Department of Artificial Intelligence and Machine Learning

Date:	**Test – 2**	**Max. Marks: 10 + 50**
Semester: VII	**UG**	**Duration: 2 Hrs.**
Course Title: Stream Processing and Analytics		**Course Code: AI372TA**

Common to AIML and CSE (Data Science)

PART A

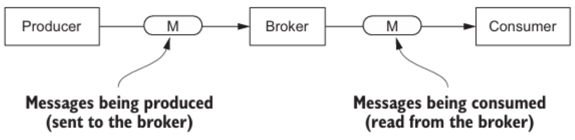
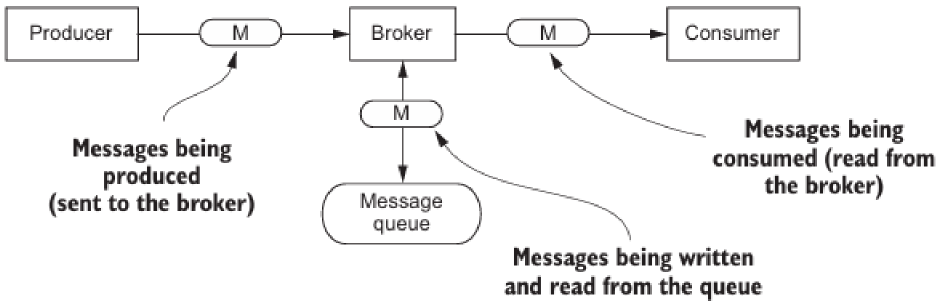
Scheme and Solution

S. No	Questions	M	BT	CO
1	List the three message delivery semantics? At most once—A message may get lost, but it will never be reread by a consumer. At least once—A message will never be lost, but it may be reread by a consumer. Exactly-once—A message is never lost and is read by a consumer once and only once	2	2	2
2	Discuss the methods to handle broker crash in a stream processing environment? Methods to handle : (i) to guarantee the message was delivered, our producer should wait for an acknowledgment that the data was indeed written to disk. (ii) have the message queuing product replicate the message to more than one broker You can configure the broker to hold as little data in memory as possible	2	2	2
3	Justify Kafka as an effective data pipe line during streaming data The main value Kafka provides to data pipelines is its ability to serve as a very large, reliable buffer between various stages in the pipeline. This effectively decouples producers and consumers of data within the pipeline and allows use of the same data from the source in multiple target applications and systems, all with different timeliness and availability requirements. This decoupling, combined with reliability, security, and efficiency, makes Kafka a good fit for most data pipelines.	**2**	**2**	**2**
4	What are the two sources of kafka connect? Two types of connectors: o Source Connector: Pulls data from external systems into Kafka. o Sink Connector: Sends data from Kafka to external systems.	2	2	2

5	Define : (i) Regex Router (ii) Time Stamp Router RegexRouter : Change the destination topic using a regular expression and a replacement string TimestampRouter : Modify the topic based on the message timestamp.			

PART B

S. No	Questions	M	BT	CO
1a.	Elaborate the need for message queueing tier in stream processing? Buffering Data: • It temporarily holds events or messages that are produced by one component (e.g., a data producer, sensor, or log generator) before they are consumed by another component (e.g., stream processors or consumers). • This ensures that the system can handle spikes in event production without overwhelming the downstream processing components. Decoupling Producers and Consumers: • The message queue provides a layer of abstraction between the producer (source of events) and consumer (e.g., processing engine, database, or another service). • This decoupling allows each component to operate independently, improving scalability and fault tolerance. Event Ordering and Delivery Guarantees: • In stream processing, message queues often provide mechanisms to ensure that events are processed in the correct order (e.g., FIFO—First In, First Out) and that no events are lost. • Some message queues also provide guarantees for at-least-once, exactly-once, or at-most-once delivery semantics, depending on the use case. Flow Control: • The message queue can help manage backpressure when downstream components are unable to keep up with the rate of incoming messages. • It can buffer the excess data and ensure that the consumers are not overwhelmed. Fault Tolerance: • If a component in the stream processing pipeline fails, the message queue can store the messages temporarily. • This ensures that no data is lost, and once the system recovers, processing can resume without data loss.	6	03	03
1b.	Discuss the key aspects of durable messaging **Persistence:** Messages are stored in a persistent storage medium, such as a database or disk. This ensures that even if the system crashes, the messages are not lost. **Message Queues:** In messaging systems like RabbitMQ, Apache Kafka, or JMS (Java Message Service), messages can be marked as durable so they are stored until they are successfully consumed by the receiver.	04	2	2

	Fault Tolerance: Durable messages offer resilience against failures. If a messaging system or its components (e.g., servers) fail, the messages will be retried or preserved for later processing. **Acknowledgment:** The system typically uses acknowledgments to confirm that a message has been processed. Until the acknowledgment is received, the system might hold onto the message to prevent loss.			
2a.	With a neat diagram, elaborate the working of the producer, the broker, and the consumer with a message queuing layer  Producer M Broker M Consumer Messages being produced (sent to the broker) Messages being consumed (read from the broker) Figure 3.3 The three core parts to a message queuing system ❑ The producer sends a message to a broker ❑ The broker puts the message into a queue ❑ The consumer reads the message from the broker Broker with queue system # Student should explain all the concepts in detail  Producer M Broker M Consumer M Message queue Messages being produced (sent to the broker) Messages being consumed (read from the broker) Messages being written and read from the queue	10	2	2
3	Consider a Real-Time Fraud Detection in Online Transactions. If a data pipeline is to be built using kafka, list and elaborate the different considerations during the implementation . #Student should elaborate all the following points ● Scalability ● Reliability & Fault Tolerance ● Data Format & Schema Evolution ● Timeliness ● Transformations: ETL vs. ELT ● Access Control ● Failure Handling and Coupling	10	3	2
4	With a neat diagram, elaborate how kafka handles the table-table and streaming joins? Give example **Table – Table join** Joining two tables is always nonwindowed and joins the current state of both tables at the time the operation is performed. With Kafka Streams, we can perform an equi-join, in which both tables have the same key that is partitioned in the same way, and therefore the join operation can be efficiently distributed between a large number of application instances and	10	03	03

	machines. Kafka Streams also supports foreign-key join of two tables—the key of one stream or table is joined with an arbitrary field from another stream or table. Used when both inputs are **KTables** (materialized tables). **Characteristics** • Represents a join between **latest states**. • Supports: **INNER**, **LEFT**, **OUTER**. **Streaming Join** when we join two streams, we are joining the entire history, trying to match events in one stream with events in the other stream that have the same key and happened in the same time windows. This is why a streaming join is also called a windowed join. Used when one input is an event stream, and the other is a **lookup table (KTable)**. **Characteristics** • Table always holds **latest value per key**. • **No windowing** needed. • Supports: **INNER**, **LEFT**.			

5.	An e-commerce platform needs to process customer orders asynchronously to ensure scalability. Instead of directly handling orders in a monolithic database, the system uses Kafka to:
	• Allow multiple services (inventory, payment, shipping) to consume order events independently.
	• Ensure fault-tolerant processing with retry mechanisms.
	Implement kafka producer and consumer for the given scenario using appropriate code.
	`# Student can give the code in python`
	`from kafka import KafkaProducer`
	`import json`
	`# Create Kafka producer`
	`producer = KafkaProducer(`
	`    bootstrap_servers='localhost:9092',`
	`    value_serializer=lambda v: json.dumps(v).encode('utf-8')`
	`)`
	
	`# Example order data`
	`order_data = {`
	`    "order_id": "ORD123",`
	`    "user_id": "U456",`
	`    "product": "Laptop",`
	`    "quantity": 1,`
	`    "price": 1200.00,`
	`    "timestamp": "2025-02-17T12:00:00Z"`
	`}`
	`# Send data to Kafka topic`
	`producer.send('orders_topic', order_data)`
	`producer.flush()`
	`print(f"Order {order_data['order_id']} sent to Kafka!")`
	`kafka import KafkaConsumer`
	`import json`
	
	`# Create Kafka consumer`
		10	03	03

```
consumer = KafkaConsumer(
    'orders_topic',
    bootstrap_servers='localhost:9092',
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    value_deserializer=lambda v: json.loads(v.decode('utf-8'))
)

print("Listening for new orders...")

# Consume messages
for message in consumer:
    order = message.value
    print(f"Processing Order ID: {order['order_id']}, Product: {order['product']}, Price: ${order['price']}")

    # Simulate processing (e.g., update inventory, charge payment)
    print(f"Order {order['order_id']} processed successfully!")
```