Large Scale Data Management

2nd Assignment

Tsirmpas Dimitris f3352315

Python data producer

The data producer emulator (`src/data_producer.py`) is a python script which registers to our kafka cluster and periodically sends random data to it. Every `INTERVAL_SECS` seconds, the script will pick one of 50 randomly generated names, one of the songs from the songs file (`data/spotify_songs.csv`) generate a random id and the current timestamp. The script will then send a json encoded and compressed message to the kafka cluster containing the above information.

```
import json
import random
import asyncio
import csv
import datetime
import time

from aiokafka import AIOKafkaProducer
from faker import Faker


INTERVAL_SECS = 1
MESSAGE_LIMIT = 1000
SONG_FILE_PATH = "data/spotify-songs.csv"
TOPIC = "test"


def serializer(value):
    return json.dumps(value).encode()


def generate_record(id: int, name: str, song: str) -> dict:
    return {
        "id": id,
        "name": name,
        "song": song,
        "time": datetime.datetime.now().strftime("%Y-%m-%d %I:%M"),
    }


def generate_names(num_names: int) -> list:
    fake = Faker()
```

```
    fake_names = [fake.name() for _ in range(num_names - 1)]
    fake_names.append("Dimitris Tsirmpas")
    return fake_names


def generate_songs(song_file_path: str) -> list:
    with open(song_file_path, "r") as file:
        reader = csv.reader(file, delimiter=",")
        songs = [row[0] for row in reader]
        random.shuffle(songs)
        return songs


async def produce():
    producer = AIOKafkaProducer(
        bootstrap_servers="localhost:29092",
        value_serializer=serializer,
        compression_type="gzip",
    )

    names = generate_names(50)
    songs = generate_songs(SONG_FILE_PATH)

    await producer.start()
    print("Data stream open. Records sent:")

    num_messages_sent = 0
    while num_messages_sent < MESSAGE_LIMIT:
        data = generate_record(
            random.randint(0, 99999),
            random.choice(names),
            random.choice(songs),
        )
        print(data)

        await producer.send(TOPIC, data)
        await producer.flush()

        num_messages_sent += 1
        time.sleep(INTERVAL_SECS)


    await producer.stop()


loop = asyncio.get_event_loop()
```

```
result = loop.run_until_complete(produce())
```

Python Consumer

The consumer script (`src\spark-stream.py`) is responsible for getting the live data from the kafka cluster, combining their information with the aforementioned data file.

The script initially reads the songs form the csv file and creates a RDD, which is immediately cached, since its data are immutable for the entire lifetime of the program.

The scipt also subscribes to the same kafka cluster and listens for any new data. The new data are read, decompressed and a new dataframe is created from the json values.

We disambiguate the song and user name columns. The two dataframes are then joined. We create two extra columns, "hour" and "date" (int and string respectively), through two user-defined-functions (`derive_hour()` and `derive_date()`) which will be used for the cassandra partition key.

The joined dataframe is subsequently sent to the cassandra cluster every `FLUSH_INTERVAL_SECS` seconds.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import (
    StructType,
    StructField,
    IntegerType,
    StringType,
)
from pyspark.sql.functions import from_json, col, udf


DEBUG = False
FLUSH_INTERVAL_SECS = 30

# spark initialization
spark = (
    SparkSession.builder.appName("SSKafka")
    .config(
        "spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0"
    )
    .getOrCreate()
)

spark.sparkContext.setLogLevel("ERROR")
```

```
# spotify songs
song_df = (
    spark
    .read
    .option("header", True)
    .option("inferSchema", True)
    .csv("file:////vagrant/data/spotify-songs.csv")
    .withColumnRenamed("name", "song_name") # disambiguate user name and song name
    .withColumnRenamed("key", "musical_key") # avoid cql reserved word "key"
    .cache() # cache dataset as the file is immutable
)

print("Song dataframe schema:")
song_df.printSchema()

# request from kafka consumer

@udf(returnType=IntegerType())
def derive_hour(datetime_string: str) -> int:
    _, time = datetime_string.split(" ")
    return int(time[:2])


@udf(returnType=StringType())
def derive_date(datetime_string: str) -> str:
    return datetime_string.split(" ")[0]


request_schema = StructType(
    [
        StructField("id", IntegerType(), False),
        StructField("name", StringType(), False),
        StructField("song", StringType(), False),
        StructField("time", StringType(), False),
    ]
)

request_df = (
    spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "localhost:29092")
    .option("subscribe", "test")
    .option("startingOffsets", "latest")
    .load()
    .selectExpr("CAST(value AS STRING)")
    .select(from_json(col("value"), request_schema).alias("data"))
```

```
    .select("data.*")
    # add columns needed for the partition key
    .withColumn("hour", derive_hour(col("time")))
    .withColumn("date", derive_date(col("time")))
)

print("Request dataframe schema:")
request_df.printSchema()

# Joining the DataFrames
joined_df = request_df.join(
    song_df, request_df.song == song_df.song_name, "inner"
).drop("song_name")

# Printing the schema of the joined DataFrame
print("Final dataframe schema:")
joined_df.printSchema()


# Specify the output mode and format
def writeToCassandra(writeDF, _):
    print(f"Flushing {writeDF.count()} records to database...")
    (writeDF
    .write
    .format("org.apache.spark.sql.cassandra")
    .mode("append")
    .options(table="listening_history", keyspace="spotify")
    .save())


if DEBUG:
    debug_query = (
        joined_df
        .writeStream
        .outputMode("update")
        .format("console")
        .option("truncate", False)
        .start()
        .awaitTermination()
    )
else:
    # retry until connection is established
    result = None
    while result is None:
        try:
            # connect
```

```
            cassandra_query = (
                joined_df.writeStream.option(
                    "spark.cassandra.connection.host", "localhost:9042"
                )
                .foreachBatch(writeToCassandra)
                .outputMode("update")
                .trigger(processingTime=f"{FLUSH_INTERVAL_SECS} seconds")
                .start()
                .awaitTermination()
            )
        except:
            pass
```

Cassandra

Schema

We use the schema below for our Cassandra table (available at `src\create_db.cql`).

The table contains all the features used by the joined dataframe. We rename the `key` column to `musical_key` since "key" is a reserved word in CQL. The partition key is comprised of the user name, day and hour of the record, enabling queries for specific users at specific time stamps. We optionally include the whole time-stamp as the clustering key, where the timestamp is of the format `YY-MM-DD HH-MM` in order for the sorting to make intuitive sense.

```
CREATE KEYSPACE IF NOT EXISTS spotify
WITH REPLICATION = {
    'class': 'SimpleStrategy',
    'replication_factor': 1
};

CREATE TABLE IF NOT EXISTS spotify.listening_history (
    id int,
    name text,
    song text,
    time text,
    hour int, -- seperate hour column for partition key
    date text, -- seperate date column for partition key
    artists text,
    duration_ms int,
    album_name text,
    album_release_date text,
    danceability float,
    energy float,
    musical_key int, -- originally "key" in the csv, which is a reserved word in cql
    loudness float,
```

```
    mode float,
    speechiness float,
    acousticness float,
    instrumentalness float,
    liveness float,
    valence float,
    tempo float,
    PRIMARY KEY ((name, date, hour), time)
) WITH CLUSTERING ORDER BY (time DESC);
```

Queries

Having populated our database with records from the producer we can run a few queries. All queries are executed efficiently since they use the whole partition key. Since the output can be very large, we export them to text files in the `output` subdirectory. Respectively, the queries can be found in the `src` subdirectory.

Table Sample

```
SELECT * FROM spotify.listening_history LIMIT 50 ;
```

Output: `output/output_full.txt`

Song names

```
SELECT song
FROM spotify.listening_history
WHERE name = 'Dimitris Tsirmpas'
AND date = '2024-03-03'
AND hour=10;
```

Output: `output/output_simple.txt`

Average statistics

```
SELECT name, date, hour, avg(danceability), avg(tempo), avg(acousticness), avg(tempo)
FROM spotify.listening_history
WHERE name = 'Dimitris Tsirmpas'
AND date = '2024-03-03'
AND hour = 10
GROUP BY hour;
```

Output: `output/output_agg.txt`