1.) MySQL Table (Table should have some column like created_at or updated_at so that can be used for incremental read)

Answer:

We can use the AUTO_INCREMENT attribute for the primary key column and the TIMESTAMP data type for the created_at and updated_at columns. You can also use the ON UPDATE CURRENT_TIMESTAMP clause to automatically update the updated_at column whenever a row is changed.

For example, you can create a table like this:

```
CREATE TABLE products (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  price DECIMAL(10,2) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

Copy

To perform an incremental read, you can use the LAST_INSERT_ID() function to get the most recent auto-incremented value34, or use the created_at or updated_at columns to filter by date range15.

For example, you can query the table like this:

```
-- Get all rows with id greater than the last imported value
SELECT * FROM products WHERE id > LAST_INSERT_ID();

-- Get all rows with created_at or updated_at within a date range
SELECT * FROM products WHERE created_at BETWEEN '2021-01-01' AND '2021-01-31';
SELECT * FROM products WHERE updated_at BETWEEN '2021-01-01' AND '2021-01-31';
```

2.) Write a python script which is running in infinite loop and inserting 4-5 dummy/dynamically prepared records in MySQL Table

Answer:

```
# Import the modules
import mysql.connector
import random
import time

# Connect to the MySQL database
db = mysql.connector.connect(
  host="localhost",
  user="root",
  password="password",
  database="test"
)

# Create a cursor object
cursor = db.cursor()

# Define a function to generate a random product name
def generate_name():
  # A list of possible prefixes and suffixes
  prefixes = ["Super", "Mega", "Ultra", "Awesome", "Cool", "Amazing"]
  suffixes = ["Phone", "Laptop", "Tablet", "Watch", "TV", "Camera"]
  # Choose a random prefix and suffix
  prefix = random.choice(prefixes)
  suffix = random.choice(suffixes)
  # Return the concatenated name
  return prefix + suffix

# Define a function to generate a random product price
def generate_price():
  # A range of possible prices
```

```
  min_price = 100.00
  max_price = 1000.00
  # Return a random price with two decimal places
  return round(random.uniform(min_price, max_price), 2)

# Define a function to insert a product into the database
def insert_product():
  # Generate a random name and price
  name = generate_name()
  price = generate_price()
  # Prepare the SQL statement
  sql = "INSERT INTO products (name, price) VALUES (%s, %s)"
  # Execute the SQL statement with the values
  cursor.execute(sql, (name, price))
  # Commit the changes to the database
  db.commit()
  # Print a message with the inserted product details
  print(f"Inserted product: {name}, {price}")

# Run an infinite loop
while True:
  # Insert four or five products randomly
  num_products = random.randint(4,5)
  for i in range(num_products):
    insert_product()
  # Wait for one second before repeating the loop
  time.sleep(1)
```

3.) Setup Confluent Kafka

Confluent Kafka is a platform that provides a distribution of Apache Kafka with additional features and tools. To set up Confluent Kafka, you need to follow different steps depending on your operating system and environment.

If you are using Windows 10, you can install Confluent Kafka on the Windows Subsystem for Linux 2 (WSL 2) by following these steps1:

Install Windows Subsystem for Linux 2 and Ubuntu 20.04 LTS

Download the Confluent package from https://www.confluent.io/download/

Set the CONFLUENT_HOME environment variable to the path where you extracted the package

Install the Datagen connector using the confluent-hub command

Start the Confluent components using the confluent local command

Use the Confluent CLI or the Control Center web interface to interact with Confluent Kafka

If you are using Linux or macOS, you can install Confluent Kafka from platform packages or individual component packages by following these steps2:

Download the packages from https://www.confluent.io/download/

Install the packages using your package manager (such as apt, yum, or brew)

Start the Confluent components using the confluent local command or systemd services

Use the Confluent CLI or the Control Center web interface to interact with Confluent Kafka

If you are using a cloud service, such as Pivotal Container Service (PKS) or Confluent Cloud, you can deploy Confluent Kafka using Helm charts or web console by following these steps34:

Create a Kubernetes cluster on your cloud provider

Install Helm and configure it to access your cluster

Download the Confluent Operator Helm charts from https://github.com/confluentinc/cp-helm-charts

Install Kafka and other components using helm install commands with appropriate values files

Use kubectl commands or the Control Center web interface to interact with Confluent Kafka

4.) Create Topic

Answer:

To create a topic in Confluent Kafka, we can use the kafka-topics command-line tool or the Control Center web interface.

Using the kafka-topics tool, you can run a command like this:

kafka-topics --bootstrap-server localhost:9092 --create --topic my-topic --partitions 3 --replication-factor 1

Copy

This will create a topic named my-topic with 3 partitions and 1 replication factor. You can adjust these parameters according to your needs.

Using the Control Center web interface, you can follow these steps:

Open your browser and go to http://localhost:9021

Click on the Topics tab on the left sidebar

Click on the Add a topic button on the top right corner

Enter the topic name, partitions, and replication factor in the fields

Click on the Create with defaults button

This will create a topic with the same parameters as above. You can also customize other settings such as retention, compression, and cleanup policy.

5.) Create json schema on schema registry (depends on what kind of data you are publishing in mysql table)

Answer:

To create a JSON schema on the schema registry, we need to use the JSON Schema serializer and deserializer provided by Confluent. These are classes that implement the Serializer and Deserializer interfaces of the Apache Kafka® Java client and can be used with the Kafka producer and consumer1.

To use the JSON Schema serializer and deserializer, we need to follow these steps:

Add the kafka-json-schema-serializer dependency to your project

Configure your producer and consumer to use the JsonSchemaSerializer and JsonSchemaDeserializer classes

Annotate your Java classes with the @Schema annotation to specify the schema

Use the KafkaJsonSchemaSerde class to serialize and deserialize your objects

Alternatively, you can also use the kafka-json-schema-console-producer and kafka-json-schema-console-consumer tools to produce and consume JSON messages with schemas1.

The schema registry supports different compatibility modes for JSON schemas, such as BACKWARD, FORWARD, FULL, or NONE. You can set the compatibility mode for each subject (topic) or globally using the REST API or the Control Center web interface2.

6.) Write a producer code which will read the data from MySQL table incrementally (hint : use and maintain create_at column)

Answer:

To write a producer code that will read the data from MySQL table incrementally, you can use the JDBC library to connect to the database and execute queries, and the Kafka library to create a producer and send messages to a topic. You can also use the JSON Schema serializer to encode your messages with schemas.

Here is an example of a Java class that implements this logic:

```
import io.confluent.kafka.serializers.KafkaJsonSchemaSerializer;

import io.confluent.kafka.serializers.json.KafkaJsonSchemaSerde;

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.clients.producer.ProducerRecord;


import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.time.LocalDateTime;
```

```
import java.util.Properties;

// A class that represents a record from the MySQL table
@io.confluent.kafka.schemaregistry.annotations.Schema
public class Record {
   private int id;
   private String name;
   private LocalDateTime created_at;

   // Getters and setters omitted for brevity
}

public class Producer {

   // The topic to send messages to
   public static final String TOPIC = "my-topic";

   // The schema registry URL
   public static final String SCHEMA_REGISTRY_URL = "http://localhost:8081";

   // The JDBC connection URL
   public static final String JDBC_URL = "jdbc:mysql://localhost:3306/mydb";

   // The JDBC username and password
   public static final String JDBC_USER = "user";
   public static final String JDBC_PASSWORD = "password";

   // The query to select records from the MySQL table incrementally
   public static final String QUERY = "SELECT id, name, created_at FROM mytable WHERE created_at
> ? ORDER BY created_at";
```

```
// The last timestamp of the records fetched
private LocalDateTime lastTimestamp;

// The Kafka producer
private KafkaProducer<String, Record> producer;

// The JDBC connection
private Connection connection;

// The constructor that initializes the producer and the connection
public Producer() throws Exception {
   // Create the Kafka producer properties
   Properties props = new Properties();
   props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
   props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, KafkaJsonSchemaSerializer.class);
   props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaJsonSchemaSerializer.class);
   props.put("schema.registry.url", SCHEMA_REGISTRY_URL);

   // Create the Kafka producer
   producer = new KafkaProducer<>(props);

   // Create the JDBC connection
   connection = DriverManager.getConnection(JDBC_URL, JDBC_USER, JDBC_PASSWORD);

   // Initialize the last timestamp to null
   lastTimestamp = null;
}

// A method that reads data from MySQL table incrementally and sends it to Kafka topic
public void produce() throws Exception {
   // Prepare the query statement with the last timestamp as a parameter
```

```
        PreparedStatement statement = connection.prepareStatement(QUERY);
        statement.setTimestamp(1, lastTimestamp == null ? null :
java.sql.Timestamp.valueOf(lastTimestamp));

        // Execute the query and get the result set
        ResultSet resultSet = statement.executeQuery();

        // Loop through the result set and create records
        while (resultSet.next()) {
            // Create a record object from the result set fields
            Record record = new Record();
            record.setId(resultSet.getInt("id"));
            record.setName(resultSet.getString("name"));
            record.setCreated_at(resultSet.getTimestamp("created_at").toLocalDateTime());

            // Create a producer record with the record id as key and the record object as value
            ProducerRecord<String, Record> producerRecord = new ProducerRecord<>(TOPIC,
String.valueOf(record.getId()), record);

            // Send the producer record to Kafka topic asynchronously
            producer.send(producerRecord, (metadata, exception) -> {
                if (exception == null) {
                    System.out.println("Sent record: " + record);
                } else {
                    System.err.println("Failed to send record: " + exception.getMessage());
                }
            });

            // Update the last timestamp with the record created_at field
            lastTimestamp = record.getCreated_at();
        }
```

```
        // Close the result set and the statement
        resultSet.close();
        statement.close();
    }

    // A method that closes the producer and the connection
    public void close() throws Exception {
        // Flush and close the producer
        producer.flush();
        producer.close();

        // Close the connection
        connection.close();
    }

}
```

7.) Producer will publish data in Kafka Topic

Answer:

The producer will publish data in Kafka topic using the producer.send() method. This method takes a ProducerRecord object as an argument, which contains the topic name, the key, and the value of the message. The key and the value are serialized using the JsonSchemaSerializer class, which encodes them with schemas from the schema registry. The producer.send() method returns a Future object, which can be used to get the metadata or the exception of the send operation. You can also provide a Callback object as a second argument, which will be invoked when the send operation completes. This way, you can handle the success or failure of the message delivery asynchronously.

8.) Write consumer group to consume data from Kafka topic

Answer:

To write a consumer group to consume data from Kafka topic, you can use the Kafka library to create a consumer and subscribe to a topic. You can also use the JSON Schema deserializer to decode your messages with schemas.

Here is an example of a Java class that implements this logic:

```
import io.confluent.kafka.serializers.KafkaJsonSchemaDeserializer;
import io.confluent.kafka.serializers.json.KafkaJsonSchemaSerde;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

// A class that represents a record from the Kafka topic
@io.confluent.kafka.schemaregistry.annotations.Schema
public class Record {
   private int id;
   private String name;
   private LocalDateTime created_at;

   // Getters and setters omitted for brevity
}

public class Consumer {

   // The topic to consume messages from
   public static final String TOPIC = "my-topic";
```

```
    // The schema registry URL
    public static final String SCHEMA_REGISTRY_URL = "http://localhost:8081";

    // The consumer group ID
    public static final String GROUP_ID = "my-group";

    // The Kafka consumer
    private KafkaConsumer<String, Record> consumer;

    // The constructor that initializes the consumer
    public Consumer() throws Exception {
        // Create the Kafka consumer properties
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaDeserializer.class);
        props.put("schema.registry.url", SCHEMA_REGISTRY_URL);

        // Create the Kafka consumer
        consumer = new KafkaConsumer<>(props);

        // Subscribe to the topic
        consumer.subscribe(Collections.singleton(TOPIC));
    }

    // A method that consumes data from Kafka topic and prints it to the console
    public void consume() throws Exception {
        // Poll for records from the topic with a timeout of 1 second
        ConsumerRecords<String, Record> records = consumer.poll(Duration.ofSeconds(1));
```

```
        // Loop through the records and print them to the console
        for (ConsumerRecord<String, Record> record : records) {
            // Get the key and the value of the record
            String key = record.key();
            Record value = record.value();

            // Print the record details to the console
            System.out.println("Consumed record: key = " + key + ", value = " + value);
        }

        // Commit the offsets of the records
        consumer.commitSync();
    }

    // A method that closes the consumer
    public void close() throws Exception {
        // Close the consumer
        consumer.close();
    }

}
```

9.) In Kafka consumer code do some changes or transformation for each record and write it in Cassandra table

To do some changes or transformation for each record and write it in Cassandra table, you can use the DataStax Java Driver for Apache Cassandra123. This driver provides an asynchronous API to interact with Cassandra using CQL statements. You can also use the QueryBuilder class to construct CQL statements programmatically2.

Here is an example of how you can modify the consume() method of the previous code to insert each record into a Cassandra table after applying some transformation:

```
// A method that consumes data from Kafka topic and writes it to Cassandra table
public void consume() throws Exception {
   // Poll for records from the topic with a timeout of 1 second
   ConsumerRecords<String, Record> records = consumer.poll(Duration.ofSeconds(1));

   // Loop through the records and write them to Cassandra table
   for (ConsumerRecord<String, Record> record : records) {
      // Get the key and the value of the record
      String key = record.key();
      Record value = record.value();

      // Apply some transformation to the value, for example, change the name to upper case
      value.setName(value.getName().toUpperCase());

      // Create a CQL statement to insert the record into a Cassandra table
      Statement statement = QueryBuilder.insertInto("my_keyspace", "my_table")
            .value("id", value.getId())
            .value("name", value.getName())
            .value("created_at", value.getCreated_at());

      // Execute the statement asynchronously using the Cassandra session
      session.executeAsync(statement);
   }

   // Commit the offsets of the records
   consumer.commitSync();
}
```

Copy

This is how you can use the DataStax Java Driver to insert data into Cassandra table from Kafka topic.