

# EURO2020 LAMBDA ARCHITECTURE

PER UN PUGNO DI DATI  
ALEXANDRU PAVEL, ANTONIO TURCO

25 luglio 2021

## INDICE

1	Introduzione e obiettivi	3
2	Architettura e stack tecnologico	3
2.1	Docker . . . . .	4
2.2	HDFS . . . . .	5
2.3	Kafka . . . . .	6
2.4	Dask . . . . .	7
2.5	Faust . . . . .	8
2.6	MongoDB . . . . .	9
3	Dataset e preprocessing	10
3.1	API-Football . . . . .	10
3.2	Dataset storico Europei . . . . .	11
4	Ingestion	12
4.1	Topic . . . . .	12
4.2	Formato dati . . . . .	12
5	Analisi Batch	13
5.1	Master dataset . . . . .	13
5.2	Query . . . . .	14
5.3	Persistenza delle view . . . . .	15
6	Analisi Streaming	16
6.1	Parametri finestre . . . . .	16
6.2	Strategia di aggregazione statistiche . . . . .	16
7	Serving layer	17
7.1	Collezioni . . . . .	17
7.2	Dashboard . . . . .	18
8	Conclusioni e sviluppi futuri	20

## ELENCO DELLE FIGURE

Figura 1	Architettura lambda per la gestione dei dati di Euro 2020 . . .	3
Figura 2	Deploy attraverso l'uso di Docker . . . . .	5
Figura 3	Esempio di interazione tra Producer e Consumer Kafka, dal punto di vista dei client . . . . .	6
Figura 4	Esempio di ndarray pronto per essere suddiviso su più nodi .	7
Figura 5	Flusso di una tipica applicazione Dask . . . . .	8
Figura 6	Esempio minimale di un applicazione Faust che sfrutta una Table per la persistenza dei dati e un agente per gestire i nuovi dati da Kafka . . . . .	9
Figura 7	Esempio dell'interfaccia di Mongo charts . . . . .	10
Figura 8	Esempio di ciò che può essere estratto tramite API-Football .	11
Figura 9	Visualizzazione delle viste batch nel serving layer all'interno della dashboard Mongo-Charts . . . . .	18
Figura 10	Numero istantaneo di passaggi effettuati dalle due squadre, raggruppati ad intervalli di 5 minuti (Italia - Inghilterra) . . .	19
Figura 11	Numero totale di passaggi effettuati dalle due squadre, nel- l'arco dell'intera partita (Italia - Inghilterra) . . . . .	19
Figura 12	Esempio di dashboard visualizzabile su Mongo Charts (pas- saggi totali, possesso palla, tiri totali, falli totali) . . . . .	19

## ELENCO DELLE TABELLE

Tabella 1	Differenza tra i dati forniti da API-Football e quelli prepro- cessati considerando i delta delle statistiche tra i diversi mi- nuti della partita (ad esempio passaggi totali. . . . .	13
-----------	---	----

## 1 INTRODUZIONE E OBIETTIVI

In questa relazione verrà illustrata l'architettura lambda creata per gestire i dati provenienti dalle partite dell'ultimo Europeo di calcio.

Tale ambito può beneficiare fortemente da tale trattamento dei dati, infatti si presta in maniera naturale ad avere una tipologia di analisi sia real-time (durante la partita) che batch (pre/post-partita).

Gli obiettivi di tale lavoro quindi sono i seguenti:

- Integrare diverse fonti di dati;
- Definire una pipeline per l'elaborazione dei dati;
- Aggregare e fondere i dati tra streaming e batch;
- Gestire il tutto in maniera scalabile e fault tolerant.

## 2 ARCHITETTURA E STACK TECNOLOGICO

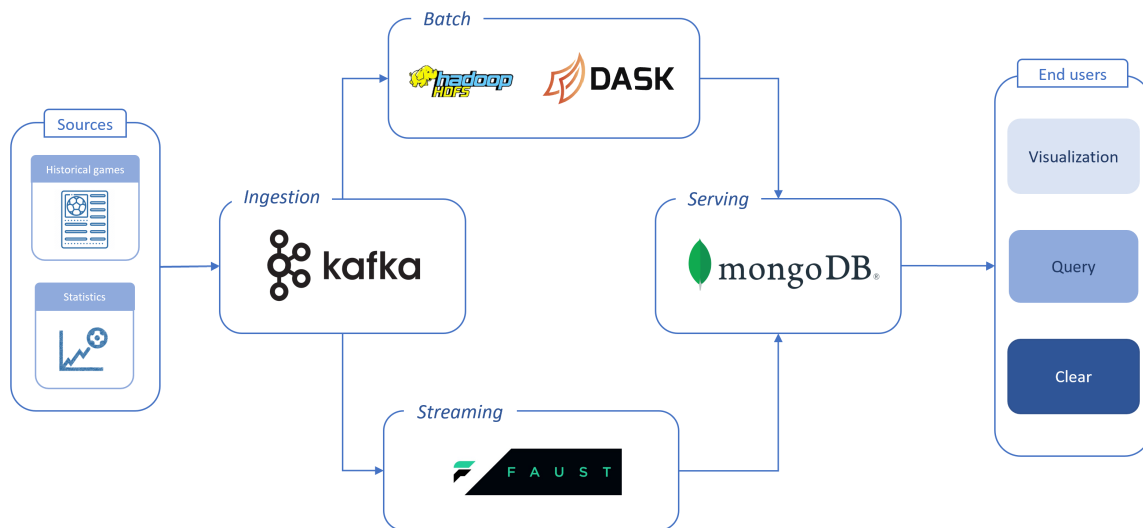


Figura 1: Architettura lambda per la gestione dei dati di Euro 2020

L'architettura lambda permette il processamento di grandi quantità di dati sfruttando due flussi di calcolo. Definendo quindi un batch layer ed uno streaming layer, il primo permette di elaborare query che hanno bisogno di un **alto throughput** mentre il ramo streaming permette di avere risposte alle query con **bassa latenza** [1].

Questi due flussi ricevono i dati da un coda definita nel layer di ingestion. La soluzione adottata sfrutta Kafka per la sua robustezza ai guasti e le proprietà di scalabilità.

Il linguaggio di riferimento che è stato usato nella definizione del progetto è stato Python, dettaglio che ha definito le restanti tecnologie. Infatti nonostante tecnologie come Spark o Hive siano maggiormente indicate per la definizione di tale architettura (in generale tutti i framework del progetto Apache per i Big Data), si è scelto di usare Dask per la parte batch e Faust per la parte streaming. Infatti questi due framework lavorano nativamente con Python e permettono di utilizzare con maggiore flessibilità librerie per processare dati (come Pandas, Numpy, etc..) in un ambiente distribuito e fault-tolerant.

I risultati dei due rami batch e streaming vengono aggregati all'interno del serving layer, che viene gestito attraverso il database NoSQL MongoDB. Il motivo della scelta è dovuto alla possibilità di scalare orizzontalmente all'aumentare del flusso di dati ed anche per una suite di framework interni che sono risultati utili.

Infatti tra i vari strumenti che MongoDB offre, è presente la tecnologia Mongo-Charts. Tale libreria permette di definire con pochi click una **dashboard** che accede alle viste prodotte dai rami batch e streaming, aggiornando periodicamente i grafici mostrati quando le sorgenti ricevono nuovi dati.

Infine l'intero stack tecnologico viene gestito attraverso Docker (tramite docker-compose) per comodità di deploy e di gestione dei vari servizi che devono essere avviati per processare il flusso di dati.

## 2.1 Docker

Docker [2] è uno strumento di virtualizzazione nato per definire un insieme di servizi in maniera tale da essere eseguiti come processi isolati, potenzialmente anche sulla stessa macchina. Di fondamentale importanza per il progetto, infatti è stato possibile definire l'ambiente di sviluppo e di rilascio attraverso poche linee di codice. Il concetto su cui si basa Docker sono i cosiddetti *Container*.

Un container è un unità atomica per Docker che contiene il codice e tutte le sue dipendenze, in questa maniera l'applicazione può essere eseguita rapidamente e in maniera affidabile da un ambiente di sviluppo all'altro.

Quindi attraverso la definizione di un container si ottiene un modulo software leggero e indipendente che contiene tutto ciò che un servizio ha bisogno: codice, strumenti di sistema, librerie esterne e impostazioni.

### 2.1.1 Docker-Compose

Docker-Compose è un sottomodulo di Docker che permette di definire in un unico file un insieme di container tra loro **logicamente collegati**. Infatti basta definire un singolo file YAML per configurare tutti i servizi necessari al progetto. Definito il file, basterà un singolo comando per avviare l'intero progetto ed un altro per terminarne l'esecuzione.

Un esempio di avvio dei vari servizi può essere ottenuto tramite:

```
1 docker-compose up -d
```

Che avvierà tutti i servizi definiti nel *docker-compose.yml* senza altre interazioni da parte dello sviluppatore.

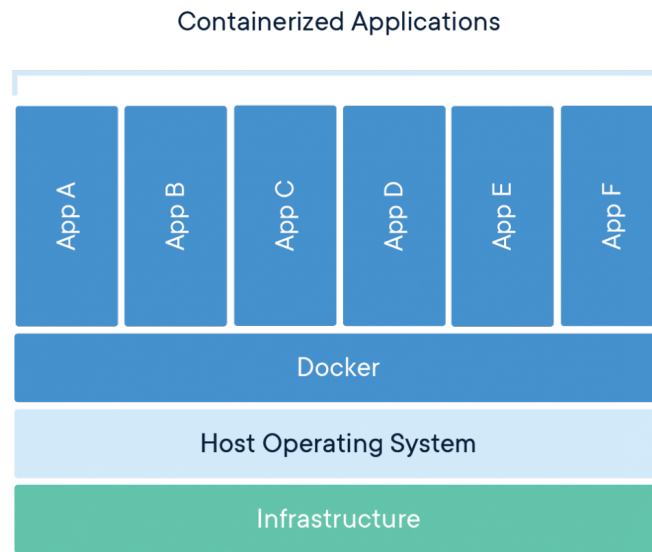


Figura 2: Deploy attraverso l'uso di Docker

## 2.2 HDFS

HDFS è il filesystem distribuito definito dal progetto Apache Hadoop scritto in Java. Permette la memorizzazione di file in maniera scalabile tramite l'utilizzo di cluster. Attraverso questa tecnologia è quindi possibile definire un *data store* in grado di mantenere memorizzati i dati in maniera replicata su più nodi.

Alla base di HDFS si individuano i seguenti servizi:

- **Name Node**, Permette di tener traccia dei file e quindi gestire l'intero file system. Mantiene memorizzati i vari metadati utili per la ricostruzione della topologia del sistema. In particolare questo nodo ha la responsabilità di mantenere informazioni riguardo il numero di blocchi, la loro locazione e il fattore di replicazione di quest'ultimi. Attraverso il Name Node il client viene instradato verso il corretto Data Node
- **Data Node**, memorizza i dati effettivi in blocchi di memoria. Il client quindi deve comunicare con i Data Node per ottenere i dati veri e propri. In questa architettura, il Data Node rappresenta il nodo *slave/worker*, che deve mandare un messaggio di *Heartbeat* ad intervalli regolari in modo tale che il nodo *master* (Name Node) possa tener traccia dello stato del filesystem. Tecnica utile soprattutto per capire se il Data Node fallisce nell'erogare il proprio servizio e in tal caso sostituirlo con un altro nodo del cluster attivo.

### 2.3 Kafka

Kafka [3] è una tecnologia open-source del gruppo Apache, nata per definire una **coda eventi** scalabile in maniera totalmente distribuita. Quindi è uno strumento utile per la definizioni di pipeline per il trattamento dei dati ad alte performance in ambito *analytics*.

In questo caso d'uso viene utilizzata in particolare per l'ingestion dei dati, poichè permette di disaccoppiare chi produce i dati da chi li legge. Così da poter cambiare facilmente i vari moduli software che hanno le responsabilità di lettura/scrittura dei dati di Kafka. Inoltre i dati vengono inviati/letti in maniera totalmente **asincrona** garantendo alte performance, siccome non si verificherà mai la situazione in cui un processo client dovrà attendere la terminazione di un altro.

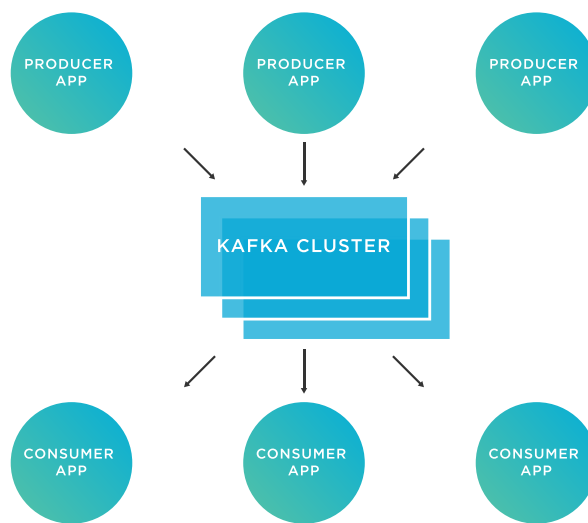


Figura 3: Esempio di interazione tra Producer e Consumer Kafka, dal punto di vista dei client

I concetti alla base per poter definire una pipeline Kafka sono i seguenti e sono riassunti nell'immagine 3:

- **Eventi**, registra che qualcosa è avvenuto all'interno del dominio applicativo. Sono i record che vengono registrati all'interno della coda Kafka e normalmente sono identificati da chiave, valore, timestamp e ulteriori metadati accessori;
- **Topic**, Definiscono logicamente una serie di eventi collegati tra loro. Normalmente a questa relazione logica ne corrisponde una fisica in modo tale da mantenere record simili vicini tra loro. Un topic quindi può essere suddiviso in varie partizioni così da garantire la scalabilità orizzontale nell'architettura distribuita. Un topic può essere memorizzato anche in maniera persistente, così da trasformare, se necessario, Kafka in una sorta di database No-SQL chiave-valore.
- **Produttori**, Sono dei client Kafka che possono scrivere su uno o più topic della coda.

- **Consumatori**, Sono dei client Kafka che possono leggere da uno o più topic della coda. Per poter leggere i dati del topic bisogna prima sottoscrivere a quel topic, notificandolo al server Kafka, in maniera simile a quanto avviene nel pattern *Observer*.

## 2.4 Dask

Dask [4] è un framework open-source scritto in Python che permette la definizione di pipeline per l'analytics su larga scala. Tale strumento si inserisce perfettamente nell'**ecosistema Python**, offrendo dei wrapper alle librerie più comuni (come Pandas e NumPy) e per scalare orizzontalmente all'occorrenza.

Oltre alla definizione di questi wrapper, Dask definisce un suo scheduler che è centrale per centrare gli obiettivi di scalabilità di questo tool. Tale scheduler non è fisso, ma può essere cambiato in base al contesto. Infatti ne esistono diverse implementazioni di questo che permettono di gestire comodamente anche casi iniziali in cui la mole di dati non è ancora enorme. Permettendo quindi la definizione di una pipeline Dask anche su laptop, mantenendo però l'interfaccia uguale per ogni implementazione e quindi appena il carico di lavoro inizia ad aumentare ci si può spostare comodamente su server senza dover cambiare quasi nulla.

Attraverso la libreria *dask.delayed*, permette di trasformare qualsiasi funzione Python in una funzione Dask, in modo tale da trasformare senza troppi problemi anche vecchie codebase in codice pronto per lavorare su larga scala.

	Array	Chunk
<b>Bytes</b>	11.75 GiB	1.47 GiB
<b>Shape</b>	(1576800000,)	(197100000,)
<b>Count</b>	8 Tasks	8 Chunks
<b>Type</b>	float64	numpy.ndarray

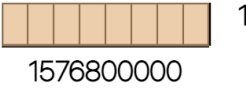


Figura 4: Esempio di ndarray pronto per essere suddiviso su più nodi

Nell'immagine 4 si può osservare come un classico *numpy.ndarray* venga suddiviso in **chunks** in modo tale da poter gestire array di grandi dimensioni su più nodi. Quindi ogni nodo vedrà un classico array NumPy mentre lo scheduler gestirà tutta l'integrazione necessaria per fare merge dei singoli chunk.

L'immagine 5 chiarisce la pipeline di un job scritto in Dask. Nella parte iniziale si definisce attraverso la libreria Dask il grafo di computazione in maniera **lazy**, ovvero si definisce solo quali funzioni devono essere chiamate e in quale ordine. L'output di questa fase è il **Task Graph** che viene dato in input allo scheduler. Quest'ultimo elemento ha quindi la responsabilità di eseguire realmente le funzioni richieste dal grafo, distribuendole tra i vari nodi worker.

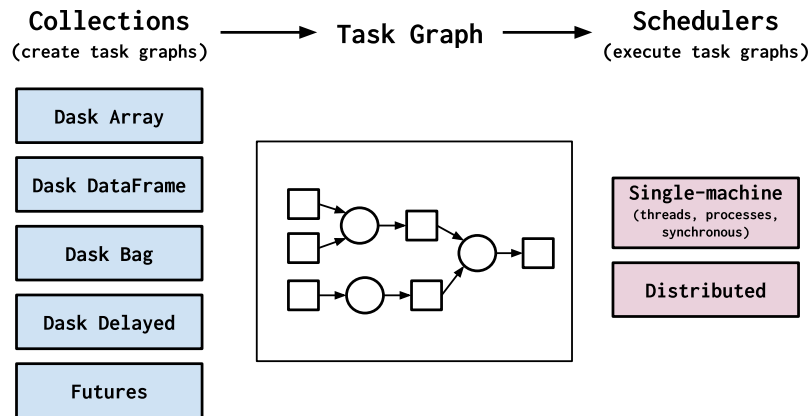


Figura 5: Flusso di una tipica applicazione Dask

## 2.5 Faust

Faust [5] è una libreria nata per definire applicazioni streaming nativamente in Python. Come con Dask, la possibilità di lavorare nativamente con Python ne permette di integrarsi comodamente con l'ecosistema del linguaggio. Un altro vantaggio per questa soluzione è anche la compatibilità piena con Kafka, infatti funziona out-of-the-box direttamente con Kafka senza dover installare altri package. L'unico difetto di questo strumento è che non è retrocompatibile con tutte le versioni di Python precedenti alla 3.6.

La libreria si basa sul concetto di **agente** che definisce un consumer Kafka per uno specifico topic. Definito l'agente il framework è responsabile per tutto ciò che concerne l'instaurazione della connessione con Kafka e la ricezione dei dati. Quindi lato client basta definire un loop in cui viene specificato cosa bisogna fare con ogni nuovo evento della coda.

Faust mette a disposizione alcune strutture dati come le *Tables*, che permettono di mantenere uno stato e di agire come un database. Infatti tali strutture dati sono dizionari chiave/valore distribuiti che sfruttano come motore RocksDB, sistema No-SQL nato per applicazioni embedded e scritto in C++. Quindi un sistema nato con l'idea di **minimizzare la latenza** durante il processamento dei dati.

Tali tabelle possono ovviamente contenere aggregati di ciò che viene memorizzato durante la lettura della coda Kafka, in particolare è possibile definire *finestre*. In questa maniera si possono effettuare le tipiche operazioni presenti in Spark Streaming, dove si calcolano aggregati solo su porzioni temporali del dato. Tali finestre possono essere anche sovrapposte tra loro con l'opzione *hopping* da definire alla creazione della finestra.

L'immagine 6, presa dalla documentazione ufficiale, permette di osservare come sia semplice definire un consumer Kafka con poche righe di codice per il calcolo dei click per ogni URL. A dimostrazione di quanto sia semplice con questo framework definire pipeline streaming su Kafka.



```
# data sent to 'clicks' topic sharded by URL key.
# e.g. key="http://example.com" value="1"
click_topic = app.topic('clicks', key_type=str, value_type=int)

# default value for missing URL will be 0 with `default=int`
counts = app.Table('click_counts', default=int)

@app.agent(click_topic)
async def count_click(clicks):
    async for url, count in clicks.items():
        counts[url] += count
```

Figura 6: Esempio minimale di un applicazione Faust che sfrutta una Table per la persistenza dei dati e un agente per gestire i nuovi dati da Kafka

## 2.6 MongoDB

MongoDB [6] è un database NoSQL document-oriented che utilizza un formato simile al JSON per mantenere i dati in memoria. Ormai è una delle soluzioni non relazionali più robuste sul mercato, con un ecosistema interno che permette di definire diverse pipeline sulle viste presenti in memoria.

Al suo interno MongoDB si basa sulla definizione di tre entità:

- **Database**, come nelle soluzioni relazionali rappresenta un contenitore di tabelle che in MongoDB sono chiamate collezioni. Per attivare un istanza in particolare basta usare il comando: *use <nome-db>*
- **Collezione**, rappresenta una collezione di documenti. Può essere pensata come una tabella dei sistemi tradizionali relazionali. Una collezione non impone nessuno schema sui documenti contenuti e quindi i vari dati memorizzati possono presentare campi differenti. Quindi l'unica *best practice* consigliata da MongoDB è quella di memorizzare in una collezione solo documenti che sono collegati tra loro logicamente.
- **Documento**, è l'unità di base di memorizzazione. Sono memorizzati attraverso un formato *JSON-like* chiamato *BSON*. Tale formato permette una memorizzazione binaria dei dati JSON e quindi permette di utilizzare strutture dati complesse all'interno dei campi del documento. In generale questo formato binario è progettato per essere efficiente sia per lo spazio richiesto dai dati, sia per la velocità di ricerca, mantenendo però la leggibilità del formato JSON.

### 2.6.1 Mongo Charts

Tra i tanti strumenti presenti nella suite Mongo, uno dei più interessanti è sicuramente Mongo Charts. Tale strumento permette di definire una dashboard di **visualizzazione dei dati** tramite un interfaccia UI intuitiva come si può vedere nell'immagine 7. Infatti basta definire una serie di sorgenti da cui il tool può ricevere dati e automaticamente inferirà lo schema presente nelle varie collezioni. A questo punto è possibile definire quale grafico si vuole utilizzare e indicare quali campi vogliono essere messi in relazione. Fatto questo si otterrà una visualizzazione dei

dati richiesti, che verrà aggiornata ogni qual volta viene aggiornata la sorgente dati da cui *Charts* dipende.

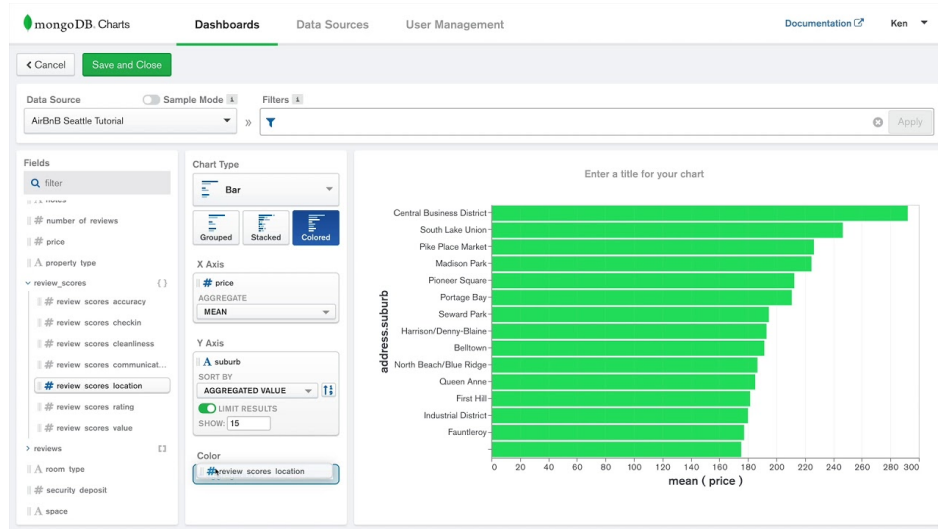


Figura 7: Esempio dell'interfaccia di Mongo charts

Questo strumento è quindi di fondamentale importanza per permettere anche a chi non è un esperto di manipolare dati ed estrarre grafici che possono aggiungere valore all'analisi di un dataset. Infatti qualsiasi cambiamento che non modifichi lo schema della collezione non avrà impatto sulla pipeline definita nella visualizzazione. Quindi un eventuale figura non tecnica una volta che definisce la regola tramite interfaccia grafica per la visualizzazione si troverà il grafico aggiornato automaticamente ogni volta che i dati all'interno della collezione sono modificati.

### 3 DATASET E PREPROCESSING

Prima di entrare nel dettaglio dell'architettura lambda, è bene chiarire la fonte dei dati utilizzati e i vari step di preprocessing effettuati per ottenere un input più adatto per gli scopi della pipeline.

Infatti molti dei dati trovati in rete erano in un formato già aggregato e/o erano il report di un'analisi già effettuata su dati grezzi non disponibili pubblicamente.

Per simulare quindi un formato di dati più grezzo, si è deciso di prendere alcune di queste fonti e riportarle ad uno stato meno aggregato, così da poter ripetere interrogazioni classiche dell'ambito sull'architettura lambda.

Il formato dati di questi dataset viene mostrato nelle prossime sezioni.

#### 3.1 API-Football

L'ambito sportivo non è molto aperto nel diffondere dati di dettaglio relativi a partite di livello professionistico. Questo perché i dati di ogni partita sono di alto valore per le varie squadre, permettendo di avere un vantaggio strategico. Per

questo motivo è stato difficile trovare dati a più basso livello (come la posizione in tempo reale dei giocatori, anche in relazione agli eventi della partita) accessibili pubblicamente, relativi a un game feed per la parte streaming della pipeline.

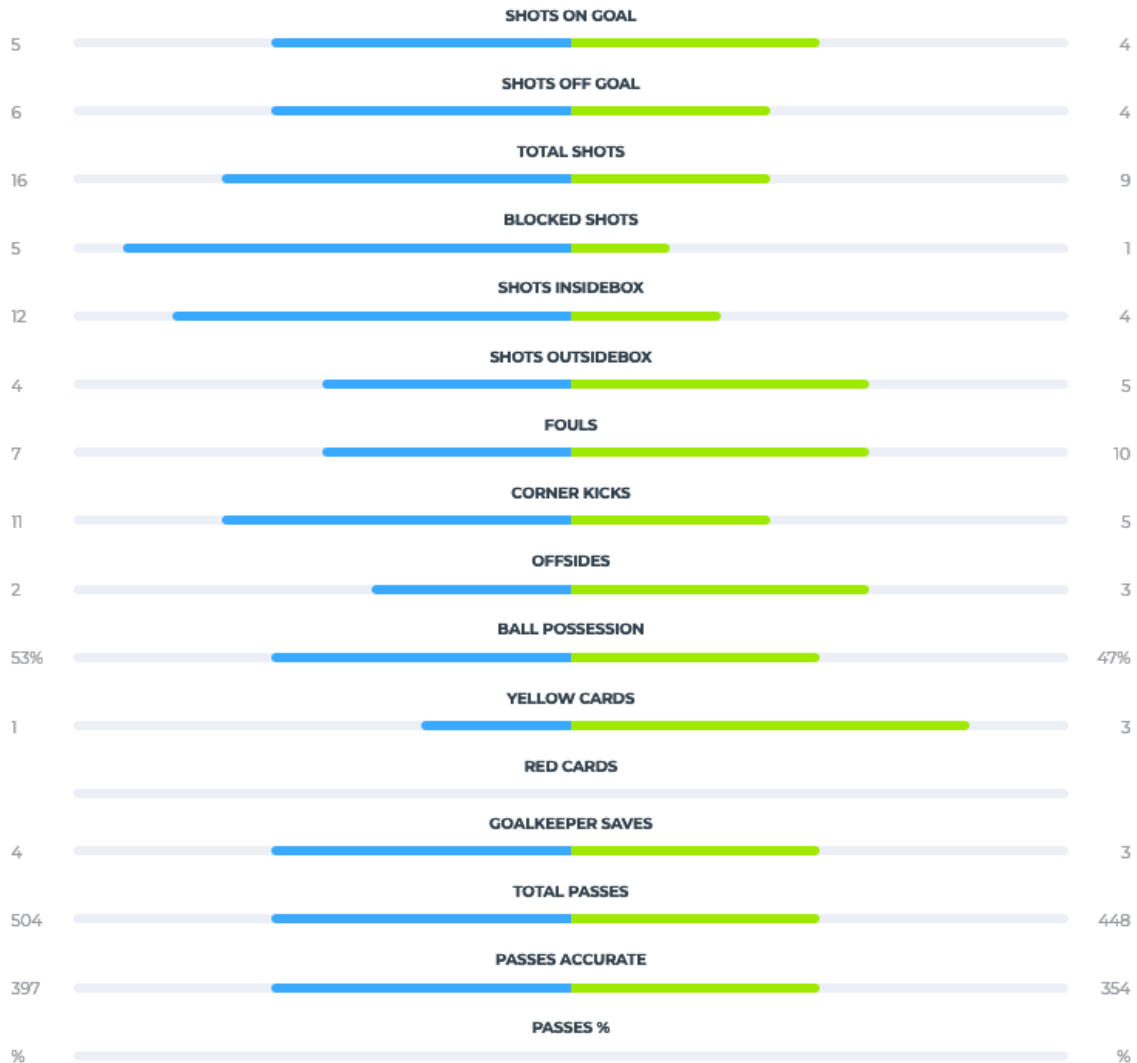


Figura 8: Esempio di ciò che può essere estratto tramite API-Football

Fortunatamente *API-Football* [7] permette di fare 100 chiamate al giorno (gratuitamente) sui propri endpoint. In particolare è stato interessante raccogliere i dati durante le varie partite dell'europeo riguardanti le statistiche del match. Sfortunatamente questi dati venivano già aggregati con l'avanzare della partita (come si vede in figura 8), quindi si è dovuto estrarre il delta tra ogni minuto ottenuto di statistiche, così da simulare gli eventi della partita minuto per minuto.

### 3.2 Dataset storico Europei

Questo Dataset [8] contiene i dati di quasi tutte le partite interazionali a partire dal 1880. In particolare per l'europeo contiene quasi tutte le partite tolte le prime edizioni del periodo 1960-1970. Da questo dataset quindi si è scelto di rimuovere

tutte le partite che non riguardassero le partite degli Europei o delle qualificazioni allo stesso. Dopodiché per permettere un processamento streaming di questi dati, si è deciso di dividere il dataset in due parti. La prima contenente tutti i dati delle edizioni passate, mentre la seconda contenente solo i dati dell'ultima edizione (ovvero Euro 2020). In questa maniera la prima parte potrà simulare il master dataset all'interno della parte batch mentre la seconda parte dovrà essere mandata tramite Kafka e appesa sempre al master dataset.

## 4 INGESTION

Come già descritto in precedenza in questa parte della pipeline si è deciso di usare Kafka. In questa sezione quindi si descriveranno le scelte progettuali relative all'utilizzo di questa tecnologia.

### 4.1 Topic

Per gestire le varie tipologie di dato sono stati definiti i seguenti topic:

- **fixtures**, dove sono stati inviati tutti i dati relativi alle partite completate durante l'ultima edizione dell'europeo (un evento per partita).
- **fixture\_<id>**, dove sono stati inviati i dati di API-Football relativi alle statistiche di un singolo match (identificato da un id). Quindi viene allocato un topic per ogni partita (in particolare cinque, con un centinaio di eventi a partita), in modo tale da separare logicamente i dettagli dei vari match.

### 4.2 Formato dati

Il **dataset storico** contiene principalmente un insieme di file tutti in formato csv, che condividono il seguente schema:

- **Date**: data in cui la partita ha preso evento
- **Home team**: nome della squadra di casa
- **Away team**: nome della squadra fuori casa
- **Home score**: goal segnati dalla squadra in casa
- **Away score**: goal segnati dalla squadra in trasferta
- **Tournament**: torneo di cui la partita fa parte. Per l'europeo sono di interesse le categorie 'Uefa Euro' e 'Uefa Euro Qualifications'
- **City**: città in cui si è disputata la partita
- **Country**: nazione in cui si è disputata la partita

- **Neutral:** flag booleano che indica se la partita si è giocata in campo neutro o meno

Il formato dei dati che viene estratto da **API-Football** invece è un file JSON con i campi uguali a quelli indicati nell'immagine 8. In particolare si tratta di statistiche (come possesso palla, passaggi effettuati, etc..) relative alle due squadre che stanno giocando la partita, relative al minuto di gioco in cui vengono estratte.

Come già accennato questi dati sono stati riportati ad un livello più grezzo (si veda la tabella 1), ovvero considerando ad esempio il numero istantaneo di passaggi effettuati, o di variazione di possesso palla, piuttosto che considerare il valore totale o aggregato nell'arco della partita.

Questo preprocessing ha consentito di avere più flessibilità nella gestione dei dati, permettendo di effettuare diverse analisi, ad esempio valutando i momenti della partita in cui le statistiche sono variate più significativamente. Inoltre con una opportuna gestione delle window (vedere sezione 6) è possibile ottenere l'aggregato delle statistiche che aumenta durante l'arco della partita.

Minute	1	2	3	4	5
Original	0	10	50	100	120
Preprocessed	0	10	40	50	20

**Tabella 1:** Differenza tra i dati forniti da API-Football e quelli preprocessati considerando i delta delle statistiche tra i diversi minuti della partita (ad esempio passaggi totali).

## 5 ANALISI BATCH

L'analisi batch dell'architettura lambda è caratterizzata dal bisogno di processare grandi moli di dati. Per fare questo è ovvio che nel trade-off bisogna sacrificare la latenza nella risposta ma si può cercare di massimizzare il throughput del sistema per renderlo più efficiente. Come già descritto precedentemente, questa parte dell'architettura si appoggia a Dask.

### 5.1 Master dataset

Il master dataset nell'architettura rappresenta la fonte dati delle interrogazioni batch. Questo dataset non deve essere mai modificato se non con operazioni in append, così da garantire che i dati vengano tenuti in memoria così come sono stati ricevuti e permettere quindi operazioni di rollback delle viste prodotte. Per ottenere questo risultato in Dask, è bastato inserire le varie partizioni dei dati relativi alle partite in una cartella unica.

A questo punto è stato possibile leggere i vari csv all'interno della cartella come un unico dataset grazie all'istruzione `dask.read_csv('master_dataset/*')`.

Infatti attraverso l'operatore star, è possibile indicare la lettura di un'intera cartella come Dataframe in Dask. Questo ha permesso al codice di rendersi indipendente dalla metodologia di memorizzazione del dataset (tranne per il formato) e di trattarlo come un normale Dataframe Pandas.

## 5.2 Query

Per testare l'efficacia del framework batch, si è deciso di realizzare differenti query. Queste query non sono pensate per ottenere effettivamente un'analisi di valore all'interno del dominio, ma solo per dimostrare l'efficacia dello strumento e la facilità con cui si possono aggiungere ulteriori viste all'interno dell'architettura.

Generalmente quello che si può notare in quasi tutti questi script, è la necessità di avviare la connessione con lo scheduler Dask sulla porta 8786 e la chiamata al metodo `compute()` per avviare il calcolo del grafo in ambiente distribuito.

Tutte le interrogazioni sfruttano l'API ad alto livello di Dask per chiamare il wrapper per i Dataframe. Si può notare infatti che se si tolgono le chiamate allo scheduler il codice presentato è praticamente identico a codice Pandas classico.

Questa grande facilità di utilizzo permette quindi anche a sviluppatori non esperti in ambito distribuito di poter scrivere interrogazioni batch senza grossi problemi.

Infine si vuole far notare che il codice di queste interrogazioni è privo della parte di persistenza su MongoDB. Questo perché i dati prodotti da Dask sono strutture dati Python standard e quindi la memorizzazione può avvenire comodamente con i metodi disponibili su librerie come PyMongo. Quindi per evitare di appesantire troppo la notazione verrà mostrato un unico esempio nella sezione 5.3 relativo alla persistenza delle query in MongoDB.

### 5.2.1 Goal totali

In questa query si vogliono ottenere i goal totali realizzati da ogni squadra in tutte le edizioni degli europei.

```

1 from dask.distributed import Client
2 import dask.dataframe as dd
3
4 client = Client('tcp://localhost:8786')
5 fixtures = dd.read_csv('master_dataset/fixtures/*.csv')
6
7 home_goals = fixtures.groupby('home_team').home_score.sum()
8 away_goals = fixtures.groupby('away_team').away_score.sum()
9
10 total_goals = home_goals.merge(away_goals)
11 total_goals['score'] = total_goals['home_score'] + total_goals['away_score']
12
13 total_goals = total_goals.drop('home_score', axis=1)
14 total_goals = total_goals.drop('away_score', axis=1)
15
16 results = total_goals.compute()
```

### 5.2.2 Partite ospitate

In questa query si vuole estrarre quale nazione ha ospitato più partite durante l'europeo. Si noti che non per forza quella nazione ha giocato quella partita, tale circostanza si verifica quando la nazione ospita una certa edizione dell'europeo.

```
1 from dask.distributed import Client
2 import dask.dataframe as dd
3
4 client = Client('tcp://localhost:8786')
5 fixtures = dd.read_csv('master_dataset/fixtures/*.csv')
6
7 host_counted = fixtures.groupby(['country']).country.count()
8
9 host_counted = host_counted.compute()
```

### 5.2.3 Partite giocate in casa

In questa query si vogliono restituire per ogni nazione, il numero di partite giocate in casa e il numero di partite giocate in trasferta.

```
1 from dask.distributed import Client
2 import dask.dataframe as dd
3
4 client = Client('tcp://localhost:8786')
5 fixtures = dd.read_csv('master_dataset/fixtures/*.csv')
6
7 home_games = fixtures.groupby(['home_team']).home_team.count()
8 away_games = fixtures.groupby(['away_team']).away_team.count()
9 home_away_games = home_games.merge(away_games)
10
11 home_away_games = home_away_games.compute()
```

## 5.3 Persistenza delle view

In questa sezione si vuole mostrare un codice di esempio per la persistenza dei risultati prodotti da Dask in MongoDB. Per fare questo si è scelto di usare la libreria PyMongo che permette di usare la parte client di MongoDB da Python.

```
1 import pymongo
2
3 client = pymongo.MongoClient("localhost",27017)
4
5 def save_dask_aggregate(aggregate):
6     db = client.batch_view
7     collection = db.aggregate_collection
8
9     for record in aggregate:
10         collection.insert_one(record.to_dict())
```

Come si può vedere dal codice si è scelto di salvare tutte le viste batch all'interno del database *batch\_view*, e ogni interrogazione viene salvata nella sua collezione specifica. Per esempio la prima query viene salvata in *batch\_view.total\_goals*.

Infine si noti la chiamata a `to_dict()` che astrae il fatto che per ogni record bisogna scegliere il formato di memorizzazione e quindi lo schema per la specifica collezione. Quindi questa funzione maschera il tipo di schema scelto per i vari record da ogni interrogazione.

## 6 ANALISI STREAMING

Nell'analisi streaming come già indicato prima si è deciso di usare Faust. In questa sezione quindi verranno indicate le varie scelte progettuali relative all'uso di questa tecnologia per la gestione del flusso streaming inviato da Kafka.

### 6.1 Parametri finestre

Grazie a Faust si possono definire finestre per osservare porzioni del dato streaming di interesse. In questa soluzione si è deciso di definire due tipologie di finestre:

- Finestra di un minuto: prende principalmente i dati come arrivano e li persiste in MongoDB. Lasciando a quest'ultimo nella parte di visualizzazione l'onere di definire la politica di aggregazione (ad esempio per monitorare le statistiche istantanee della partita raggruppandole a gruppi di 5 minuti). In questo modo è possibile analizzare i singoli momenti della partita per osservare di quanto sono variate le statistiche.
- Finestra sull'intera partita: si prendono tutti i dati che arrivano in streaming per una determinata partita e vengono aggregati minuto per minuto (sommando ad esempio i passaggi effettuati in modo incrementale), così da dare in output l'andamento generale della partita sempre aggiornato fino all'ultimo evento ricevuto.

### 6.2 Strategia di aggregazione statistiche

Indipendentemente dalla tipologia di finestra, il codice per la gestione delle aggregazioni per le statistiche si comporta nella maniera indicata di seguito:

```

1 import faust
2
3 app = faust.App(
4     'statistics_stream',
5     broker='kafka://localhost:9092'
6 )
7
8 # WINDOW_SIZE = 1 oppure 100 minuti
9 # WINDOW_STEP = 1 minuto
10
11 topic_<id> = app.topic('fixture_<id>') # una variabile per ogni topic di Kafka
12

```



```

13 game_stats_table_<id> = app.Table('game_stats_<id>', default=int)
14     .hopping(WINDOW_SIZE, WINDOW_STEP, key_index=True)
15
16 def extract_stats(game_stats_table, stats):
17     for key, value in stats.items():
18         game_stats_table[key] += value # somma i valori della singola statistica
19         nella finestra corrente
20
21 @app.agent(topic_<id>)
22 async def fixture_<id>(minutes):
23     async for event in minutes.events():
24         data_topic = event.message.topic
25         stats = event.value # tutte le statistiche di un singolo minuto di gioco
26         extract_stats(game_stats_table_<id>, stats)
27         persist_table(game_stats_table_<id>, data_topic)

```

Si può nuovamente notare come il codice, tolto il ciclo di lettura (righe 20-26), non si discosta molto da un classico codice Python. Infatti le varie *Table* distribuite di Faust hanno un'interfaccia che si comporta quasi come un dizionario Python (sono basate su RocksDB). L'unica differenza anche qui, come in Dask, risiede nel dover inizializzare l'ambiente del framework. Una volta che questa operazione è conclusa, lo sviluppatore può scrivere codice con lo stesso paradigma di Python.

Anche in questo caso la parte di memorizzazione in MongoDB non viene esplicitamente mostrata per brevità, infatti anche qui si può usare un codice simile a quello presente nella sezione 5.3. Questo sempre perché una volta che avviene la computazione in ambito distribuito si può memorizzare il dato con ad esempio PyMongo.

## 7 SERVING LAYER

L'intera architettura trova il punto d'incontro nel serving layer. In questo strato bisogna integrare i flussi ricevuti sia dalla parte batch che dalla parte streaming e presentarli agli utenti finali che dovranno effettuare analisi sui vari report prodotti dalle query.

Per semplificare questo tipo di analisi e disaccoppiare le competenze necessarie all'accesso ai dati, si è deciso di definire una dashboard tramite interfaccia grafica. In questa maniera chiunque voglia accedere ai dati, non dovrà possedere un bagaglio tecnico elevato, ma gli basterà capire come funziona la UI proposta da Mongo Charts. L'effetto è quindi una democratizzazione all'accesso dei report prodotti dall'architettura anche dal personale non tecnico, con un conseguente aumento del valore dei report prodotti.

### 7.1 Collezioni

Per chiarire il contenuto che si può osservare all'interno di MongoDB, si vogliono ricapitolare qui le collezioni che vengono memorizzate:

- `batch_view.total_goals`, memorizza i goal totali come descritto nella sezione [5.2.1](#)
- `batch_view.total_hosts`, memorizza le partite ospitate come descritto nella sezione [5.2.2](#)
- `batch_view.home_away_games`, memorizza il rapporto partite in casa/trasferita come descritto nella sezione [5.2.3](#)
- `streaming_view.fixture_<game-id>`, memorizza le statistiche processate in streaming come descritto nella sezione [6.2](#)

## 7.2 Dashboard

La dashboard come già descritto precedentemente è basata su Mongo-Charts. I dati vengono estratti, in maniera automatica, dalle batch view ogni ora mentre dalle streaming view ogni minuto.

**BATCH CHARTS** In figura 9 si può osservare un esempio di grafici estratti dalle batch view. Come si può vedere è possibile definire con poche interazioni i classici grafici come bar plot, pie charts, mosaic plots, etc... Infine una volta definita la pipeline è possibile scegliere la palette di colori e i nomi dei campi per rendere la visualizzazione ancora più efficace.

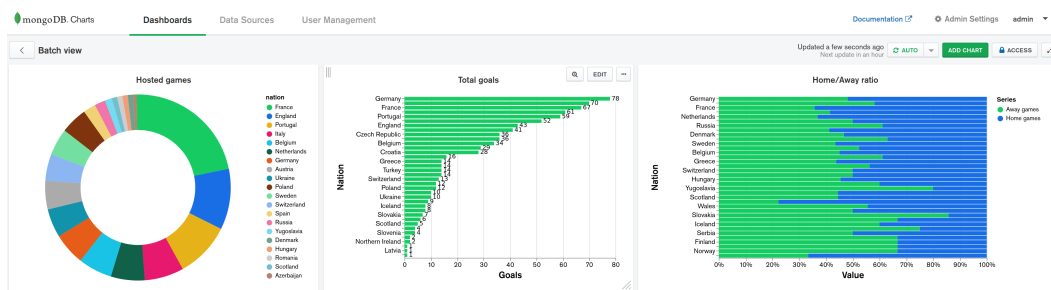


Figura 9: Visualizzazione delle viste batch nel serving layer all'interno della dashboard Mongo-Charts

**STREAMING CHARTS** Per analizzare l'utilità di questo strumento in ambito streaming si è deciso di mostrare una porzione di dashboard che è stata estrapolata durante l'elaborazione dei dati della partita *Inghilterra - Italia*. Il grafico 11 mostra come i dati aggregati su una finestra di 90 minuti vengono visualizzati mano a mano che il flusso viene aggiornato. In questa maniera è facile percepire il trend della partita e capire quale squadra si è comportata meglio rispetto un certo aspetto (in questo caso il totale dei passaggi).

Nel grafico 10 si mostrano invece meglio le potenzialità di Mongo-Charts, dove si è preso lo stesso dato e si sono definiti dei *bin* di 5 minuti l'uno. Questo permette di osservare, in determinate finestre di interesse, quale squadra stia avendo la meglio su una certa statistica.

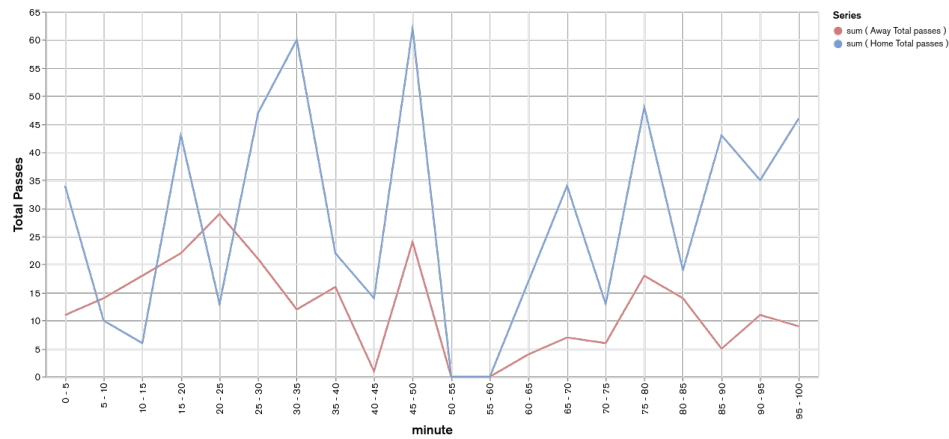


Figura 10: Numero istantaneo di passaggi effettuati dalle due squadre, raggruppati ad intervalli di 5 minuti (Italia - Inghilterra)

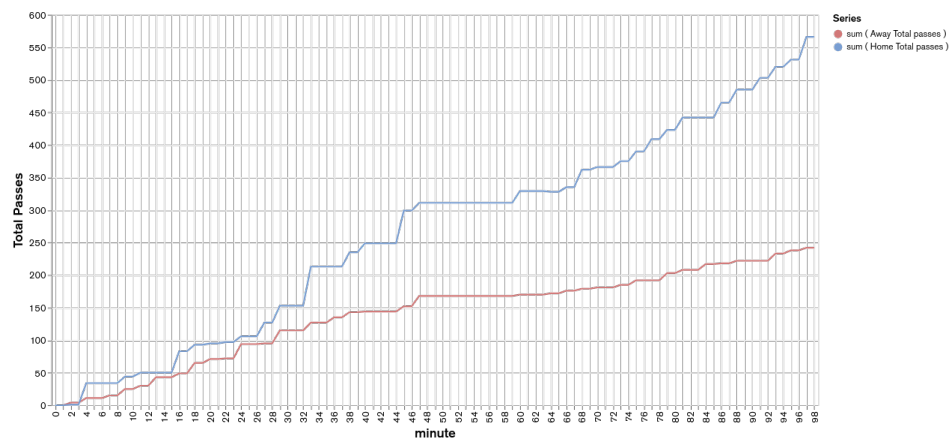


Figura 11: Numero totale di passaggi effettuati dalle due squadre, nell'arco dell'intera partita (Italia - Inghilterra)

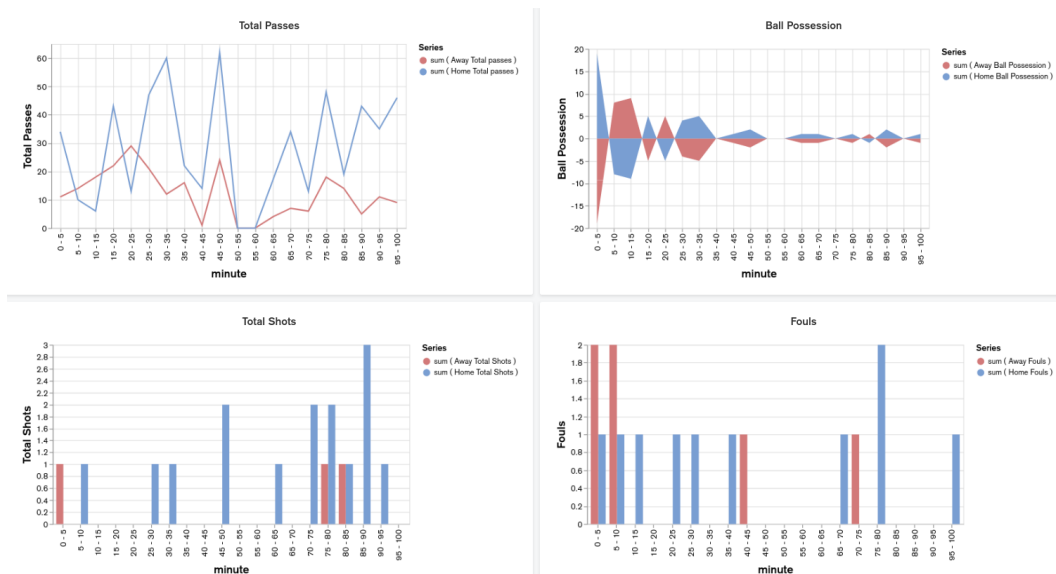


Figura 12: Esempio di dashboard visualizzabile su Mongo Charts (passaggi totali, possesso palla, tiri totali, falli totali)

## 8 CONCLUSIONI E SVILUPPI FUTURI

La definizione di un'architettura lambda permette di bilanciare nella maniera ottimale il trade-off tra bassa latenza e alto throughput per query che lavorano su grandi moli di dati.

L'ambito degli Europei si è rivelato **sfidante**, per via della difficoltà nell'estrarre i dati, ma anche ideale per questo tipo di analisi. Infatti l'analytics di dati sportivi rientra nel caso d'uso di questa architettura dove si vogliono elaborare sia grosse quantità di dati per un'analisi pre-partita che elaborare dati in poco tempo per un rapporto statistico durante la partita.

Inoltre l'ecosistema Python si sta dimostrando sempre più maturo per gestire architetture per l'analytics grazie alla nascita di framework nativi per tale linguaggio.

MongoDB si è rivelata una scelta vincente, sia per le proprietà di scalabilità tipiche dei sistemi NoSQL, ma soprattutto per la suite interna di tool per la **visualizzazione**. Poter realizzare in pochi passi una pipeline per la definizione di grafici, permette di ridurre di molto il divario tra figure tecniche e personale addetto all'analisi dei dati.

L'architettura quindi si è rivelata efficace e negli sviluppi futuri si potrebbe sicuramente cercare di estendere tale architettura per analizzare dati di qualsiasi competizione.

## RIFERIMENTI

- [1] James Warren and Nathan Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.
- [2] Docker. <https://www.docker.com/>.
- [3] Apache Kafka. <https://kafka.apache.org/>.
- [4] Dask: Scalable analytics in Python. <https://dask.org/>.
- [5] Faust: Python stream processing. <https://faust.readthedocs.io/en/latest/>.
- [6] MongoDB. <https://www.mongodb.com/>.
- [7] Api-Football. <https://www.api-football.com/documentation-v3#operation/get-fixtures-statistics/>.
- [8] Historical Euro results dataset. <https://www.kaggle.com/martj42/international-football-results-from-1872-to-2017/>.