COMP4651 (USTreamers)

CHO, Won Seo (20544167)
HAN, Yoonseo (20554435)
SUNG, Jin (20545240)

Introduction

The project aims to provide real-time information into the winning probability of NBA games by leveraging Apache Kafka, Apache Flink, and PostgreSQL databases within a containerized environment. The system architecture involves several components, including a real time data generator, backend server for information processing and probability prediction, and a frontend application for users.

Data Pipeline

This section covers the process of handling the stream of data end-to-end from the data producer to broker, from broker to consumer and from consumer to data sink.

Data Producer

The python file `main.py`, containerized by Docker via `python_app`, produces and transfers the game updates in the NBA games, and sends it towards Kafka. `updates` are generated by the function `generate_nba_news()` and sent to Kafka through module called `confluent_kafka`.

```
producer.produce(topic,        # sends to broker:29092, topic: 'game_updates'
                 key=updates['player'],
                 value=json.dumps(updates),   # serialized in json and sent to kafka
                 on_delivery=delivery_report)  # func to check if msg is delivered
```

Data Broker

Apache Kafka is set up within Apache Zookeeper, and Zookeeper will be responsible in managing the brokers inside the Kafka clusters. Streams of data will be accessible at Kafka via port 29092 as containers in same docker network will be advertised by `KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:29092`.

The stream of data could be monitored from CLI by using the command:

```
kafka-console-consumer --topic game_updates --bootstrap-server broker:29092
```

to check for the events that are under the topic called `game_updates`.

Data Consumer

There is a layer of Apache Flink listening to the streams of events from Kafka. In `DataStreamJob.java`, it creates a data stream `updateStream` from broker, ready to be further processed using Flink operations.

```
KafkaSource<Update> source = KafkaSource.<Update>builder()
    .setBootstrapServers("broker:29092")  // retriving from broker container
    .setTopics("game_updates")  // specify topic
    .setGroupId("FlinkNBA")
    .setStartingOffsets(OffsetsInitializer.earliest())
    .setValueOnlyDeserializer(new JSONValueDeserializationSchema())   // deserialize
    .build();
```

```
DataStream<Update> updateStream = env.fromSource(source,
                                  WatermarkStrategy.noWatermarks(), "Kafka source");
```

Then, `addSink()` function will be utilised to sink the `Update` objects into the sql table in `postgres` container. To derive the winning probability of the team in a specific game, additional statistical data should be further processed; therefore `updateStream` is further mapped into data format called `Gamestat`, mapped by each distinct `gameId`.

```
updateStream.map(
        update -> { return new Gamestat object;} // callback to generate Gamestat obj
                 ).keyBy(Gamestat::getGameId)
                  .reduce((gamestat, t1) -> {
                      reduce_actions })
                  .addSink(JdbcSink.sink( "SQL Statement to store reduced objects",
                      execOptions, connOptions))
                  .name("Insert into Gamestat table");
```

The mapped objects would then be further reduced to categorise and aggregate compute the variables that contribute to winning probability, through `reduce_actions`. After MapReduce operation, distinct game summary will be updated in the DB for each NBA games.

Once Flink is executed, the Flink's JobManager receives the job submission and coordinates the execution, distributing the job across the available TaskManagers in the cluster to execute the assigned tasks. The Flink Web UI could be accessed by port 8081 by default, through `localhost:8081`.

Database
The PostgreSQL database is containerized by Docker via `postgres` and stores the data from Apache Flink. The database will be accessible via port 5432, and it will store the incoming data from flink into two tables: `updates` and `gamestats`. The `gamestats` table is responsible in storing the variables dependent to winning probability per each `gameId`.

The following commands could be executed in the bash process inside the `postgres` container to monitor the data storage.

```
  $ psql -U posgre
  $ select * from gamestats;
```

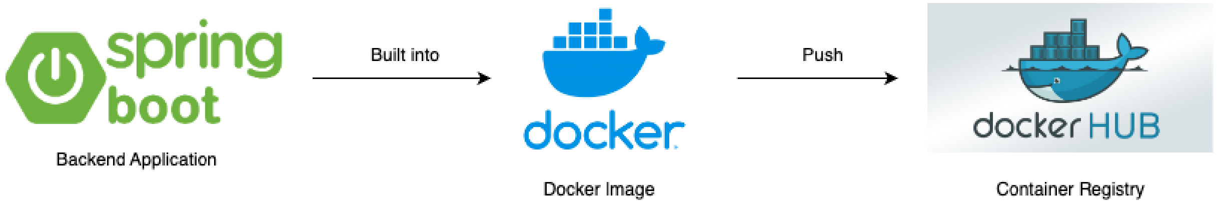
Backend: Development of Web APIs with Spring Boot

This section covers the process of developing web APIs using a Java Framework known as Spring Boot, which is connected to PostgreSQL from the perspective of a backend developer. We aim to explain the overview of the technology stack employed, our approach to API design and database schema, and the overall workflow of development.

For our project, we developed API endpoint that allowed the frontend to access records of winning probability stored in a database.

Technology Stack and Deployment Flow

Our backend application, powered by Spring Boot, is also containerized using Docker. This enables the application to run in an isolated environment. By building our application into a Docker image and subsequently running the container, the application becomes accessible on the host machine via *localhost:8080*.



As illustrated in the diagram above, we used Spring Boot to implement the algorithm for calculating winning rates and to manage web requests. Additionally, Docker Hub is utilized to host the built images.

Our algorithm adjusts win probabilities based on several factors, including team ranking, scores, and fouls. These values are dynamically updated in real-time to reflect changes in the game, with higher rankings or scores boosting win probabilities and more fouls reducing them. Through normalization, we ensure the total probabilities always sum to 100%, accurately reflecting each team's chances of winning.

API and Data Structure Design

Although the detailed explanation of API and database schema design is beyond the scope of this course, it is essential to outline our approach.

A single endpoint is established for executing read operations on `win_probability` records. This endpoint handles requests and returns responses including a `success` boolean flag, along with the updated record content in JSON format. It also uses appropriate HTTP status codes to indicate the result of the operation.

The API response format is structured as follows:

```
{
  "homeWinProbability": 50.0,
  "awayWinProbability": 50.0
}
```

Regarding data structure, we used PostgreSQL, a relational database, to store our project's data. The schema for the tables managed within our project is as follows:

Attribute	Type	Note
gameid	String	Primary Key
home_score	Integer	
home_foul	Integer	
away_score	Integer	
away_foul	Integer	
situation	Integer	

Attribute	Type	Note
id	Integer	Primary Key
name	String	
rank	Integer	

Table 1: The schema of **gamestats and team** table.

Attribute	Type	Note
gameid	String	Primary Key
player	String	
team	String	
situation	Integer	
updateTime	Timestamp	

Attribute	Type	Note
id	Integer	Primary Key
home_win_prob	Double	
away_win_prob	Double	

Table 2: The schema of **update and win_probability** table.

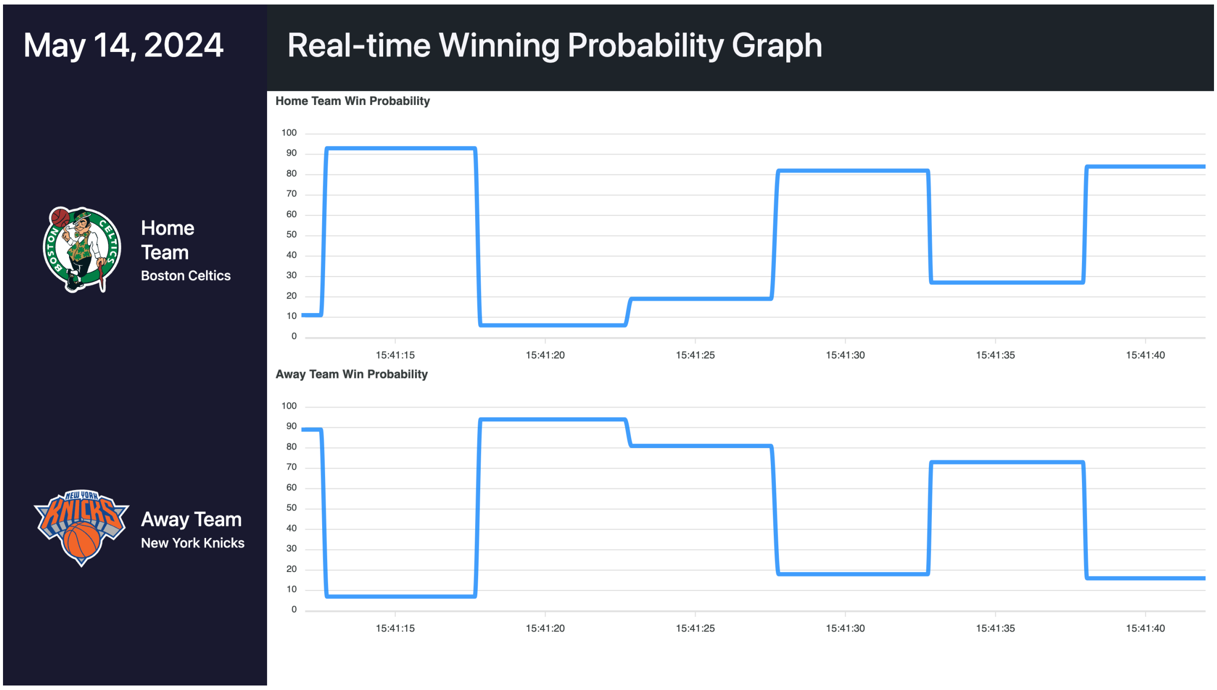
Algorithm

Our algorithm adjusts win probabilities based on several factors, including team ranking, scores, and fouls. These values are dynamically updated in real-time to reflect changes in the game, with higher rankings or scores boosting win probabilities and more fouls reducing them. Through normalization, we could keep the sum of probabilities at 100%, accurately reflecting each team's chances of winning.

Frontend: Development of Web page with ReactJS

The following section covers the process of front end development to represent each team's winning probability in a real time line chart. The following chart represents the change in winning probability for both Home team and Away team throughout the game based on real time game stats and is updated every 10 seconds based on the probability received from the backend API.

Page set up and structure



May 14, 2024

Real-time Winning Probability Graph

The frontend is composed of 4 main parts: Header, Sidebar, Home team probability real time graph and Away team probability real time graph. The Sidebar reflects the logos of the Home Team and Away Team.

A single parent component is responsible in receiving data from the backend and sending the probabilities to the respective teams. Request is made every 10 seconds and is reflected in the form of real time line graphs.

Data retrieval from backend container

The communication between the backend and frontend containers is facilitated through RESTful APIs. The backend exposes "/api/winningProb" as endpoints that provide data for the realtime probability. The frontend consumes these endpoints to fetch data and display it to users.

Communication between containers connected to the same network was facilited using the designated hostnames. The following project used the name "backend" for the backend container hence the frontend container can access it using the hostname backend or the service name *backend/api/winningProb* .

Technology Stack and Deployment flow

The following frontend application is then containerized using docker container. Using the Node.js LTS version as the base image, working directories were set up and the respective files in the local machine were moved to the front end application container and built. The production server is configured using Nginx which is a light weight Web Server dealing with static files. Port 80 is exposed to allow external access to the application served by Nginx where a 3000:80 mapping was used to gain access via port 3000.

Future development plans

Our team is hoping to be able to further develop the project to be able to give a more accurate prediction of the game by additionally encountering variables. Furthermore, our team is hoping to improve the user interface of the real time graph to give intuitive information towards the users in having a better understanding of the key events happening in real time.