Task 3 Conceptual questions
Q1. What features does Apache Pulsar support have which the previous distributed data stream framework (e.g., Kafka) does not support?

Separation of storage layer and serving layer

Kafka was designed to colocate the serving layer and storage layer together as a single unit making them to exist within the same node. On the other hand, Pulsar is designed with separation of serving layer and storage layer meaning that they can exist in different nodes making dynamic scalability, zero downtime upgrades, and infinite storage capacity upgrades.

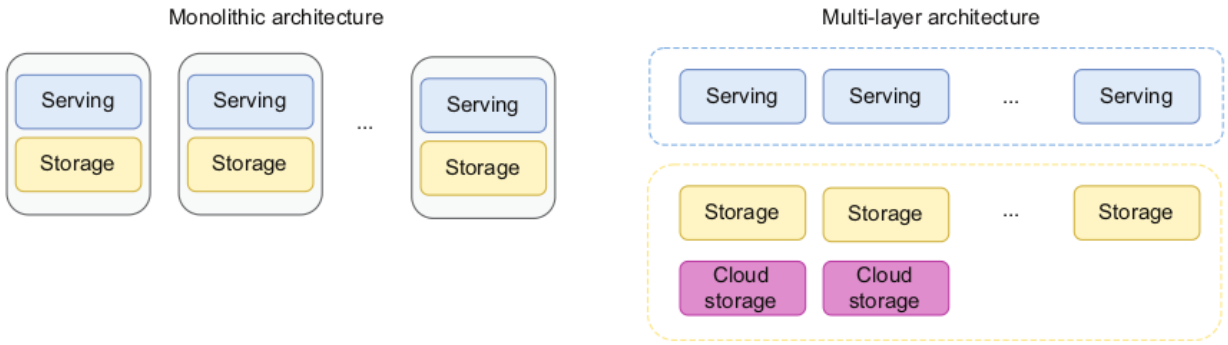


Figure 1.16 Monolithic distributed architectures co-locate the serving and storage layers, while Pulsar uses a multilayer architecture that decouples the storage and serving layers from one another, which allows them to scale independently.

Dynamic scaling on peak usage high traffic

Pulsar's stateless brokers in the serving layer also enable the ability to scale the infrastructure down once the spike has passed. Kafka architecture cannot scale down the nodes due to the fact that the nodes contain data on their attached hard drives. As long as the data hasn't been processed, we cannot remove nodes.Apache Kafka, the broker can only serve requests for data that is stored on an attached disk. This limits the usefulness of autoscaling the cluster in response to traffic spikes, because the newly added nodes to the Kafka cluster will not have any data to serve and, therefore, will not be able to handle any incoming requests to read existing data from the topics. The newly added nodes will only be able to handle write requests.

Auto recovery

Pulsar broker nodes are stateless, they can be replaced by spinning up a new instance of the service to replace the one that failed without a disruption of service or any other data replacement considerations.

segment-based strategy to store message

Kafka is based on partition based storage.
"The way consumption is implemented in Kafka is by dividing up the partitions in the log over the consumer instances so that each instance is the exclusive consumer of a 'fair share' of partitions at any point in time". Each partition within a topic can only have one consumer at a

time, and the partitions are distributed evenly across the consumers within the group. If a consumer group has fewer members than partitions, then some consumers will be assigned to multiple partitions, but if you have more consumers than partitions, the excess consumers will remain idle and only take over in the event of a consumer failure.

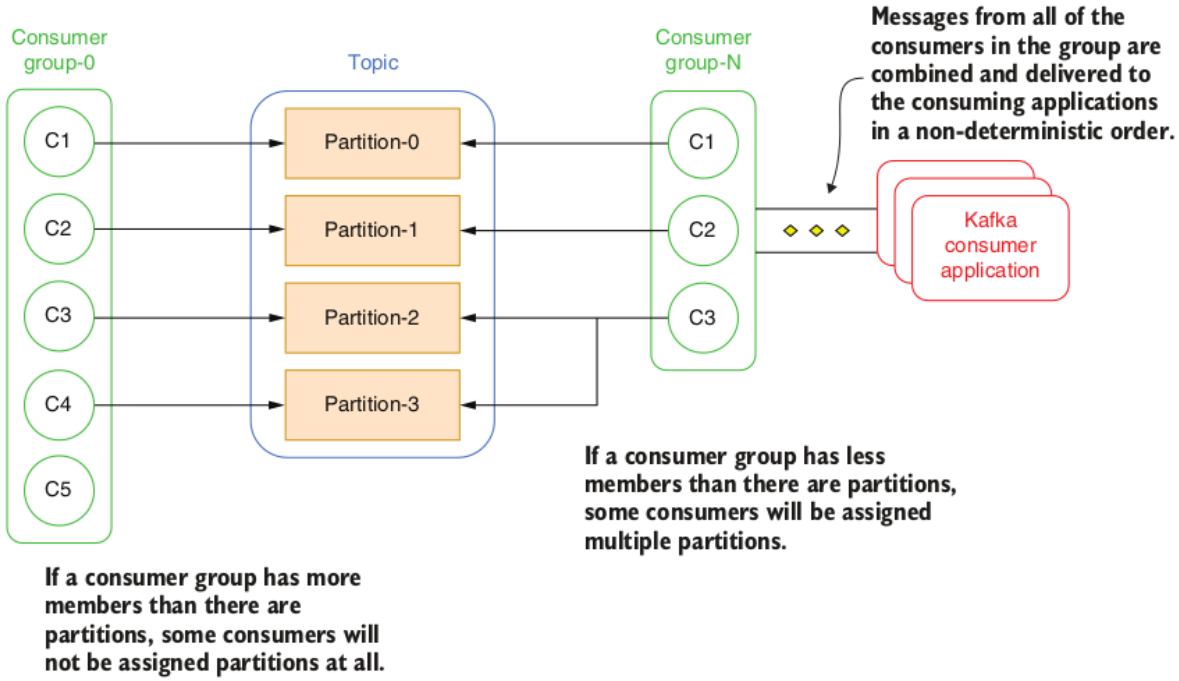


Figure 1.17 Kafka's consumer groups are closely tied to the partition concept. This limits the number of concurrent topic consumers to the number of topic partitions.

Pulsar is based on Segment centric storage in pulsar

https://docs.confluent.io/5.5.5/kafka/introduction.html

Q2. What is the issue with using a batch processing approach on data at scale? How do modern
data stream processing systems such as Apache pulsar can overcome the issue of batch Processing?
Cannot process real time data
Latency

Q3. What is the underlying messaging pattern used by Apache Pulsar? What is the advantage of such a messaging pattern?
Publisher and Subscriber pattern is used by pulsar.
Advantages

- Producers/Publishers can submit messages directly into Topic/Broker without worrying that a Subscriber/consumer is available to receive the message.

- Consumers can be lightweight in terms of resources to consume messages. In the case of large volume of messages, consumers can scale out to meet the demand
- Eliminate Polling Message topics allow instantaneous, push-based delivery, eliminating the need for message consumers to periodically check or “poll” for new information and updates. This promotes faster response time and reduces the delivery latency that can be particularly problematic in systems where delays cannot be tolerated.

Q.4 What are different modes of subscription? When are each modes of subscription used?
Different modes of subscription

- Exclusive :: It is used when you want to ensure that each message is processed exactly once and by a known consumer.
- Failover :: It is useful if you want your application to continue processing messages in the event of a system failure and another consumer to take over if the first consumer were to fail for any reason.
- Shared :: It is useful for implementing work queues, where message ordering isn’t important and distribution of messages is in round robin fashion, as it allows you to scale up the number of consumers on the topic quickly to process the incoming messages and but high throughput is
- Key-shared :: Each consumer within the subscription receives only a portion of the messages published to a topic.

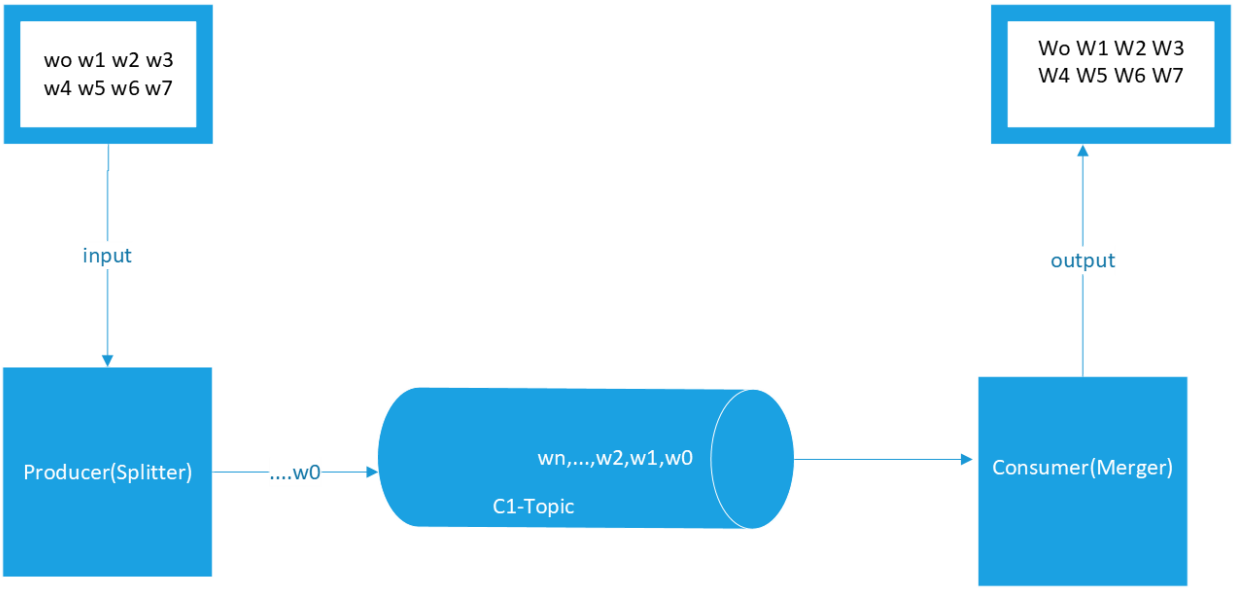
Q.5 What is the role of the ZooKeeper? Is it possible to ensure reliability in streaming frameworks such as Pulsar or Kafka without ZooKeeper?
Zookeeper helps to keep the track of configuration information, naming, provide distributed synchronization to bookkeeper and pulsar.
In a distributed system, services need to agree on different pieces of information, such as the current configuration values and the owner of a topic. This is a problem particularly for distributed systems due the fact that there are multiple copies of the same component running concurrently with no real way to coordinate information between them. Using databases will introduce a serialization point meaning all the calling services would be blocked waiting for the same lock on a table, which essentially eliminates all the benefits of distributed computing.
Example: checking that the state of a BookKeeper ledger is open before writing any data to it. If some other process has closed the ledger, it would be reflected in the ZooKeeper data, and the process would know not to proceed with the write operation. Conversely, if a process were to close a ledger, this information would be sent to ZooKeeper so that it could be propagated to the other services, so they would know it is closed before they attempted to write to it.

4.1. Identify an issue with the current implementation in terms of handling big data i.e., words are in the order of millions.
The current implementation is designed to work with a single node/process resulting in the following issues.

- Single point of failure

- Scaling is achieved by scaling up instead of scaling out

4.2. Redesign the current implementation by using apache pulsar to demonstrate splitting and merging of data i.e., how the same operation on each word (i.e., splitting) and the resultant string (i.e., merging) can be handled by consumers and producers. Provide a diagram to demonstrate how your architecture will look like i.e.,
a. How data splitting and merging is handled by consumer and producer?
Producer is designed to do splitting operations within a single sentence. After sentence has been splitted by white-space and producer sends the length of words to topic named “C1-Topic” And then it will keep on sending all the words to the same topic Synchronous fashion.
Consumer is designed to do merge operations to get back the identical sentence used by the Producer. Consumers collect the first message from topic “C1-Topic” which represents the number of words. Based on the length of words, the consumer keeps on receiving each word using proper decode followed with upper case transformation.
b. Label broker, consumer and producer.



4.3. Provide an implementation (preferably in Python programming language) of your architecture with Apache Pulsar. You can set up your architecture on a single virtual machine by creating different sessions for each consumer and producer.
I have included source code producer.py and consumer.py within the zip file.
I have tested on multiple machines.