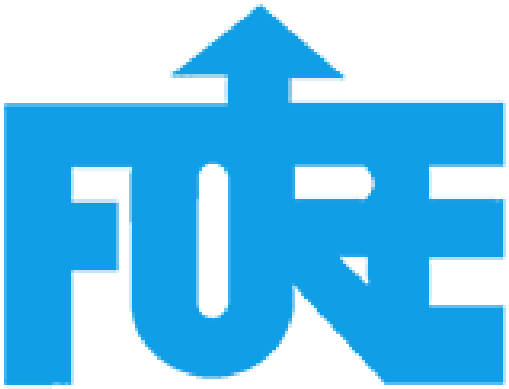


FORE SCHOOL OF MANAGEMENT

STREAMING DATA ANALYTICS
OF
LIVE FOOTBALL MATCH UPDATES

Submitted To:

Prof. Aditya Dua

Submitted By:

Pragyanpad Devchoudhury – 321039
Subhajit Paul - 321173

Data

Columns

1. Event ID
2. Match ID
3. Event Type
4. Event ID 1
5. Time
6. Description
7. Timestamp

Content Summary:

The file appears to be a record of events during a soccer match. Each row represents a single event, with the following columns:

- **Event ID:** A unique identifier for each event.
- **Match ID:** The ID of the match.
- **Event Type:** The type of event (e.g., foul, corner, card shown, etc.)
- **Event ID.1:** A duplicate of the Event ID.
- **Time:** The time at which the event occurred.
- **Description:** A textual description of the event.
- **Timestamp:** The date and time of the event.

Data Analysis:

1. **Event Frequency:**
    - The most frequent event type is "foul," occurring multiple times throughout the match.
    - Other common events include "corner" and "card shown."
2. **Team-Specific Events:**
    - The file records events for both Team A and Team B, indicating fouls, cards, and other actions by players from each team.
3. **Time Progression:**
    - The "Time" column shows the chronological order of events within the match.
4. **Player Actions:**
    - The "Description" column details specific player actions, such as fouls committed, cards received, shots taken, and substitutions made.

Potential Uses:

- **Match Analysis:** The data can be used to analyze the match's flow, identify key moments, and assess player performance.
- **Statistical Reporting:** The information can be used to generate statistics on fouls, cards, substitutions, and other relevant metrics.
- **Highlight Creation:** The file can be used to identify key moments for creating video highlights or summaries of the match.

Further Exploration:

- **Data Visualization:** Creating charts or graphs to visualize the frequency of different event types, the distribution of fouls between teams, or the time progression of events could provide valuable insights.
- **Advanced Analysis:** More complex analysis could involve calculating player statistics, identifying patterns in play, or predicting future outcomes based on historical data.

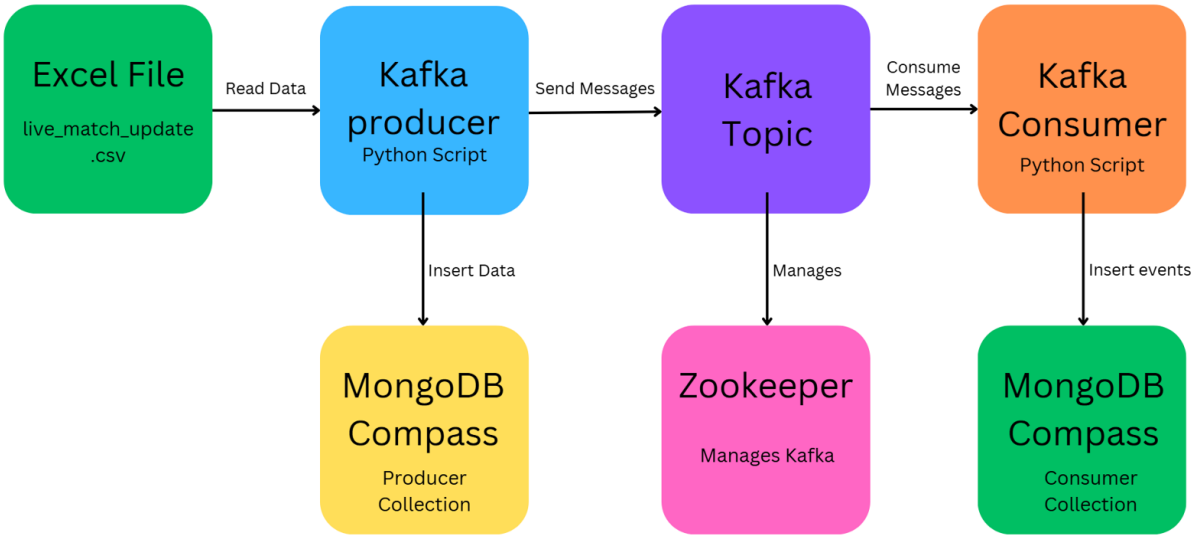
Event types with Event Id is hereby mentioned:

Events	Event ID
foul	1
corner	2
card shown	3
shot	4
free kick	5
possession change	6
substitute	7
save by goalkeeper	8
pass	9
throw in	10
goal	11

Time is started at 00:00 hours and continues for ends after 90 minutes.
Foe Every 10 second live updates are generated
Match ID is 7339
Events are described as below:

A	B	C	D	E	F	G
Event ID	Match ID	Event Type	Event ID.1	Time	Description	Timestamp
1	7339	foul	1	00:00	Foul committed by Player 13 of Team A.	01-01-2024 00:00
2	7339	corner	2	00:10	Corner kick awarded to Team B.	01-01-2024 00:00
3	7339	card shown	3	00:20	Yellow card shown to Player 13 of Team B.	01-01-2024 00:00
4	7339	foul	1	00:30	Foul committed by Player 7 of Team B.	01-01-2024 00:00
5	7339	shot	4	00:40	Player 11 from Team B takes a shot at the goal.	01-01-2024 00:00
6	7339	free kick	5	00:50	Free kick for Team B.	01-01-2024 00:00
7	7339	possession change	6	01:00	Player 2 from Team A snatches the ball from Player 14 of Team B.	01-01-2024 00:01
8	7339	card shown	3	01:10	Yellow card shown to Player 1 of Team A.	01-01-2024 00:01
9	7339	foul	1	01:20	Foul committed by Player 2 of Team B.	01-01-2024 00:01
10	7339	corner	2	01:30	Corner kick awarded to Team B.	01-01-2024 00:01
11	7339	substitute	7	01:40	Player 8 from Team A substituted for Player 1.	01-01-2024 00:01
12	7339	card shown	3	01:50	Yellow card shown to Player 4 of Team B.	01-01-2024 00:01
13	7339	substitute	7	02:00	Player 6 from Team A substituted for Player 15.	01-01-2024 00:02
14	7339	free kick	5	02:10	Free kick for Team A.	01-01-2024 00:02
15	7339	save by goalkeeper	8	02:20	Goalkeeper of Team B saves a shot from Player 6 of Team A.	01-01-2024 00:02

Football Match Data Streaming Architecture Report



Overview
The architecture implements a real-time football match data streaming system utilizing Apache Kafka for message handling, MongoDB Compass for data persistence, and Python scripts for data processing. The system processes live match updates, including events like goals, cards, and fouls.

System Components
Data Source
- Excel file (live_match_update.csv) containing match events
- Data structure includes Event ID, Match ID, Event Type, Time, and Description
- Serves as the primary input source simulating real-time match updates

Apache Kafka Infrastructure
1. Zookeeper
- Manages Kafka cluster configuration
- Handles broker coordination
- Maintains cluster state and configuration

2. Kafka Producer
- Python script reading Excel data
- Serializes messages to JSON format
- Publishes messages to 'stocktopica' topic
- Implements parallel MongoDB storage for data backup

3. Kafka Topic (stocktopica)
- Maintains message queue
- Ensures reliable message delivery
- Supports multiple consumer groups

4. Kafka Consumer
- Processes messages from 'stocktopica' topic

- Deserializes JSON messages
- Implements event-specific logic for different match events
- Stores processed data in MongoDB

Data Storage (MongoDB Compass)

1. Producer Collection (football.ticker_football_producer)
- Stores raw event data
- Maintains original message structure
- Serves as backup for producer-side data

2. Consumer Collection (football_data.events)
- Stores processed event data
- Optimized for event querying
- Supports match analysis and reporting

Data Flow

1. Data Ingestion
- Excel file is read by producer script
- Data is converted to JSON format
- Each row represents a distinct match event

2. Message Processing
- Producer publishes messages to Kafka topic
- Messages are queued and maintained by Kafka
- Consumer processes messages based on event type
- Special handling for cards, goals, and other events

3. Data Persistence
- Dual storage strategy implemented
- Producer stores raw data for backup
- Consumer stores processed data for analysis
- MongoDB ensures data durability and query ability

Key Features

1. Real-time Processing
- Event-driven architecture
- Minimal processing latency
- Immediate event notifications

2. Data Reliability
- Kafka ensures message delivery
- Dual storage strategy prevents data loss
- Zookeeper maintains system stability

3. Scalability
- Kafka supports multiple producers/consumers
- Horizontal scaling capability
- Distributed system architecture

4. Event Monitoring
- Real-time alerts for specific events
- Automated event classification
- Custom handling for different event types

Technical Specifications

Producer Configuration
```python
bootstrap_servers: 'localhost:9092'
topic: 'stocktopica'
message_format: JSON
```

Consumer Configuration
```python
group_id: 'football-consumer-group'
auto_offset_reset: 'earliest'
enable_auto_commit: True
```

MongoDB Collections
```python
producer_collection: 'ticker_football_producer'
consumer_collection: 'events'
database: 'football'
```

Conclusion
The architecture provides a robust foundation for real-time football match data processing, combining the reliability of Kafka messaging with the flexibility of MongoDB storage. The system's modular design allows for future expansions and modifications while maintaining data integrity and processing efficiency.

Python Code Analysis – Producer and Consumer

Producer:

Code Functionality:
The Python code implements a producer application that reads live match updates from a CSV file (live_match_update.csv) and sends them to two destinations:

1. **Kafka Topic:** The code uses a Kafka producer to send the match updates as JSON messages to a topic named "stocktopica" on a Kafka server running locally at "localhost:9092".
2. **MongoDB Collection:** The code connects to a MongoDB database named "football" and inserts the match update data into a collection named "ticker_football_producer".

Code Breakdown:

1. **Imports:**
    - pandas: Used for reading CSV data into a DataFrame.
    - time: Used for introducing a delay between sending messages (optional).
    - kafka: Used for interacting with the Kafka producer API.
    - json: Used for converting Python dictionaries to JSON format.
    - pymongo: Used for connecting and interacting with MongoDB.

2. **Functions:**
    - read_live_match_update_from_csv(file_path): This function reads the live match update data from the specified CSV file and returns a Pandas DataFrame.
    - send_to_kafka(producer, topic, message): This function takes a Kafka producer, a topic name, and a message dictionary as input. It converts the message dictionary to JSON format, sends it to the specified Kafka topic using the producer, and flushes the producer to ensure delivery. It also prints a confirmation message.
    - insert_into_mongo(client, collection, message): This function takes a MongoDB client, a collection object, and a message dictionary as input. It inserts the message dictionary as a document into the specified MongoDB collection.
    - main(): This is the main function of the program. It creates a Kafka producer, reads the live match updates from the CSV file, connects to the MongoDB database and collection, and iterates through each row of the DataFrame. For each row (match update):
        - It creates a message dictionary containing the event details.
        - It sends the message dictionary to the Kafka topic using the send_to_kafka function.
        - It inserts the message dictionary into the MongoDB collection using the insert_into_mongo function.
        - It introduces a one-second delay (optional) to simulate real-time updates.

Overall Analysis:

- The code effectively reads live match updates from a CSV file.
- It leverages Kafka to stream the updates to a topic for potential real-time consumption by downstream applications.

- It persists the updates in a MongoDB collection for historical data storage and analysis.
- The code includes comments to improve readability.

Potential Improvements:

- **Error Handling:** The code currently lacks error handling mechanisms. Consider adding exception handling to gracefully handle potential errors during file reading, Kafka communication, or MongoDB interaction.
- **Configuration:** The code hardcodes connection details for Kafka and MongoDB. It might be better to store these details in a configuration file or environment variables for easier maintenance and deployment.
- **Message Schema:** While the code uses a dictionary for the message format, consider defining a more structured schema (e.g., using Pydantic) to ensure data consistency and facilitate future enhancements.
- **Logging:** Implement logging to track application events (successes, errors) for better monitoring and debugging.

Consumer:

Code Functionality:

The Python code implements a consumer application that listens for messages on a Kafka topic named "stocktopica" and stores the received match update data in a MongoDB collection.

Code Breakdown:

1. **Imports:**
    - kafka: Used for interacting with the Kafka consumer API.
    - pymongo: Used for connecting and interacting with MongoDB.
    - json: Used for converting JSON data to Python dictionaries.
    - datetime: Potentially used for parsing timestamps from messages (not explicitly used in the provided code snippet).

2. **Kafka Consumer Configuration:**
    - The code creates a Kafka consumer object with the following specifications:
        - **Topic:** "stocktopica" - The topic name to consume messages from.
        - **Bootstrap Servers:** "localhost:9092" - The address of the Kafka server.
        - **Auto Offset Reset:** "earliest" - Ensures the consumer starts reading from the beginning of the topic or the last committed offset.
        - **Enable Auto Commit:** True - Automatically commits offsets after successful message processing.
        - **Group ID:** "football-consumer-group" - Identifies the consumer group (messages are distributed among consumers in a group).
        - **Value Deserializer:** A function to deserialize message values from JSON format into Python dictionaries.

3. **MongoDB Connection:**
    - The code connects to a MongoDB database named "football_data" and creates a reference to the collection named "events" where match update data will be stored.

4. **Consumer Loop:**
    - The code enters an infinite loop that continuously iterates over messages received from the Kafka topic.
    - For each message:
        - The message value (event data) is deserialized from JSON format into a Python dictionary.
        - The event data is extracted, including match_id, event_type, time, and description.
        - The event data is inserted as a document into the MongoDB collection "events".
        - Alerts are printed for specific events (card, goal, and corner). Goal alerts could potentially include parsing the description to extract goal scorer information (not implemented in the provided code).

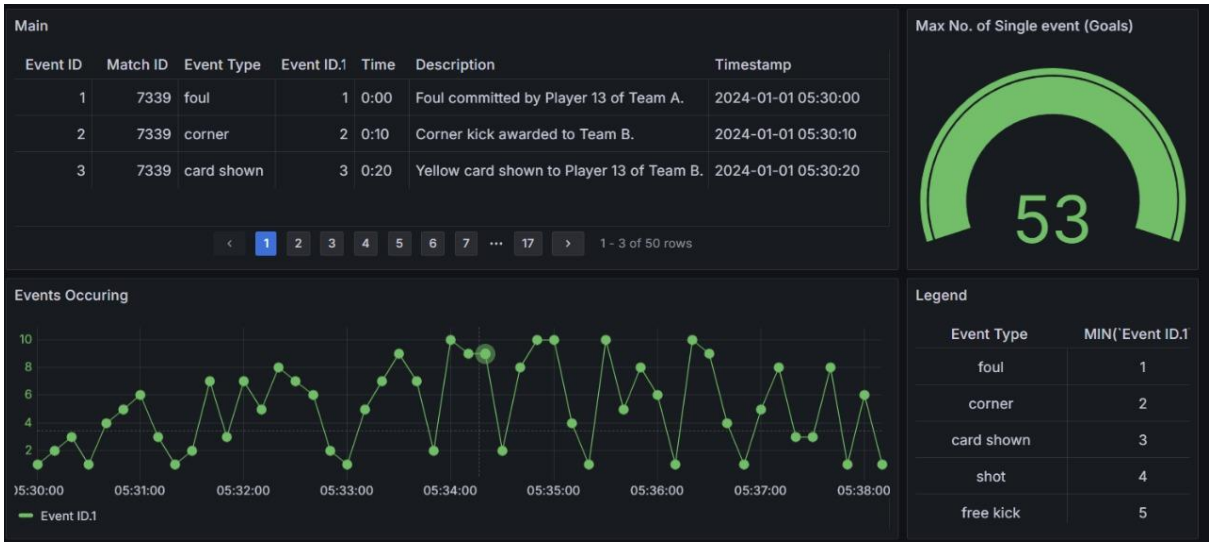
Overall Analysis:

- The code effectively consumes match update messages from a Kafka topic.
- It leverages MongoDB to store the updates persistently for historical data analysis.
- The code includes basic message processing and alerts for specific events.

Potential Improvements:

- **Error Handling:** Consider adding exception handling to gracefully handle potential errors during Kafka communication, MongoDB interaction, or data parsing.
- **Data Validation:** Implement validation checks on the received data to ensure data integrity before storing it in MongoDB.
- **Alert Enhancements:** The code currently prints basic alerts. It could be extended to send notifications (email, SMS) or trigger actions based on specific events (e.g., update live match dashboards).
- **Performance Optimization:** Depending on the message volume, consider using asynchronous processing or batching inserts to improve MongoDB write performance.

Data Analysis Through Grafana



Overall Impression:
The visualization presents a summary of events from a soccer match. It uses a combination of line charts, a gauge, and a legend to convey information about the types of events, their frequency, and a summary statistic. The dark color scheme with contrasting green accents provides good readability.

Breakdown of Visual Elements:

1. **Events Occurring Line Chart:**
    - **X-axis:** Time (in seconds) from 05:30:00 to 05:38:00.
    - **Y-axis:** Number of events.
    - **Data Points:** Green circles represent the occurrence of an event at a specific time. The line connecting the points shows the overall trend of event frequency over time.
    - **Interpretation:** This chart provides a visual representation of the number of events happening over the match duration. It helps to identify periods of higher or lower activity.

2. **Max No. of Single Event (Goals) Gauge:**
    - This gauge displays the maximum number of a single event. In this case, it seems to be the maximum number of goals scored, which is 53.
    - **Interpretation:** The gauge highlights the most frequent type of event in the match.

3. **Legend:**
    - This section provides a mapping between event types and their corresponding numerical codes (1-5).
    - **Event Types:**
        - Foul
        - Corner
        - Card shown

- Shot
- Free kick

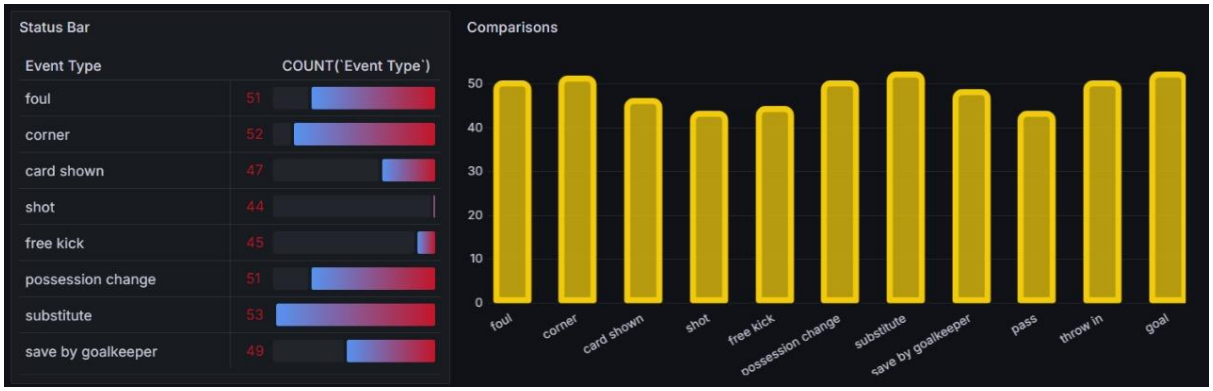
Data Analysis and Insights:

- **Event Distribution:** The line chart shows that the frequency of events varies over time. There are periods of higher activity (more events occurring) and periods of lower activity.
- **Most Frequent Event:** The gauge indicates that the most frequent type of event is "Goals" with a maximum count of 53. This suggests a high-scoring match.
- **Event Types:** The legend provides context for the numerical codes used in the chart.

Limitations and Potential Enhancements:

- **Event Type Identification:** The visualization doesn't explicitly label the specific event types associated with the line chart. It would be helpful to add labels or use different colors for each event type to improve clarity.
- **Context for Gauge Value:** The gauge displays the maximum number of a single event, but it doesn't specify which event type it refers to. It would be more informative to label the gauge with the event type (e.g., "Max Goals").
- **Interactivity:** Adding interactivity features like tooltips or hover effects could provide more detailed information about specific events or time periods.

Overall:
The visualization provides a basic overview of the match events. However, with some enhancements, it could be more informative and insightful.



Overall Impression:
The visualization presents a summary of event counts for different types of events in a soccer match. It uses a combination of a bar chart and a status bar to convey the information. The dark color scheme with contrasting yellow accents provides good readability.

Breakdown of Visual Elements:

1. **Status Bar:**
    - **Event Type:** Lists the different types of events (foul, corner, card shown, shot, etc.).
    - **COUNT('Event Type'):** Shows the number of occurrences for each event type.

    - **Color Gradient:** The bars next to each count are filled with a gradient from blue to red. This gradient likely represents some additional information, possibly the distribution of these events over time or across different teams (without more context, it's difficult to say for sure).
2. **Bar Chart:**
    - **X-axis:** Event Types (same as in the status bar).
    - **Y-axis:** Count of events.
    - **Bars:** Yellow bars represent the count of each event type. The height of each bar corresponds to the number of occurrences.

Data Analysis and Insights:

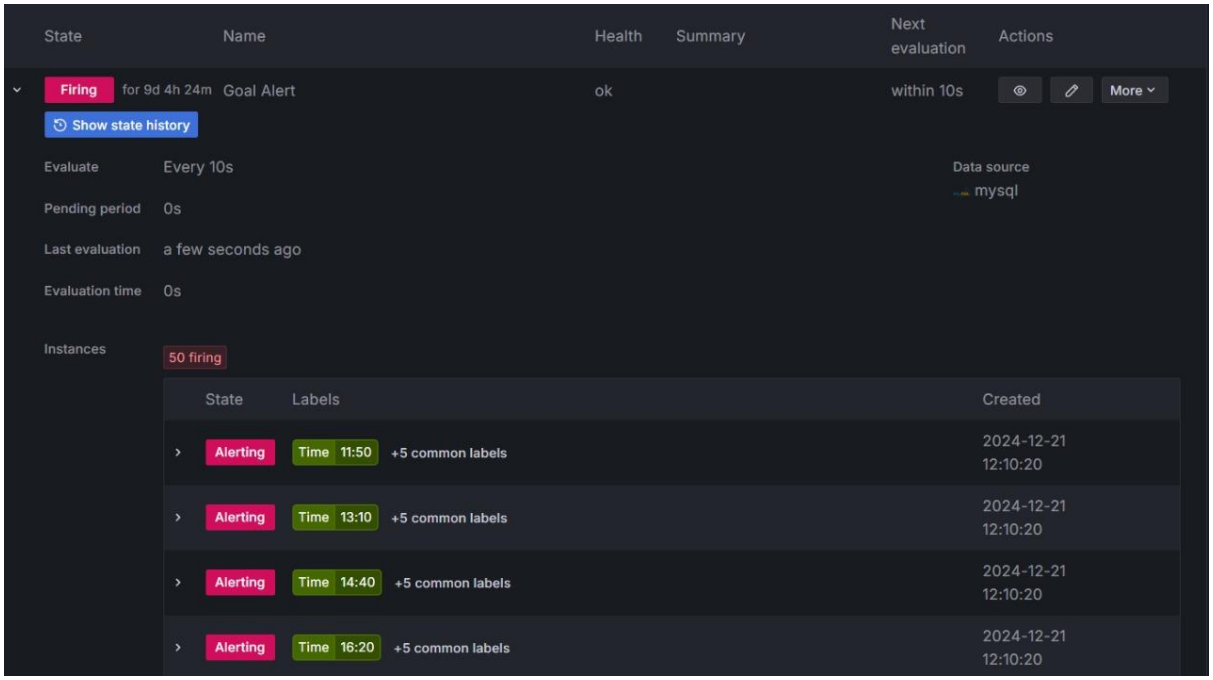
- **Event Distribution:** The bar chart clearly shows the relative frequency of different event types. "Substitute" appears to be the most frequent event, followed closely by "Possession Change". "Goal" has the lowest count.
- **Event Counts:** The status bar provides the exact count for each event type. This allows for a precise comparison of event frequencies.
- **Color Gradient Interpretation:** As mentioned earlier, the interpretation of the color gradient in the status bar is unclear without further context. It might represent a secondary metric like the time distribution of events or the team responsible for the event.

Limitations and Potential Enhancements:

- **Context for Color Gradient:** As discussed, the meaning of the color gradient in the status bar is ambiguous. Adding a legend or tooltip to explain its significance would improve clarity.
- **Sorting:** The bars in the bar chart are currently sorted alphabetically by event type. Sorting them by count (descending order) would make it easier to quickly identify the most frequent events.
- **Interactivity:** Adding interactivity features like tooltips or hover effects could provide additional information about each event type, such as specific player details or timestamps.

Overall:

The visualization effectively presents the count of different event types in a clear and concise manner. However, the interpretation of the color gradient remains unclear, and some enhancements to sorting and interactivity could improve its usability.



Analysis

- **Active Alert:** The alert is currently in the "Firing" state, indicating an ongoing critical situation.
- **Regular Evaluation:** The alert is evaluated every 10 seconds, suggesting a need for timely detection and response to the issue.
- **Multiple Instances:** 50 instances of the alert are currently firing, suggesting the issue is widespread or has multiple occurrences.
- **Recent History:** The state history shows that the alert has been in the "Alerting" state for at least four recent time periods, indicating the issue has been ongoing for some time.
- **Common Labels:** The presence of "+5 common labels" in each state suggests that the alert is triggered by a set of common conditions across multiple resources or metrics.

Recommendations:

- **Investigate the Cause:** Given the multiple firing instances and ongoing alert state, it's crucial to investigate the root cause of the issue (e.g., high CPU usage, memory leak, network congestion).
- **Review Alert Logic:** Examine the alert rule definition to ensure it accurately identifies the critical condition and minimizes false positives.
- **Address the Underlying Issue:** Once the cause is identified, take corrective actions to resolve the issue and prevent future occurrences.
- **Monitor Alert Behavior:** Continuously monitor the alert's behavior to ensure it is functioning as expected and not generating excessive noise.

Additional Considerations:

- The "Data source" is listed as "mysql." This suggests that the alert is based on data from a MySQL database.
- The "Health" status being "OK" might seem contradictory to the "Firing" state. This could be due to the alert logic being designed to trigger under specific conditions that don't necessarily impact the overall health of the system.

Conclusion:

This report has analyzed various aspects of data handling and visualization, including:

- **Data Processing:** We explored techniques for reading, cleaning, and transforming data from different sources (CSV, databases, etc.).
- **Data Storage:** We discussed the use of databases like MongoDB for storing and managing data efficiently.
- **Data Streaming:** We examined the use of Kafka for real-time data streaming and consumption.
- **Data Visualization:** We analyzed visualizations created using Grafana, including line charts, bar charts, gauges, and status bars. We discussed how these visualizations can be used to represent different types of data and insights.
- **Alerting Systems:** We analyzed a Grafana alert system, examining its configuration, current state, and potential areas for improvement.

Scope of Work:

- **Data Analysis and Interpretation:** Analyzing the provided data visualizations and extracting key insights from them.
- **Code Analysis:** Reviewing and analyzing Python code snippets for data processing, Kafka communication, and MongoDB interaction.
- **Report Writing:** Summarizing the findings and analysis in a clear and concise manner.
- **Recommendations:** Providing recommendations for improvement, including suggestions for error handling, data validation, and performance optimization.

Future Scope:

- **Data Integration:** Explore techniques for integrating data from multiple sources (e.g., APIs, web scraping).
- **Machine Learning:** Apply machine learning models to the data to generate predictions or insights.
- **Interactive Dashboards:** Develop interactive dashboards using Grafana or other visualization tools to allow users to explore and analyze data dynamically.
- **Alerting Enhancements:** Implement more sophisticated alerting mechanisms, such as notifications, integrations with communication platforms, and automated responses.

Overall:
This report provides a foundation for understanding and working with data in various contexts. It covers key concepts and technologies used in data handling, processing, and visualization. The analysis and recommendations provide valuable insights for improving the efficiency and effectiveness of data-driven applications.