# Arenadata™ Streaming

*Версия - v1.3-ENG*

## Architectural features of Arenadata Streaming

*2021 Arenadata™*

# Оглавление


**1 Overview** **3**

**2 Persistence** **4**

**3 Efficiency** **6**

**4 The Producer** **8**
- 4.1 Load balancing . . . 8
- 4.2 Asynchronous send . . . 8

**5 The Consumer** **9**
- 5.1 Consumer Position . . . 9
- 5.2 Offline Data Load . . . 10
- 5.3 Consumer Offset Tracking . . . 10
- 5.4 Migrating offsets from ZooKeeper to ADS . . . 11

**6 Message Delivery Semantics** **12**

**7 Replication** **14**

**8 Log Compaction** **16**
- 8.1 Log Compaction Basics . . . 17
- 8.2 What guarantees does log compaction provide? . . . 18
- 8.3 Log Compaction Details . . . 19

The documentation describes the architectural features of the platform Arenadata Streaming.

The instruction can be useful to administrators, programmers, developers and employees of information technology departments who implement and maintain the system.

**Important:** Contact information support service – e-mail: info@arenadata.io

# Глава 1

# Overview

Developed by **Arenadata** platform **ADS** has the ability to act as a unified platform for handling all the real-time data feeds a large company might have. To do this we had to think through a fairly broad set of use cases:

- **ADS** would have to have high-throughput to support high volume event streams such as real-time log aggregation;
- **ADS** would need to deal gracefully with large data backlogs to be able to support periodic data loads from offline systems;
- **ADS** also meant the system would have to handle low-latency delivery to handle more traditional messaging use-cases.

**Arenadata** wanted to support partitioned, distributed, real-time processing of these feeds to create new, derived feeds. This motivated our partitioning and consumer model.

Finally in cases where the stream is fed into other data systems for serving, the system would have to be able to guarantee fault-tolerance in the presence of machine failures.

Supporting these uses led **Arenadata** to a design with a number of unique elements, more akin to a database log than a traditional messaging system.

# Глава 2

# Persistence

**ADS** relies heavily on the filesystem for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact disks are both much slower and much faster than people expect depending on how they are used; and a properly designed disk structure can often be as fast as the network.

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result the performance of linear writes on a **JBOD** configuration with six *7200rpm* **SATA RAID-5** array is about *600MB/sec* but the performance of random writes is only about *100k/sec* – a difference of over *6000X*. These linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system. A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. A further discussion of this issue can be found in this ACM Queue article; they actually find that sequential disk access can in some cases be faster than random memory access.

To compensate for this performance divergence, modern operating systems have become increasingly aggressive in their use of main memory for disk caching. A modern OS will happily divert all free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache. This feature cannot easily be turned off without using direct I/O, so even if a process maintains an in-process cache of the data, this data will likely be duplicated in OS pagecache, effectively storing everything twice.

Furthermore, the platform **ADS** building on top of the **JVM**, and anyone who has spent any time with **Java** memory usage knows two things:

- The memory overhead of objects is very high, often doubling the size of the data stored (or worse);
- Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.

As a result of these factors using the filesystem and relying on pagecache is superior to maintaining an in-memory cache or other structure – we at least double the available cache by having automatic access to all free memory, and likely double again by storing a compact byte structure rather than individual objects. Doing so will result in a cache of up to *28-30 GB* on a *32 GB* machine without GC penalties. Furthermore, this cache will stay warm even if the service is restarted, whereas the in-process cache will need to be rebuilt in memory (which for a *10 GB* cache may take *10 minutes*) or else it will need to start with a completely cold cache (which likely means terrible initial performance). This also greatly simplifies the code as all logic for maintaining coherency between the cache and filesystem is now in the OS, which tends to do so more efficiently and more correctly than one-off in-process attempts. If your disk usage favors linear reads then read-ahead is effectively pre-populating this cache with useful data on each disk read.

This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush it all out to the filesystem in a panic when we run out of space, we invert that. All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk. In effect this just means that it is transferred into the kernel's pagecache.

This style of pagecache-centric design is described in an article Notes from the Architect.

The persistent data structure used in messaging systems are often a per-consumer queue with an associated BTree or other general-purpose random access data structures to maintain metadata about messages. BTrees are the most versatile data structure available, and make it possible to support a wide variety of transactional and non-transactional semantics in the messaging system. They do come with a fairly high cost, though: Btree operations are $O(\log N)$. Normally $O(\log N)$ is considered essentially equivalent to constant time, but this is not true for disk operations. Disk seeks come at $10\ ms$ a pop, and each disk can do only one seek at a time so parallelism is limited. Hence even a handful of disk seeks leads to very high overhead. Since storage systems mix very fast cached operations with very slow physical disk operations, the observed performance of tree structures is often superlinear as data increases with fixed cache – i.e. doubling your data makes things much worse than twice as slow.

Intuitively a persistent queue could be built on simple reads and appends to files as is commonly the case with logging solutions. This structure has the advantage that all operations are $O(1)$ and reads do not block writes or each other. This has obvious performance advantages since the performance is completely decoupled from the data size—one server can now take full advantage of a number of cheap, low-rotational speed $1+TB\ SATA$ drives. Though they have poor seek performance, these drives have acceptable performance for large reads and writes and come at $1/3$ the price and $3x$ the capacity.

Having access to virtually unlimited disk space without any performance penalty means that **ADS** can provide some features not usually found in a messaging system. For example, in **ADS**, instead of attempting to delete messages as soon as they are consumed, can retain messages for a relatively long period (say a week). This leads to a great deal of flexibility for consumers.

# Глава 3

# Efficiency

Specialists of **Arenadata** put significant effort into efficiency. One of our primary use cases is handling web activity data, which is very high volume: each page view may generate dozens of writes. Furthermore, we assume each message published is read by at least one consumer (often many), hence we strive to make consumption as cheap as possible.

We have also found, from experience building and running a number of similar systems, that efficiency is a key to effective multi-tenant operations. If the downstream infrastructure service can easily become a bottleneck due to a small bump in usage by the application, such small changes will often create problems. By being very fast we help ensure that the application will tip-over under load before the infrastructure. This is particularly important when trying to run a centralized service that supports dozens or hundreds of applications on a centralized cluster as changes in usage patterns are a near-daily occurrence.

Once poor disk access patterns have been eliminated, there are two common causes of inefficiency in this type of system: too many small I/O operations, and excessive byte copying.

The small I/O problem happens both between the client and the server and in the server's own persistent operations. To avoid this, **ADS** protocol is built around a "message set" abstraction that naturally groups messages together. This allows network requests to group messages together and amortize the overhead of the network roundtrip rather than sending a single message at a time. The server in turn appends chunks of messages to its log in one go, and the consumer fetches large linear chunks at a time.

This simple optimization produces orders of magnitude speed up. Batching leads to larger network packets, larger sequential disk operations, contiguous memory blocks, and so on, all of which allows Kafka to turn a bursty stream of random message writes into linear writes that flow to the consumers.

The other inefficiency is in byte copying. At low message rates this is not an issue, but under load the impact is significant. To avoid this **ADS** employ a standardized binary message format that is shared by the producer, the broker, and the consumer (so data chunks can be transferred without modification between them).

The message log maintained by the broker is itself just a directory of files, each populated by a sequence of message sets that have been written to disk in the same format used by the producer and consumer. Maintaining this common format allows optimization of the most important operation: network transfer of persistent log chunks. Modern **unix** operating systems offer a highly optimized code path for transferring data out of pagecache to a socket; in **Linux** this is done with the *sendfile* system call.

To understand the impact of *sendfile*, it is important to understand the common data path for transfer of data from file to socket:

1. The operating system reads data from the disk into pagecache in kernel space.
2. The application reads the data from kernel space into a user-space buffer.
3. The application writes the data back into kernel space into a socket buffer.

4. The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network.

This is clearly inefficient, there are four copies and two system calls. Using *sendfile*, this re-copying is avoided by allowing the OS to send the data from pagecache to the network directly. So in this optimized path, only the final copy to the NIC buffer is needed.

A common use case to be multiple consumers on a topic. Using the zero-copy optimization above, data is copied into pagecache exactly once and reused on each consumption instead of being stored in memory and copied out to user-space every time it is read. This allows messages to be consumed at a rate that approaches the limit of the network connection.

This combination of pagecache and *sendfile* means that on a **ADS** cluster where the consumers are mostly caught up you will see no read activity on the disks whatsoever as they will be serving data entirely from cache.

For more background on the *sendfile* and zero-copy support in **Java**, see this article.

# Глава 4

# The Producer

## 4.1 Load balancing

The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier. To help the producer do this all **ADS** nodes can answer a request for metadata about which servers are alive and where the leaders for the partitions of a topic are at any given time to allow the producer to appropriately direct its requests.

The client controls which partition it publishes messages to. This can be done at random, implementing a kind of random load balancing, or it can be done by some semantic partitioning function. The interface **ADS** exposes for semantic partitioning by allowing the user to specify a key to partition by and using this to hash to a partition (there is also an option to override the partition function if need be). For example if the key chosen was a user id then all data for a given user would be sent to the same partition. This in turn will allow consumers to make locality assumptions about their consumption. This style of partitioning is explicitly designed to allow locality-sensitive processing in consumers.

## 4.2 Asynchronous send

Batching is one of the big drivers of efficiency, and to enable batching the producer will attempt to accumulate data in memory and to send out larger batches in a single request. The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound (say $64k$ or $10\ ms$). This allows the accumulation of more bytes to send, and few larger I/O operations on the servers. This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput.

# Глава 5

# The Consumer

The **ADS** consumer works by issuing "fetch" requests to the brokers leading the partitions it wants to consume. The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position. The consumer thus has significant control over this position and can rewind it to re-consume data if need be.

An initial question we considered is whether consumers should pull data from brokers or brokers should push data to the consumer. In this respect **ADS** follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. Some logging-centric systems, such as **Scribe** and **Apache Flume**, follow a very different push-based path where data is pushed downstream. There are pros and cons to both approaches. However, a push-based system has difficulty dealing with diverse consumers as the broker controls the rate at which data is transferred. The goal is generally for the consumer to be able to consume at the maximum possible rate; unfortunately, in a push system this means the consumer tends to be overwhelmed when its rate of consumption falls below the rate of production (a denial of service attack, in essence). A pull-based system has the nicer property that the consumer simply falls behind and catches up when it can. This can be mitigated with some kind of backoff protocol by which the consumer can indicate it is overwhelmed, but getting the rate of transfer to fully utilize (but never over-utilize) the consumer is trickier than it seems. Previous attempts at building systems in this fashion led us to go with a more traditional pull model.

Another advantage of a pull-based system is that it lends itself to aggressive batching of data sent to the consumer. A push-based system must choose to either send a request immediately or accumulate more data and then send it later without knowledge of whether the downstream consumer will be able to immediately process it. If tuned for low latency, this will result in sending a single message at a time only for the transfer to end up being buffered anyway, which is wasteful. A pull-based design fixes this as the consumer always pulls all available messages after its current position in the log (or up to some configurable max size). So one gets optimal batching without introducing unnecessary latency.

The deficiency of a naive pull-based system is that if the broker has no data the consumer may end up polling in a tight loop, effectively busy-waiting for data to arrive. To avoid this **ADS** has parameters in pull request that allow the consumer request to block in a "long poll" waiting until data arrives (and optionally waiting until a given number of bytes is available to ensure large transfer sizes).

## 5.1 Consumer Position

Keeping track of what has been consumed is, surprisingly, one of the key performance points of a messaging system.

Most messaging systems keep metadata about what messages have been consumed on the broker. That is, as a message is handed out to a consumer, the broker either records that fact locally immediately or it may wait for acknowledgement from the consumer. This is a fairly intuitive choice, and indeed for a single machine server it is not clear where else this state could go. Since the data structures used for storage in many messaging systems scale poorly,

this is also a pragmatic choice – since the broker knows what is consumed it can immediately delete it, keeping the data size small.

What is perhaps not obvious is that getting the broker and consumer to come into agreement about what has been consumed is not a trivial problem. If the broker records a message as consumed immediately every time it is handed out over the network, then if the consumer fails to process the message (say because it crashes or the request times out or whatever) that message will be lost. To solve this problem, many messaging systems add an acknowledgement feature which means that messages are only marked as sent not consumed when they are sent; the broker waits for a specific acknowledgement from the consumer to record the message as consumed. This strategy fixes the problem of losing messages, but creates new problems. First of all, if the consumer processes the message but fails before it can send an acknowledgement then the message will be consumed twice. The second problem is around performance, now the broker must keep multiple states about every single message (first to lock it so it is not given out a second time, and then to mark it as permanently consumed so that it can be removed). Tricky problems must be dealt with, like what to do with messages that are sent but never acknowledged.

**ADS** handles this differently. Our topic is divided into a set of totally ordered partitions, each of which is consumed by exactly one consumer within each subscribing consumer group at any given time. This means that the position of a consumer in each partition is just a single integer, the offset of the next message to consume. This makes the state about what has been consumed very small, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgements very cheap.

There is a side benefit of this decision. A consumer can deliberately rewind back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

## 5.2 Offline Data Load

Scalable persistence allows for the possibility of consumers that only periodically consume such as batch data loads that periodically bulk-load data into an offline system such as **Hadoop** or a relational data warehouse.

In the case of **Hadoop** we parallelize the data load by splitting the load over individual map tasks, one for each node/topic/partition combination, allowing full parallelism in the loading. **Hadoop** provides the task management, and tasks which fail can restart without danger of duplicate data – they simply restart from their original position.

## 5.3 Consumer Offset Tracking

The high-level consumer tracks the maximum offset it has consumed in each partition and has the capability to commit offsets so that it can resume from those offsets in the event of a restart. **ADS** provides the option to store all the offsets for a given consumer group in a designated broker (for that group) called the group coordinator. i.e., any consumer instance in that consumer group should send its offset commits and fetches to that group coordinator (broker). For high-level consumers, this operation is performed automatically, while the simple consumer manages its offsets manually, since currently automatic transmission of offsets to simple consumers is not supported **Java**, they can only manually capture or extract offsets in **ZooKeeper**. When using **Scala**, a consumer can obviously commit or receive offsets in the coordinator.

When the group coordinator receives an *OffsetCommitRequest*, it appends the request to a special compacted **ADS** topic named *__consumer_offsets*. The broker sends a successful offset commit response to the consumer only after all the replicas of the offsets topic receive the offsets. In case the offsets fail to replicate within a configurable timeout, the offset commit will fail and the consumer may retry the commit after backing off. The brokers periodically compact the offsets topic since it only needs to maintain the most recent offset commit per partition. The coordinator also caches the offsets in an in-memory table in order to serve offset fetches quickly.

When the coordinator receives an offset fetch request, it simply returns the last committed offset vector from the offsets cache. In case coordinator was just started or if it just became the coordinator for a new set of consumer groups (by becoming a leader for a partition of the offsets topic), it may need to load the offsets topic partition into

the cache. In this case, the offset fetch will fail with an *CoordinatorLoadInProgressException* and the consumer may retry the *OffsetFetchRequest* after backing off.

## 5.4 Migrating offsets from ZooKeeper to ADS

To migrate consumers and their offsets from **ZooKeeper** to **ADS**:

1. Set *offsets.storage=ads* and *dual.commit.enabled=true* in your consumer config.
2. Do a rolling bounce of your consumers and then verify that your consumers are healthy.
3. Set *dual.commit.enabled=false* in your consumer config.
4. Do a rolling bounce of your consumers and then verify that your consumers are healthy.

A roll-back (i.e., migrating from **ADS** back to **ZooKeeper**) can also be performed using the above steps if you set *offsets.storage=zookeeper*.

# Глава 6

# Message Delivery Semantics

Now that we understand a little about how producers and consumers work, let's discuss the semantic guarantees **ADS** provides between producer and consumer. Clearly there are multiple possible message delivery guarantees that could be provided:

- *At most once* – Messages may be lost but are never redelivered;
- *At least once* – Messages are never lost but may be redelivered;
- *Exactly once* – this is what people actually want, each message is delivered once and only once.

It's worth noting that this breaks down into two problems: the durability guarantees for publishing a message and the guarantees when consuming a message.

Many systems claim to provide "exactly once" delivery semantics, but it is important to read the fine print, most of these claims are misleading (i.e. they don't translate to the case where consumers or producers can fail, cases where there are multiple consumer processes, or cases where data written to disk can be lost).

**ADS**'s semantics are straight-forward. When publishing a message we have a notion of the message being "committed" to the log. Once a published message is committed it will not be lost as long as one broker that replicates the partition to which this message was written remains "alive". The definition of committed message, alive partition as well as a description of which types of failures we attempt to handle will be described in more detail in the next section. For now let's assume a perfect, lossless broker and try to understand the guarantees to the producer and consumer. If a producer attempts to publish a message and experiences a network error it cannot be sure if this error happened before or after the message was committed. This is similar to the semantics of inserting into a database table with an autogenerated key.

The **ADS** producer also supports an idempotent delivery option which guarantees that resending will not result in duplicate entries in the log. To achieve this, the broker assigns each producer an ID and deduplicates messages using a sequence number that is sent by the producer along with every message. Also the producer supports the ability to send messages to multiple topic partitions using transaction – like semantics: i.e. either all messages are successfully written or none of them are. The main use case for this is exactly-once processing between **ADS** topics (described below).

Not all use cases require such strong guarantees. For uses which are latency sensitive we allow the producer to specify the durability level it desires. If the producer specifies that it wants to wait on the message being committed this can take on the order of *10 ms*. However the producer can also specify that it wants to perform the send completely asynchronously or that it wants to wait only until the leader (but not necessarily the followers) have the message.

Now let's describe the semantics from the point-of-view of the consumer. All replicas have the exact same log with the same offsets. The consumer controls its position in this log. If the consumer never crashed it could just store this position in memory, but if the consumer fails and we want this topic partition to be taken over by another process

the new process will need to choose an appropriate position from which to start processing. Let's say the consumer reads some messages – it has several options for processing the messages and updating its position:

1. It can read the messages, then save its position in the log, and finally process the messages. In this case there is a possibility that the consumer process crashes after saving its position but before saving the output of its message processing. In this case the process that took over processing would start at the saved position even though a few messages prior to that position had not been processed. This corresponds to "at-most-once" semantics as in the case of a consumer failure messages may not be processed.

2. It can read the messages, process the messages, and finally save its position. In this case there is a possibility that the consumer process crashes after processing messages but before saving its position. In this case when the new process takes over the first few messages it receives will already have been processed. This corresponds to the "at-least-once" semantics in the case of consumer failure. In many cases messages have a primary key and so the updates are idempotent (receiving the same message twice just overwrites a record with another copy of itself).

So what about exactly once semantics (i.e. the thing you actually want)? When consuming from a **ADS** topic and producing to another topic, it can leverage the new transactional producer capabilities that were mentioned above. The consumer's position is stored as a message in a topic, so we can write the offset to **ADS** in the same transaction as the output topics receiving the processed data. If the transaction is aborted, the consumer's position will revert to its old value and the produced data on the output topics will not be visible to other consumers, depending on their "isolation level". In the default "read_uncommitted" isolation level, all messages are visible to consumers even if they were part of an aborted transaction, but in "read_committed", the consumer will only return messages from transactions which were committed (and any messages which were not part of a transaction).

When writing to an external system, the limitation is in the need to coordinate the consumer's position with what is actually stored as output. The classic way of achieving this would be to introduce a two-phase commit between the storage of the consumer position and the storage of the consumers output. But this can be handled more simply and generally by letting the consumer store its offset in the same place as its output. This is better because many of the output systems a consumer might want to write to will not support a two-phase commit. As an example of this, consider a **ADS Connect** connector which populates data in **HDFS** along with the offsets of the data it reads so that it is guaranteed that either data and offsets are both updated or neither is. We follow similar patterns for many other data systems which require these stronger semantics and for which the messages do not have a primary key to allow for deduplication.

So effectively **ADS** supports exactly-once delivery, and the transactional producer/consumer can be used generally to provide exactly-once delivery when transfering and processing data between topics. Exactly-once delivery for other destination systems generally requires cooperation with such systems, but **ADS** provides the offset which makes implementing this feasible. Otherwise, **ADS** guarantees at-least-once delivery by default, and allows the user to implement at-most-once delivery by disabling retries on the producer and committing offsets in the consumer prior to processing a batch of messages.

# Глава 7

# Replication

**ADS** replicates the log for each topic's partitions across a configurable number of servers (you can set this replication factor on a topic-by-topic basis). This allows automatic failover to these replicas when a server in the cluster fails so messages remain available in the presence of failures.

Other messaging systems provide some replication-related features, but, in our (totally biased) opinion, this appears to be a tacked-on thing, not heavily used, and with large downsides: slaves are inactive, throughput is heavily impacted, it requires fiddly manual configuration, etc. **ADS** is meant to be used with replication by default – in fact we implement un-replicated topics as replicated topics where the replication factor is one.

The unit of replication is the topic partition. Under non-failure conditions, each partition in **ADS** has a single leader and zero or more followers. The total number of replicas including the leader constitute the replication factor. All reads and writes go to the leader of the partition. Typically, there are many more partitions than brokers and the leaders are evenly distributed among brokers. The logs on the followers are identical to the leader's log – all have the same offsets and messages in the same order (though, of course, at any given time the leader may have a few as-yet unreplicated messages at the end of its log).

Followers consume messages from the leader just as a normal **ADS** consumer would and apply them to their own log. Having the followers pull from the leader has the nice property of allowing the follower to naturally batch together log entries they are applying to their log.

As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive". For **ADS** node liveness has two conditions:

1. A node must be able to maintain its session with **ZooKeeper** (via ZooKeeper's heartbeat mechanism).
2. If it is a slave it must replicate the writes happening on the leader and not fall "too far" behind.

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed". The leader keeps track of the set of "in sync" nodes. If a follower dies, gets stuck, or falls behind, the leader will remove it from the list of in sync replicas. The determination of stuck and lagging replicas is controlled by the *replica.lag.time.max.ms* configuration.

In distributed systems terminology we only attempt to handle a "fail/recover" model of failures where nodes suddenly cease working and then later recover (perhaps without knowing that they have died). **ADS** does not handle so-called "Byzantine" failures in which nodes produce arbitrary or malicious responses (perhaps due to bugs or foul play).

Now can more precisely define that a message is considered committed when all in sync replicas for that partition have applied it to their log. Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails. Producers, on the other hand, have the option of either waiting for the message to be committed or not, depending on their preference for tradeoff between latency and durability. This preference is controlled by the acks setting that the producer uses. Note

that topics have a setting for the "minimum number" of in-sync replicas that is checked when the producer requests acknowledgment that a message has been written to the full set of in-sync replicas. If a less stringent acknowledgement is requested by the producer, then the message can be committed, and consumed, even if the number of in-sync replicas is lower than the minimum (e.g. it can be as low as just the leader).

---

**Important:** The guarantee that ADS offers is that a committed message will not be lost, as long as there is at least one in sync replica alive, at all times

---

**ADS** will remain available in the presence of node failures after a short fail-over period, but may not remain available in the presence of network partitions.

# Глава 8

# Log Compaction

Log compaction ensures that **ADS** will always retain at least the last known value for each message key within the log of data for a single topic partition. It addresses use cases and scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance.

So far we have described only the simpler approach to data retention where old log data is discarded after a fixed period of time or when the log reaches some predetermined size. This works well for temporal event data such as logging where each record stands alone. However an important class of data streams are the log of changes to keyed, mutable data (for example, the changes to a database table).

Let's discuss a concrete example of such a stream. There is a topic containing user email addresses; every time a user updates their email address we send a message to this topic using their user id as the primary key. Now say we send the following messages over some time period for a user with id *123*, each message corresponding to a change in email address (messages for other ids are omitted):

```
1  123 => bill@microsoft.com
2          .
3          .
4          .
5  123 => bill@gatesfoundation.org
6          .
7          .
8          .
9  123 => bill@gmail.com
```

Log compaction gives us a more granular retention mechanism so that **ADS** is guaranteed to retain at least the last update for each primary key (e.g. *bill@gmail.com*). By doing this we guarantee that the log contains a full snapshot of the final value for every key not just keys that changed recently. This means downstream consumers can restore their own state off this topic without us having to retain a complete log of all changes.

Let's start by looking at a few use cases where this is useful:

1. *Database change subscription.* It is often necessary to have a data set in multiple data systems, and often one of these systems is a database of some kind (either a RDBMS or perhaps a new-fangled key-value store). For example you might have a database, a cache, a search cluster, and a **Hadoop** cluster. Each change to the database will need to be reflected in the cache, the search cluster, and eventually in **Hadoop**. In the case that one is only handling the real-time updates you only need recent log. But if you want to be able to reload the cache or restore a failed search node you may need a complete data set.
2. *Event sourcing.* This is a style of application design which co-locates query processing with application design and uses a log of changes as the primary store for the application.

3. *Journaling for high-availability.* A process that does local computation can be made fault-tolerant by logging out changes that it makes to its local state so another process can reload these changes and carry on if it should fail. A concrete example of this is handling counts, aggregations, and other "group by" – like processing in a stream query system. **Samza**, a real-time stream-processing framework, uses this feature for exactly this purpose.

In each of these cases one needs primarily to handle the real-time feed of changes, but occasionally, when a machine crashes or data needs to be re-loaded or re-processed, one needs to do a full load. Log compaction allows feeding both of these use cases off the same backing topic.

The general idea is quite simple. If we had infinite log retention, and we logged each change in the above cases, then we would have captured the state of the system at each time from when it first began. Using this complete log, we could restore to any point in time by replaying the first N records in the log. This hypothetical complete log is not very practical for systems that update a single record many times as the log will grow without bound even for a stable dataset. The simple log retention mechanism which throws away old updates will bound space but the log is no longer a way to restore the current state – now restoring from the beginning of the log no longer recreates the current state as old updates may not be captured at all.

Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each key.

This retention policy can be set per-topic, so a single cluster can have some topics where retention is enforced by size or time and other topics where retention is enforced by compaction.

This functionality is inspired by one of LinkedIn's oldest and most successful pieces of infrastructure – a database changelog caching service called Databus. Unlike most log-structured storage systems **ADS** is built for subscription and organizes data for fast linear reads and writes. Unlike Databus, **ADS** acts as a source-of-truth store so it is useful even in situations where the upstream data source would not otherwise be replayable.

## 8.1 Log Compaction Basics

Here is a high-level picture that shows the logical structure of a **ADS** log with the offset for each message (Pic.8.1.).

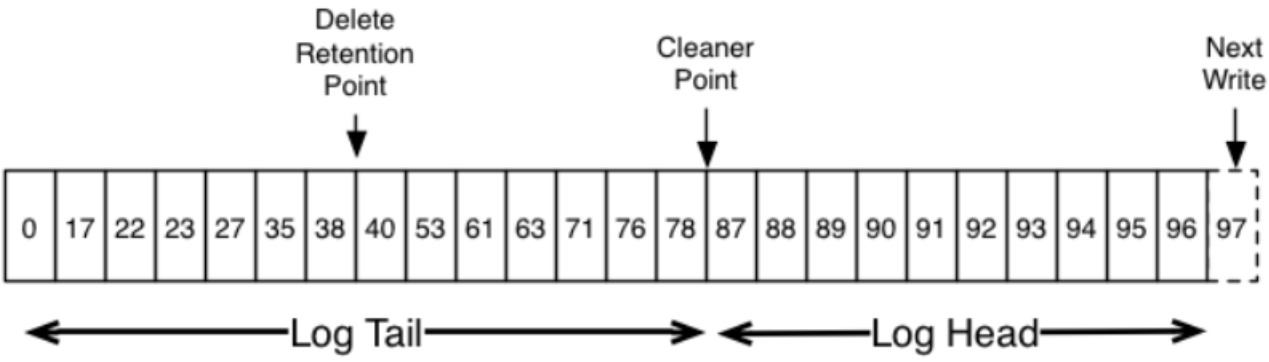



Рис.8.1.: Logical structure of ADS log

The head of the log is identical to a traditional **ADS** log. It has dense, sequential offsets and retains all messages. Log compaction adds an option for handling the tail of the log. The picture above shows a log with a compacted tail. Note that the messages in the tail of the log retain the original offset assigned when they were first written – that never changes. Note also that all offsets remain valid positions in the log, even if the message with that offset has been compacted away; in this case this position is indistinguishable from the next highest offset that does appear in the log. For example, in the picture above the offsets *36*, *37*, and *38* are all equivalent positions and a read beginning at any of these offsets would return a message set beginning with *38*.

Compaction also allows for deletes. A message with a key and a null payload will be treated as a delete from the log. This delete marker will cause any prior message with that key to be removed (as would any new message with that key), but delete markers are special in that they will themselves be cleaned out of the log after a period of time to free up space. The point in time at which deletes are no longer retained is marked as the "Delete Retention Point" in the above diagram.

The compaction is done in the background by periodically recopying log segments. Cleaning does not block reads and can be throttled to use no more than a configurable amount of I/O throughput to avoid impacting producers and consumers. The actual process of compacting a log segment looks something like this (Pic.8.2.):

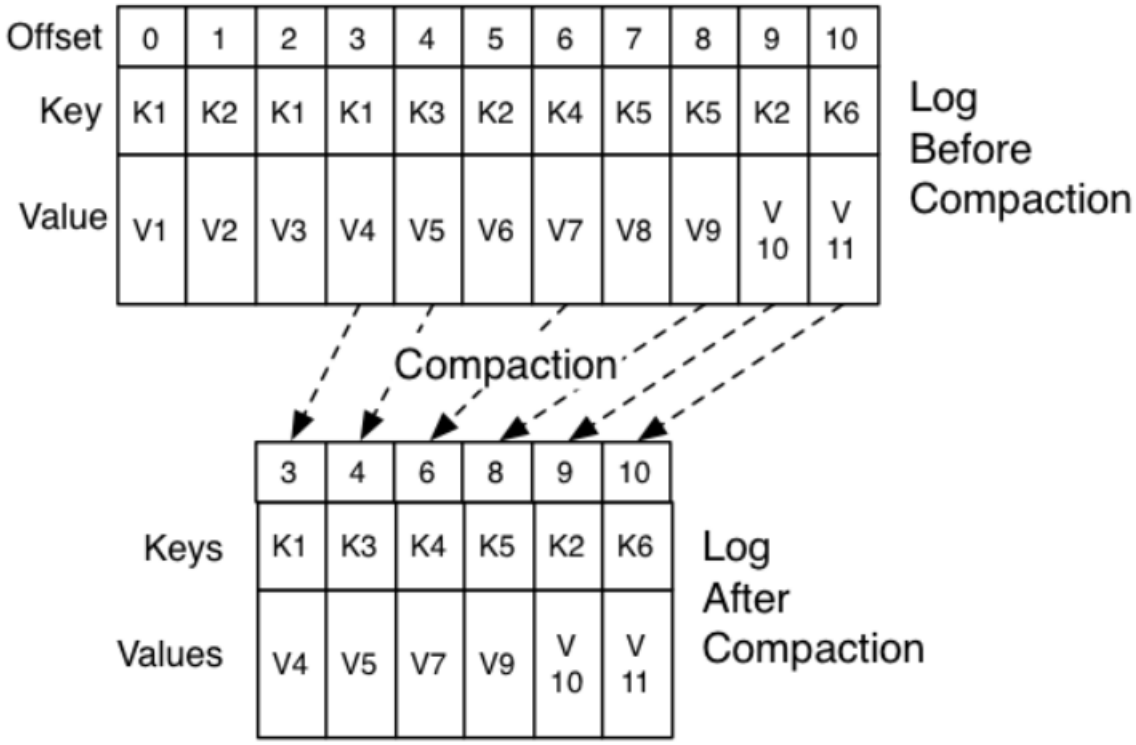



Рис.8.2.: Process of compacting a log segment

## 8.2 What guarantees does log compaction provide?

Log compaction guarantees the following:

1. Any consumer that stays caught-up to within the head of the log will see every message that is written; these messages will have sequential offsets. The topic's *min.compaction.lag.ms* can be used to guarantee the minimum length of time must pass after a message is written before it could be compacted. I.e. it provides a lower bound on how long each message will remain in the (uncompacted) head.
2. Ordering of messages is always maintained. Compaction will never re-order messages, just remove some.
3. The offset for a message never changes. It is the permanent identifier for a position in the log.
4. Any consumer progressing from the start of the log will see at least the final state of all records in the order they were written. Additionally, all delete markers for deleted records will be seen, provided the consumer reaches the head of the log in a time period less than the topic's *delete.retention.ms* setting (the default is *24 hours*). In other words: since the removal of delete markers happens concurrently with reads, it is possible for a consumer to miss delete markers if it lags by more than *delete.retention.ms*.

## 8.3 Log Compaction Details

Log compaction is handled by the log cleaner, a pool of background threads that recopy log segment files, removing records whose key appears in the head of the log. Each compactor thread works as follows:

1. It chooses the log that has the highest ratio of log head to log tail.
2. It creates a succinct summary of the last offset for each key in the head of the log.
3. It recopies the log from beginning to end removing keys which have a later occurrence in the log. New, clean segments are swapped into the log immediately so the additional disk space required is just one additional log segment (not a fully copy of the log).
4. The summary of the log head is essentially just a space-compact hash table. It uses exactly *24 bytes* per entry. As a result with *8GB* of cleaner buffer one cleaner iteration can clean around *366GB* of log head (assuming *1k* messages).