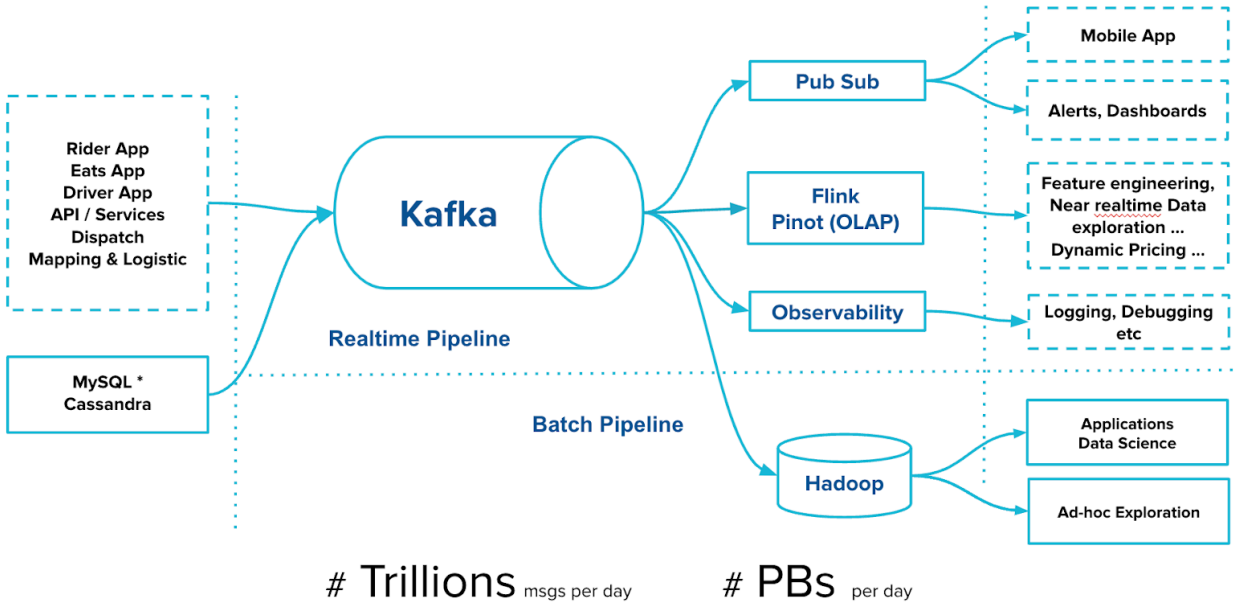
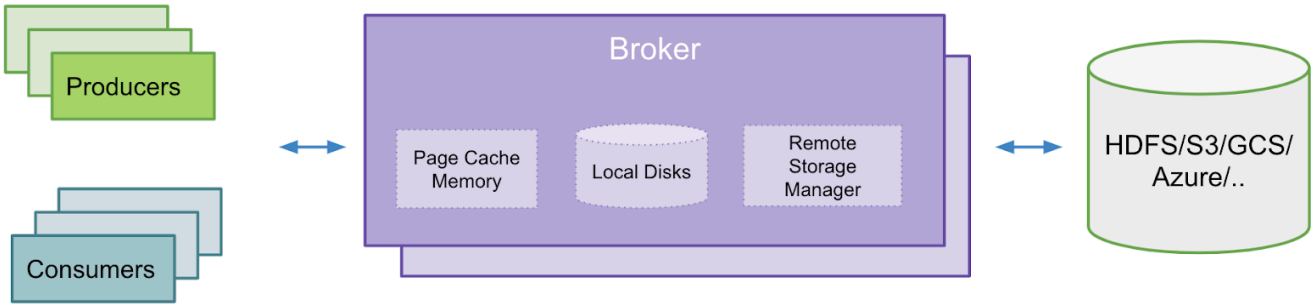
Kafka Tiered Storage at Uber



-
- Kafka stores the msgs in append-only log segments on the broker's local storage.
- Each topic can be configured with the targeted retention based on size or time.
- It gives guarantees for users to consume the data within the retention period or size even when the respective consuming applications fail or become slow for several reasons.
- Total storage on a cluster depends upon factors like the total number of topic partitions, produce throughput, and retention configuration.
- A Kafka broker typically needs to have larger storage to support the required topic partitions hosted on a broker.

Background Behind Project:

- Kafka cluster storage is typically scaled by adding more broker nodes to cluster.
- but this also adds needless memory and CPUs to cluster, making overall storage cost less efficient compared to storing older data in external storage.
- A larger cluster with more nodes also adds to deployment complexity and increases operational costs bcz of tight coupling of storage and processing.
- so, it brings several issues related to scalability, efficiency, and operations.
- We proposed Kafka Tiered Storage to avoid tight coupling of storage and processing in a broker.
- It provides two tiers of storage, called local and remote.
- These two tiers can have respective retention policies based on the respective use cases.



Goals:

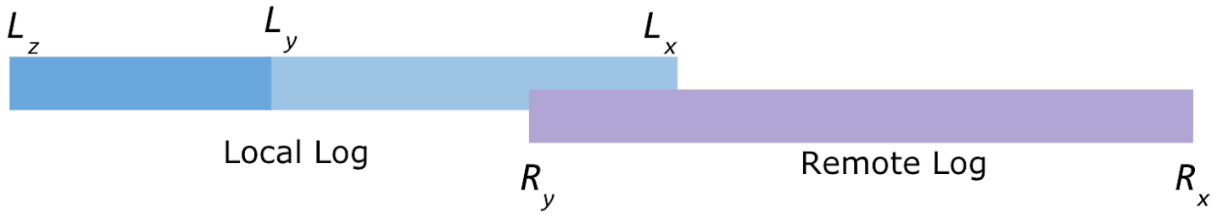
- Extended storage beyond broker
  - Memory/ Page Cache
  - Local Storage
  - Remote storage support (including cloud/object stores like S3/GCS/Azure)
  - Durability and consistency similar to local storage
  - Isolation of reading latest and historical data
  - No changes are required from clients
  - Easy tuning and provisioning of clusters
  - Improve operational and cost efficiency

```
Memory/page cache:
        - is a portion of computer's RAM that stores frequently used data.
        - it helps speed up access to data by storing parts of it in faster-
to-access memory rather than reading from slower disk storage every time.
        - The **memory/page cache** helps with fast access to the **latest
data** stored on Kafka brokers.
```

Architecture:

- Kafka cluster enabled with tiered storage is configured with two tiers of storage called local and remote.
- The local tier is the broker’s local storage where the segments are stored currently.
- The new remote tier is the extended storage, such as HDFS/S3/GCS/Azure.
- Both these tiers will have respective retention configurations based on size and time.
- The retention period for the local tier can be significantly reduced from days to a few hours.
- The retention period for the remote tier can be much longer–days, or even months.

- Applications sensitive to latency conduct tail reads and are catered to from the local tier, utilizing Kafka's efficient page cache utilization for data retrieval.
- On the other hand, applications such as backfill or those recovering from failures requiring older data than what's locally available, are served from the remote tier.
- this approach enables scalability of storage in a kafka cluster without being tied to memory and CPU resources, thus transforming kafka into a viable long-term storage option.
- It decreases local storage burden on kafka brokers, consequently reducing data to be transferred during recovery and rebalancing.
- Log segments accessible in remote tier don't require restoration on broker and can be accessed directly from remote tier.
- It eliminates necessity to expand kafka cluster storage and add new nodes when extending retention period.
- Additionally, It allows for significantly longer data retention without the requirement for separate data pipelines to transfer data from kafka to external storage.
- Tiered storage divides a topic partition's log into 2 different logical components called local log and remote log.
- Local log contains a list of local log segments and remote log contains a list of remote log segments.
- The remote log subsystem copies each topic partition's eligible segments from local storage to remote storage.
- A segment is eligible to be copied when its end offset is less than LastStableOffset of a partition.



Lz = Local log end offset
Lx = Local log start offset
Ly = Last stable offset(LSO)

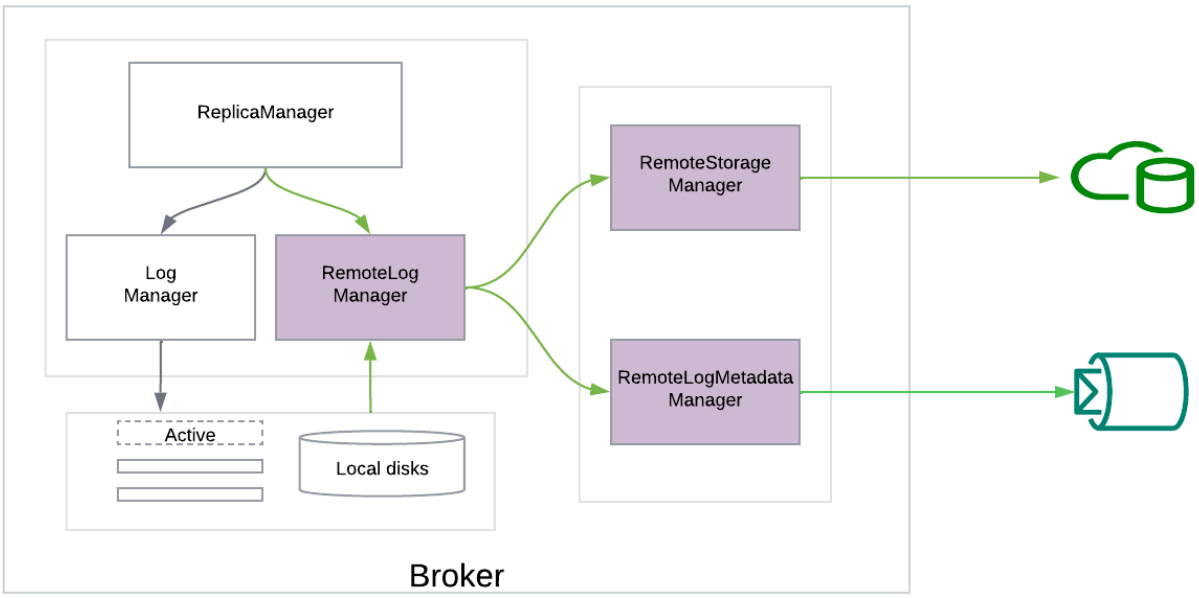
Ry = Remote log end offset
Rx = Remote log start offset

Lz >= Ly >= Lx and Ly >= Ry >= Rx

```
**Eligible for Remote Storage:**
        - A **log segment** from the **local log** is eligible to be moved to
```

```
        **remote storage** when its end offset is **less than the
LastStableOffset (LSO)** of the partition.
        - **LastStableOffset (LSO)** represents the highest offset in the
partition that is
        **guaranteed** to be available and stable (i.e., data that is fully
committed and won't be deleted)
        and msgs before it are safely replicated across all brokers.
```

High Level Architecture of kafka tiered storage components:



- components:
  - RemoteStorageManager
  - RemoteLogMetaDataManager
  - RemoteLogManager

```
Segment:
        - is a file that stores a batch of msgs for a specific topic
partition.
        - kafka organizes msgs into segments for efficient storage and
retrieval.
        - each segment file stores msgs with a specific range of offsets and
the file is stored as a log file on disk.
```

- we introduced 2 pluggable components within storage layer called RemoteStorageManager and RemoteLogMetaDataManager.
- These can be implemented by developers based on their targeted storage systems and plugged into their Kafka env.
- **RemoteStorageManager** interface provides actions for remote log segments that include copy, fetch and delete from remote storage.
- **RemoteLogMetaDataManager** interface provides lifecycle operations of metadata about remote log segments with strongly consistent semantics.
- There is a default implementation that uses an internal topic.
- Users can plugin their implementation if they intend to use another system to store remote log segment metadata.

```
RemoteLogMetaDataManager:
        - keeps track of metadata about remote log segments
        - metadata is just data that describes other data:
                - where the data is stored
                - what the data is
                - when the data can be accessed or deleted
        - It ensures strong consistency for managing metadata,
        means it reliably tracks and updates where each segment is and its
status (whether it's valid or should be deleted)
```

- **RemoteLogManager** is a logical layer responsible for managing life cycle of remote log segments. That includes:
  - Copying segments to remote storage
  - Cleaning up of expired segments in remote storage
  - Fetching the data from remote storage
- It also uses the pluggable remote storage components as and when needed.
- Each remote log segment is identified with a unique identifier called *RemoteLogSegmentId*, even for the same topic partition and offsets.

Copying segments to remote storage:

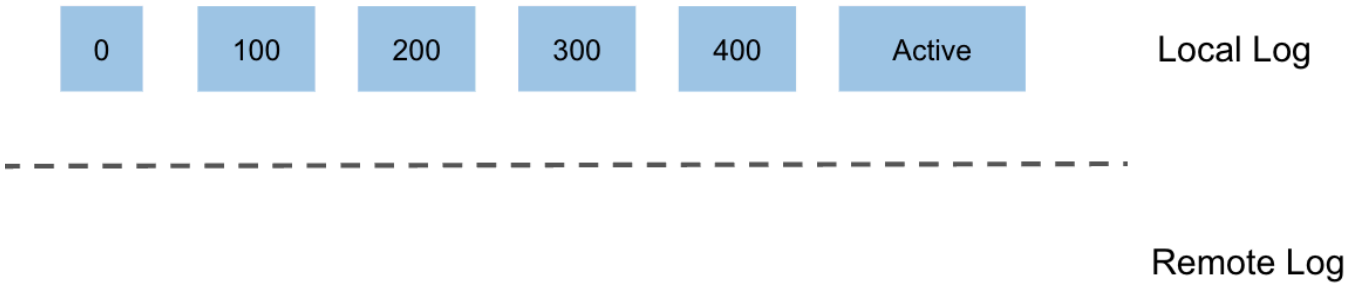
- Each topic partition stores data in a logical log, which contains a sequence of physical log segments with respective auxiliary files like segment indexes, producer state snapshots, and leader epoch checkpoints.
- Each partition replica creates these log segments, flushes them to disk and rolls over segment based on segment roll configurations based on size or time.

```
segment roll over:
        - A log segment is rolled over when it reaches a certain size limit or
a certain time period.
        - once this happens, kafka closes the current segment and starts
writing to a new segment.
```

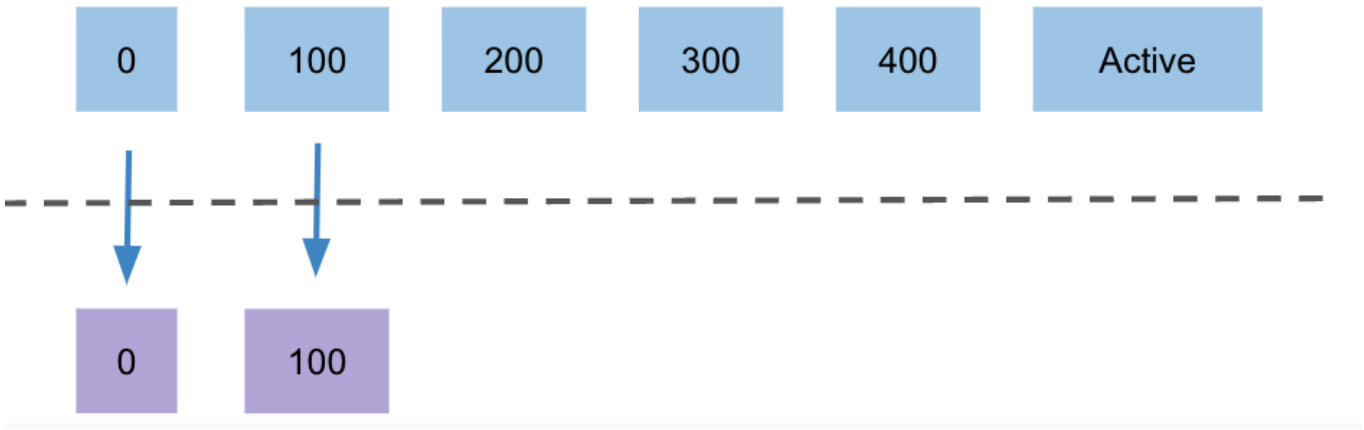
- A log segment is eligible if its end offset is less than the last-stable-offset of partition.
- The broker acting as a leader for topic partition is responsible for copying eligible log segments to remote storage.

```
Leader Broker Responsibility:
        - Leader Broker is responsible for managing log segments for a
partition.
        - It handles writing of new msgs and is responsible for copying older
log segments to remote storage in case of tiered storage.
```

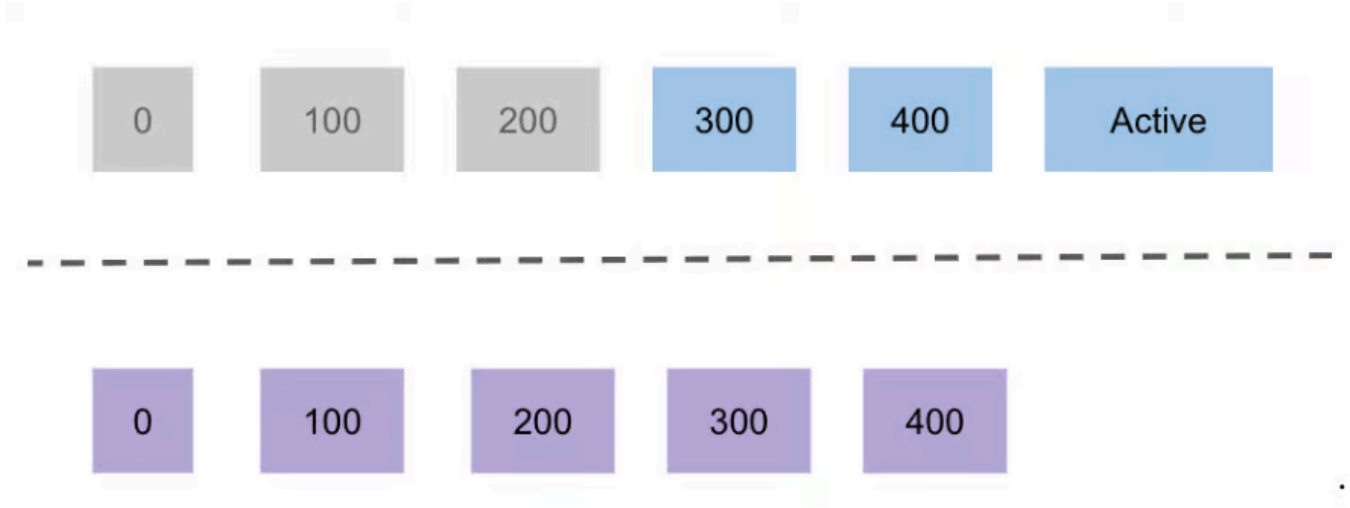
- It copis log segments from earliest segment to latest segment in a sequence.
- It uses *RemoteStorageManager* for copying the segment with its indexes like offset, timestamp, producer snapshot, and its respective leader epoch cache.
- It also adds and updates entries in *RemoteLogMetadataManager* with respective states for each copied segment.



- The above diagram depicts a topic partition's log segments with their respective start offsets. Before tiered storage is enabled, there will not be any segments in the remote storage.



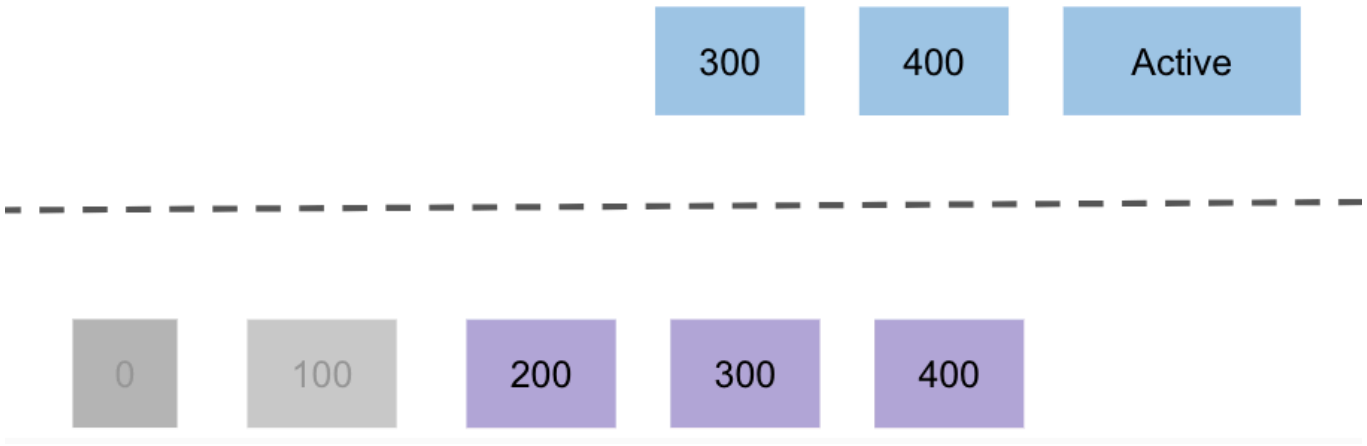
- The above diagram depicts the eligible segments started copying to remote storage after tiered storage is enabled for that topic.



- The above diagram depicts some of the segments in the local storage were deleted based on the local retention configuration. We can see that segments earlier to offset 300 were deleted, but those segments are available in remote storage.

Cleaning up of Remote Segments:

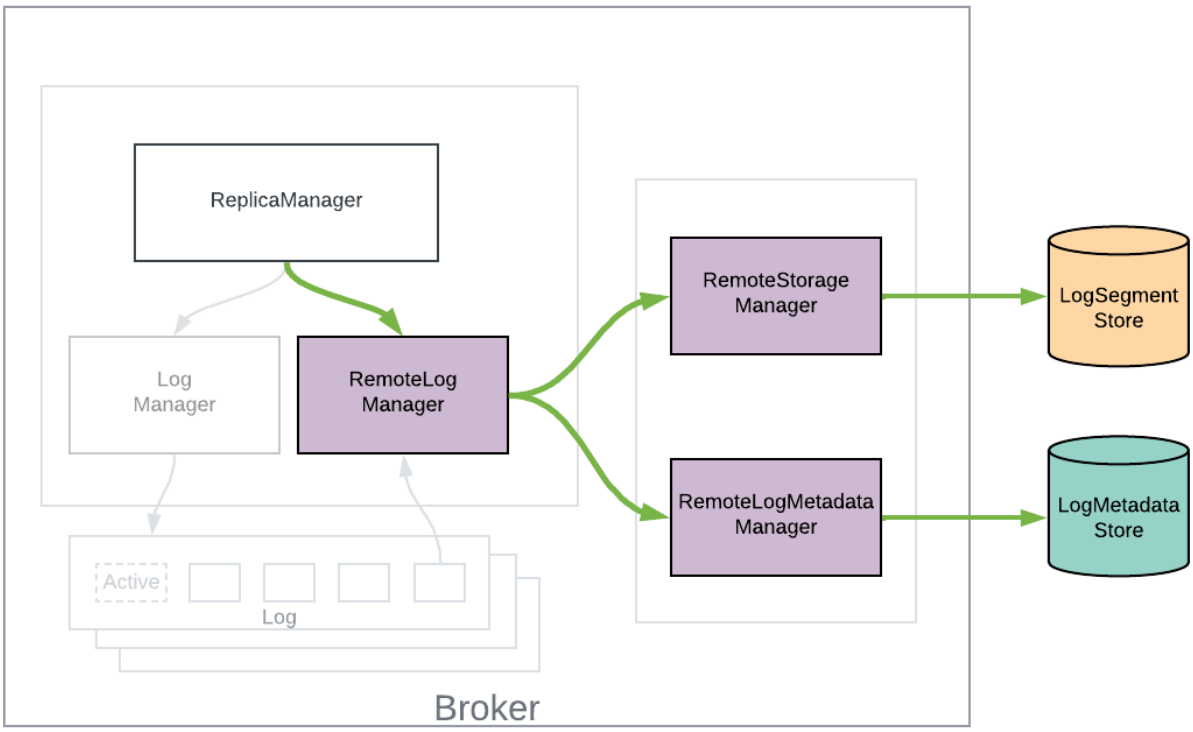
- each topic will have a retention configuration based on size and time for both local data and remote data.
- Remote data is cleaned up at regular intervals by computing the eligible segments by a dedicated thread pool.
- This is different from async cleaning up of local log segments.
- when a topic is deleted, cleaning up of remote log segmenets is done async and it will not block existing delete operation or recreate a new topic.



- The above diagram depicts the cleaning up of remote log segments based on the complete log retention configuration. Here, segments earlier to offset 200 were deleted.

Fetching segments from Remote Storage:

- when a consumer fetch request is received and it is only available in remote storage, then it is served using a dedicated thread pool.
- if targeted offset is available in broker's local storage, then it's served using existing local fetch mechanism.
- So, a broker separates the local reads from remote reads and they will not block each other.
- *RemoteLogManager* determines targeted remote segment based on desired offset and leader epoch by looking into metadata store using *RemoteLogMetadataManager*.
- It uses *RemoteStorageManager* to find the position within the segment and start fetching desired data.



Follower Replication:

- Followers replicate the data from a leader to become an in-sync replica.
- They need to maintain the message lineage across a sequence of log segments as the leader.
- They may also do truncation of the segments if needed to maintain the message ordering.

```
If a follower's log segments become too large, or if the follower needs to
catch up,
it may **truncate** (remove) old segments.
This ensures that the follower doesn't run out of storage and
that it can continue replicating from the leader efficiently.
```

- With tiered storage, follower replicas need to replicate the segments that are available on the leader’s local storage.
- Each log also needs auxiliary data like leader epoch state and producer-ID snapshots.

```
Leader epoch state:
        - keeps track of leader's state and helps in handling leader leader
elections.
Producer ID Snapshots:
```

```
        - this stores info about producers and helps ensure idempotent (no
duplicate) writes.
```

- So, a follower needs to build this auxiliary data before it starts fetching any messages from the leader.
- The follower fetch protocol makes sure to maintain the consistency and ordering of messages across all the replicas irrespective of changes in the cluster like broker replacements, failures, etc.