typically done to expedite the processing of change records while constraining the size of the log.

Among them, Kafka's cleanup policies provide sufficient flexibility to accommodate a range of data retention needs — from event-driven services to accelerated stream processing graphs and fully-fledged event-sourcing models.