Garderobe

TAP PROJECT:
Behavioral Analysis for E-Commerce

Capobianco Francesco Massimo

Indice Relazione del Progetto:

Introduzione e Obiettivi del progetto

Descrizione caso di studio: E-Commerce “Garderobe”

Il progetto simula un ambiente di e-commerce reale denominato “Garderobe”. In uno scenario di mercato moderno, un negozio online genera una mole continua di dati eterogenei: log di sistema, transazioni finanziarie, interazioni degli utenti e aggiornamenti di inventario. **La gestione e la comprensione di questi dati** sono cruciali per la competitività aziendale.

Il Problema: Analisi dei dati in tempo reale vs Batch Processing

Tradizionalmente, l'analisi dei dati avviene tramite **processi Batch** (ad esempio, report generati ogni notte). Questo approccio presenta un limite critico: la latenza. Sapere che un prodotto è andato esaurito o che una promozione non funziona con 24 ore di ritardo comporta perdite economiche.
La soluzione da adottare è **l'approccio Real-Time Streaming**: i dati vengono elaborati istantaneamente non appena vengono generati, permettendo decisioni immediate.

L’Obiettivo: Monitoraggio KPI e Profilazione Utenti

L'obiettivo principale è costruire una pipeline completa che trasformi il dato grezzo in informazione di valore. Nello specifico:
1. **Monitoraggio KPI**: Visualizzare metriche finanziarie (fatturato, vendite per prodotto).
2. **Profilazione (Machine Learning):** Utilizzare l'Intelligenza Artificiale per classificare il comportamento dell'utente (es. acquisto Impulsivo vs Ragionato) basandosi sulla durata della sessione.

Architettura del Sistema

Panoramica della pipeline TAP (Transazione, Analisi, Presentazione)

L'architettura è caratterizzata dalla pipeline TAP, che prevede:

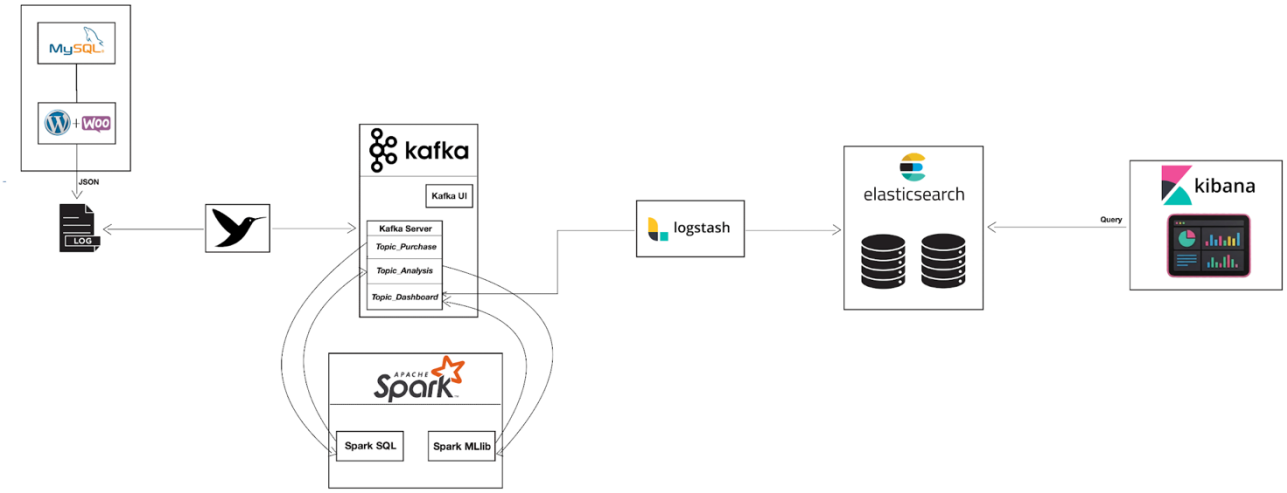
- **Transazione (Sorgente):** Dove il dato nasce (WordPress/WooCommerce).
- **Analisi (Elaborazione):** Dove il dato viene trasformato e arricchito (Spark).
- **Presentazione (Visualizzazione):** Dove il dato viene mostrato agli stakeholder (Kibana).

Tecnologie usate

Per garantire portabilità, isolamento e facilità di deploy, l'intera infrastruttura è basata su microservizi gestiti tramite Docker e Docker Compose. Questo permette di simulare un intero data center su una singola macchina.
A causa dei limiti del calcolatore utilizzato per realizzare il progetto, la **demo** verrà visualizzata e presentata accendendo e spegnendo i **servizi** in modo orchestrato.

Pipeline flusso dati

Il flusso segue una direzione lineare: dalla generazione del log alla dashboard finale.



Data ingestion

Generazione dei dati

Il lato **front-end** del sistema è un **sito WordPress** con **plugin WooCommerce**. Questa scelta simula fedelmente un ambiente di produzione reale. Ogni volta che un utente completa un ordine, il sistema genera un evento di log che contiene i dettagli della transazione (chi ha comprato, cosa, a che prezzo).

Fluent-Bit: Configurazione e ruolo di Log Collector

Per estrarre i dati da WordPress senza appesantire il database, utilizziamo Fluent-Bit.

- **Ruolo**: Agisce come un Log Collector che legge i file di log generati dal web server.
- **Funzionamento**: Appena una nuova riga viene scritta nel log, Fluent-Bit la cattura, la formatta e la invia al Message Broker (Kafka).

```
[SERVICE]
    Flush          1
    Log_Level      info
    Parsers_File   /fluent-bit/etc/parsers.conf

[INPUT]
    Name            tail
    Path            /input_logs/fluent-bit-orders.log
    Parser          json_parser
    Tag             e-commerce.purchase

[OUTPUT]
    Name            stdout
    Match           e-commerce.purchase

[OUTPUT]
    Name                kafka
    Match               e-commerce.purchase
    Brokers             e-commerce_kafkaserver:9092
    Topics              Topic_Purchase
    Format              json
    Message_Key_Field   order_id
```

```
[PARSER]
    Name        json_parser
    Format      json
    Time_Key    timestamp
    Time_Format %Y-%m-%d %H:%M:%S
```

Dettagli Tecnico del JSON

I dati viaggiano in **formato JSON** (**JavaScript Object Notation**).
È il **formato standard** per lo scambio di dati perché è **leggero** e **leggibile** sia dalle macchine che dagli umani.
Ecco un esempio di payload che include campi come: customer_email, items, total, session_duration.

```php
$order_data = [
    'order_id'         => $order->get_id(),
    'timestamp'        => current_time('mysql'), // Formato YYYY-MM-DD HH:MM:SS
    'items'            => [],
    'customer'         => [
        'first_name' => $order->get_billing_first_name(),
        'email'      => $order->get_billing_email(),
    ],
    'currency'         => $order->get_currency(),
    'total'            => (float)$order->get_total(),
    'duration_session' => $duration_session,
];
```

WordPress:

Ecco inoltre il codice di quello che succede subito dopo che un utente effettua un acquisto sull'E-Commerce, ciò corrisponde all'inizio del viaggio del dato.

```php
<?php
/**
 * Blocksy functions and definitions
 *
 * @link https://developer.wordpress.org/themes/basics/theme-functions/
 *
 * @package Blocksy
 */

if (version_compare(PHP_VERSION, '5.7.0', '<')) {
    require get_template_directory() . '/inc/php-fallback.php';
    return;
}

require get_template_directory() . '/inc/init.php';

// Codice Inserito:

add_action('init', 'start_session_timer');

function start_session_timer() {
    if (!isset($_COOKIE['visit_start_time'])) {
        setcookie('visit_start_time', time(), time() + 86400, "/");
    }
}

add_action('wp_login', 'reset_session_on_login');

function reset_session_on_login() {
    setcookie('visit_start_time', time(), time() + 86400, "/");
}

```

```php
add_action( 'woocommerce_thankyou', 'generate_tap_order_json', 10, 1 );

function generate_tap_order_json( $order_id ) {
    if ( ! $order_id ) return;

    $order = wc_get_order( $order_id );

    // Duration Session:
    $now = time();
    $start_time = isset($_COOKIE['visit_start_time']) ? intval($_COOKIE['visit_start_time']) : $now;
    $duration_session = $now - $start_time;


    $order_data = [
        'order_id'         => $order->get_id(),
        'timestamp'        => current_time('mysql'), // Formato YYYY-MM-DD HH:MM:SS
        'items'            => [],
        'customer'         => [
            'first_name' => $order->get_billing_first_name(),
            'email'      => $order->get_billing_email(),
        ],
        'currency'         => $order->get_currency(),
        'total'            => (float)$order->get_total(),
        'duration_session' => $duration_session,
    ];


    foreach ( $order->get_items() as $item_id => $item ) {
        $order_data['items'][] = [
            'product_name' => $item->get_name(),
            'quantity'     => $item->get_quantity(),
            'total'        => (float)$item->get_total(),
        ];
    }

    $json_payload = json_encode( $order_data );

    $file_path = '/var/www/html/wp-content/uploads/fluent-bit-orders.log';
    file_put_contents( $file_path, $json_payload . PHP_EOL, FILE_APPEND );
}
```

Il Cuore del sistema: Apache Kafka

Kafka: Broker

Apache Kafka è il componente centrale che garantisce la resilienza del sistema.
Funge da broker tra chi produce i dati (Fluent-Bit) e chi li consuma (Spark).
Se Spark dovesse spegnersi per manutenzione o crash, i dati non andrebbero persi, ma rimarrebbero salvati in Kafka in attesa di essere letti. Questo disaccoppia i sistemi ed evita colli di bottiglia.

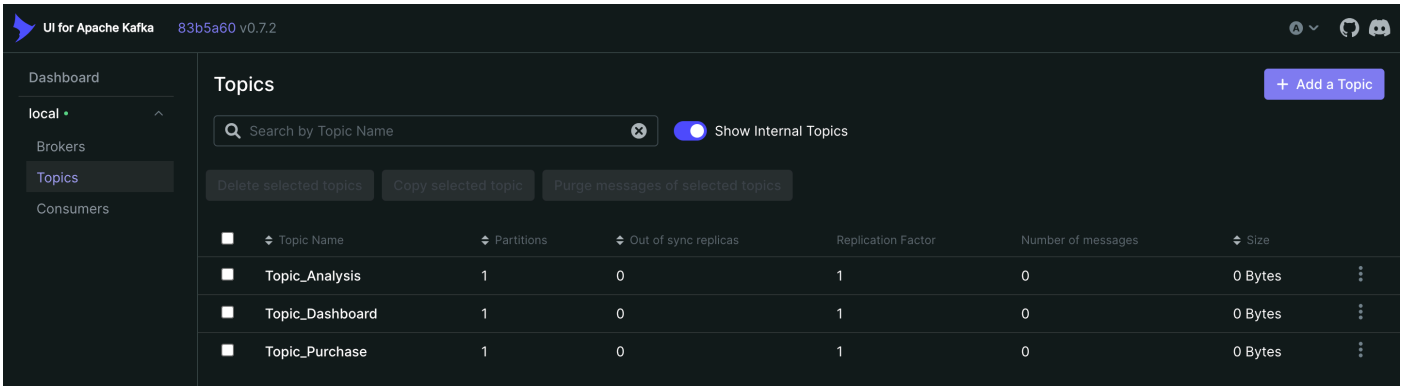
Struttura dei Topic Creati

I dati vengono organizzati in "canali" tematici chiamati Topic:

- **Topic_Purchase**: Contiene i dati grezzi appena arrivati da Fluent-Bit.
- **Topic_Analysis**: Contiene i dati elaborati da Spark SQL.
- **Topic_Dashboard**: Contiene i dati finali, arricchiti con il *modello K-Means*, pronti per Kibana.

Monitoraggio tramite Kafka UI

Per gestire visivamente i flussi e verificare che i messaggi arrivino correttamente, utilizziamo Kafka UI, un'interfaccia grafica che permette di ispezionare il contenuto dei topic e lo stato dei consumer.



Elaborazione Real-Time: Apache Spark

Spark SQL (process_orders.py)

Utilizziamo Spark Streaming per elaborare i dati in movimento.
Lo script process_orders.py esegue tre operazioni fondamentali:

- **Estrazione dati**: Legge lo stream continuo da Topic_Purchase.
- **Trasformazione**: Seleziona solo le colonne utili, calcola i totali e applica regole di business (es. definire lo spender_type come "High" o "Standard" in base all'importo).
- **Caricamento su Topic**: Scrive il risultato pulito su Topic_Analysis.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, to_json, struct, when
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, FloatType, ArrayType

def main():
    # 1. conf. iniziale
    spark = SparkSession.builder \
        .appName("E-Commerce_BehaviorAnalysis") \
        .getOrCreate()

    spark.sparkContext.setLogLevel("WARN")

    # 2. schema json: recupero dati
    json_scheme = StructType([
        StructField("order_id", IntegerType()),
        StructField("timestamp", StringType()),
        StructField("items", ArrayType(StructType([
            StructField("product_name", StringType()),
            StructField("quantity", IntegerType()),
            StructField("total", FloatType())
        ]))),
        StructField("customer", StructType([
            StructField("first_name", StringType()),
            StructField("email", StringType())
        ])),
        StructField("currency", StringType()),
        StructField("total", FloatType()),
        StructField("duration_session", IntegerType())
    ])

    # 3. Lettura da Topic_Purchase
    raw_stream = spark.readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "e-commerce_kafkaserver:9092") \
        .option("subscribe", "Topic_Purchase") \
        .option("startingOffsets", "earliest") \
        .option("failOnDataLoss", "false") \
        .load() # 'earliest' per prendere tutto lo storico riletto da FluentBit, altrimenti 'latest'

    # Processo col_Value -> String -> JSON
    json_stream = raw_stream.select(
        from_json(col("value").cast("string"), json_scheme).alias("data")
    ).select("data.*")

    # 4. Data Enrichment:

    processed_stream = json_stream.withColumn(
        "spender_type",
        when(col("total") > 150, "High Spender").otherwise("Standard Spender")
        ).withColumn(
         "stock_alert",
         when(col("total") > 500, "High Demand").otherwise("Normal Demand")
        )

    # 5. Scrittura su Topic_Analysis
    query = processed_stream.select(
        to_json(struct(col("*"))).alias("value")) \
        .writeStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "e-commerce_kafkaserver:9092") \
        .option("topic", "Topic_Analysis") \
        .option("checkpointLocation", "/opt/spark-apps/checkpoint_sql") \
        .trigger(processingTime='20 seconds') \
        .start()

    print("--- Streaming Behaviour_Analysis started ---")
    query.awaitTermination()

if __name__ == "__main__":
    main()
```

Spark MLlib (Machine Learning)

La componente di **intelligenza artificiale** è gestita dalla **libreria MLlib**.

- **Modello**: Viene utilizzato il *K-Means*, un modello ML di clustering.
- **Logica**: Il modello analizza una feature: la durata della sessione.
  In base a questo dato, classifica l'utente in due cluster:
  1. ***Impulsive***: *Bassa durata*, pochi click e acquisto rapido.
  2. ***Reasoned***: *Alta durata* (tempo di riflessione), acquisto ponderato.

Il risultato della categoria grazie a K-Means viene aggiunto al dato e inviato a Topic_Dashboard.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, to_json, struct, when
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, FloatType, ArrayType

from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans
from pyspark.ml import Pipeline

def train_kmeans_model(spark):
    csv_path = "/opt/spark-apps/training_data.csv"
    try:
        df_train = spark.read.option("header", "true") \
            .option("inferSchema", "true") \
            .csv(csv_path)

        df_train = df_train.withColumn("duration_session", col("duration_session").cast("integer"))

        assembler = VectorAssembler(inputCols=["duration_session"], outputCol="features")
        kmeans = KMeans(k=2, seed=1)
        pipeline = Pipeline(stages=[assembler, kmeans])

        # Addestramento
        model = pipeline.fit(df_train)

        # Cluster
        centers = model.stages[-1].clusterCenters()
        impulsive_cluster = 0 if centers[0][0] < centers[1][0] else 1
        return model, impulsive_cluster

    except Exception as E:
        print(f"Error Loading Training Data: {E} !!!")
        return None, 0

def main():
    spark = SparkSession \
        .builder \
        .appName("E-Commerce_Machine_Learning") \
        .getOrCreate()

    spark.sparkContext.setLogLevel("WARN")

    ml_model, impulsive_idx = train_kmeans_model(spark)

    if ml_model is None:
        print("Stopping due to training error")
        return

    # 1. Schema Input dal Topic_Analysis
    json_scheme = StructType([
        StructField("order_id", IntegerType()),
        StructField("timestamp", StringType()),
        StructField("items", ArrayType(StructType([
            StructField("product_name", StringType()),
            StructField("quantity", IntegerType()),
            StructField("total", FloatType())
        ]))),
        StructField("customer", StructType([
            StructField("first_name", StringType()),
            StructField("email", StringType())
        ])),
        StructField("currency", StringType()),
        StructField("total", FloatType()),
        StructField("duration_session", IntegerType()),
        StructField("spender_type", StringType()), # Extra field
        StructField("stock_alert", StringType())   # Extra field
    ])

    # 2. Lettura da Topic_Analysis
    raw_stream = spark.readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "e-commerce_kafkaserver:9092") \
        .option("subscribe", "Topic_Analysis") \
        .option("startingOffsets", "earliest") \
        .option("failOnDataLoss", "false") \
        .load()

    json_stream = raw_stream.select(
        from_json(col("value").cast("string"), json_scheme).alias("data")
    ).select("data.*")

    # 3. ML Prediction
    prediction_stream = ml_model.transform(json_stream)

    final_stream = prediction_stream.withColumn(
        "purchase_behaviour",
        when(col("prediction") == impulsive_idx, "Impulsive")
        .otherwise("Reasoned")
    ).drop("features", "prediction")

    # 4. Scrittura su Topic_Dashboard
    query = final_stream.select(to_json(struct(col("*"))).alias("value")) \
        .writeStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "e-commerce_kafkaserver:9092") \
        .option("topic", "Topic_Dashboard") \
        .option("checkpointLocation", "/opt/spark-apps/checkpoint_ml") \
        .trigger(processingTime='20 seconds') \
        .start()

    print("--- Machine Learning Started ---")
    query.awaitTermination()

if __name__ == "__main__":
    main()
```

Indicizzazione e Visualizzazione:

Logstash:

Logstash preleva i dati finali da Kafka per inviarli al database.

- **Gestione Offset**: Un punto critico affrontato è stato la configurazione dell'offset (earliest vs latest).
  Ho configurato Logstash in modo da poter leggere dall'inizio (earliest) e utilizzare un group_id specifico, garantendo che nessun dato storico venisse perso durante i riavvii del sistema.

```
input {
    kafka {
        bootstrap_servers => "e-commerce_kafkaserver:9092"
        topics => ["Topic_Dashboard"]
        codec => "json"
        auto_offset_reset => "earliest"
        group_id => "logstash_v1"
    }
}

filter {
    mutate {
        add_field => { "source_system" => "spark_mllib" }
    }
}

output {
    elasticsearch {
        hosts => ["elasticsearch:9200"]
        index => "ecommerce-predictions"
        ssl => false
        ssl_certificate_verification => false
    }
    stdout { codec => rubydebug }
}
```

Logstash preleva i dati completi e arricchiti dal Machine Learning e li inserisce ordinatamente nel database di Elasticsearch per poterli visualizzare.

Elasticsearch:

Elasticsearch è il motore di ricerca e analisi. A differenza dei database tradizionali (SQL), è un database NoSQL orientato ai documenti. Questo lo rende velocissimo nell'indicizzare i JSON provenienti da Spark e nel permettere interrogazioni complesse in tempo reale.

Kibana:

Kibana è l'interfaccia visuale che interroga Elasticsearch.
Abbiamo realizzato una Dashboard interattiva divisa in più “widget”:

- **CEO View:** Mostra cosa è successo (Fatturato totale, Prodotti più venduti, Profile Analysis etc.)

KPI Realizzati e Analizzati

I Key Performance Indicators implementati includono:

- **Total Revenue**: Somma in tempo reale degli incassi.
- **Top Selling Products**: Classifica dei prodotti più popolari.
- **Behavioral Segmentation**: Percentuale di acquisti impulsivi vs ragionati.

Riepilogo Finale e Spiegazione Use-Case

Il progetto dimostra come **un'architettura con tecnologie moderne** possa trasformare un *semplice log di acquisto* ("T-Shirt venduta a 20€") *in una visione strategica* ("L'utente Lorenzo è un cliente impulsivo che spende sopra la media"). **Questo permette al management di un qualsiasi E-Commerce di reagire istantaneamente alle tendenze di mercato** e *personalizzare l'offerta per massimizzare i guadagni e raggiungere i budget mensili prefissati*.

Comandi utilizzati spesso:

• Accensione Demo orchestrato:
0. rm -rf checkpoint_sql && rm -rf checkpoint_mllib
1. docker-compose up db WordPress fluent-bit kafka kafka-ui topics sparksql sparkmllib
2. docker-compose down db WordPress fluent-bit sparksql sparkmllib
3. docker-compose up logstash elasticsearch kibana

• Accesso al container WordPress:

```
docker exec -it e-commerce_wordpress bash
```

- **Installazione dell'editor di testo (Nano):**

```
apt-get update && apt-get install nano -y
```

- **Installazione di WP-CLI (Command Line Interface per WordPress):**
  Serve per creare utenti e gestire il sito da terminale.

```
curl -O https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar
chmod +x wp-cli.phar
mv wp-cli.phar /usr/local/bin/wp
```

• Modifica del file `functions.php`:

```
cd /var/www/html/wp-content/themes/blocksy
nano functions.php
```

- **Creazione Utenti (Data Generation):** *Comando per creare un singolo utente (es. Mario):*

```
wp user create mario mario@test.com --role=customer --user_pass="password123" --
allow-root
```

- **Verifica Ricezione Dati (Consumer da Terminale oppure direttamente kafka-ui):**

Comando per "ascoltare" il topic `Topic_Purchase` direttamente dal container Kafka:

```
docker exec -it e-commerce_kafkaserver /opt/kafka/bin/kafka-console-consumer.sh -
-bootstrap-server localhost:9092 --topic Topic_Purchase --from-beginning
```