RELAZIONE SECONDO PROGETTO BIG DATA – ROMA TRE 2017/2018

Gruppo: Pandamaniacs – Federico Ferri 437567 -Marco Pietrangeli 464998

Sommario

Introduzione

Il topic proposto riguardava la creazione di uno scenario per la sperimentazione di tecnologie di data streaming.

La particolarità di questo tipo di analisi, al contrario di quelle batch solitamente utilizzato nei sistemi BigData, è che esse sono caratterizzate dal processare un dataset "*infinito*" proveniente ad esempio da tweet o sensori.

In questo tipo di sistemi infatti i dati vengono prodotti continuamente e c'è bisogno di analizzarli immediatamente per ottenere risultati a bassa latenza e, se necessario, in Real-Time o Near-Real-Time.

Per questo tipo di analisi ad oggi sono conosciute due principali tipi di architetture molto efficienti: l'architettura Lambda [1] e quella Kappa [2] (o DataFlow) che analizzeremo in un paragrafo dedicato mostrando le motivazioni della nostra scelta [3].

Scenario e Obiettivi

Nel nostro scenario ci siamo messi nei panni di una Software House che sviluppa videogiochi e che necessitasse di un sistema di BigData al fine di analizzare gli interessi dei propri utenti per le progettazioni dei futuri giochi da sviluppare e degli update per quelli già in commercio. Ultimamente infatti molte software house stanno effettuando accordi con i concorrenti per inserire nei propri giochi ambientazioni, oggetti o personaggi particolarmente apprezzati dal pubblico. Le domande che ci si pone a questo punto è come decidere quali siano i giochi più apprezzati? Oppure come scegliere in quali lingue tradurre il nostro gioco? Ed è qui che si sente la necessità di un sistema BigData che analizzi gli interessi e la provenienza geografica dei nostri utenti e ci fornisca un'analisi su cui effettuare le scelte commerciali.

Da dove cominciare? Steam! La piattaforma di distribuzione digitale di giochi per computer più grande al mondo oltre ad essere il principale marketplace su cui vendere il proprio gioco è anche il miglior mezzo per recuperare dati sugli interessi e le abitudini degli utenti.

Da qui il problema che ci si è posto è stato come recuperare tali dati e come analizzarli.

Tecnologie

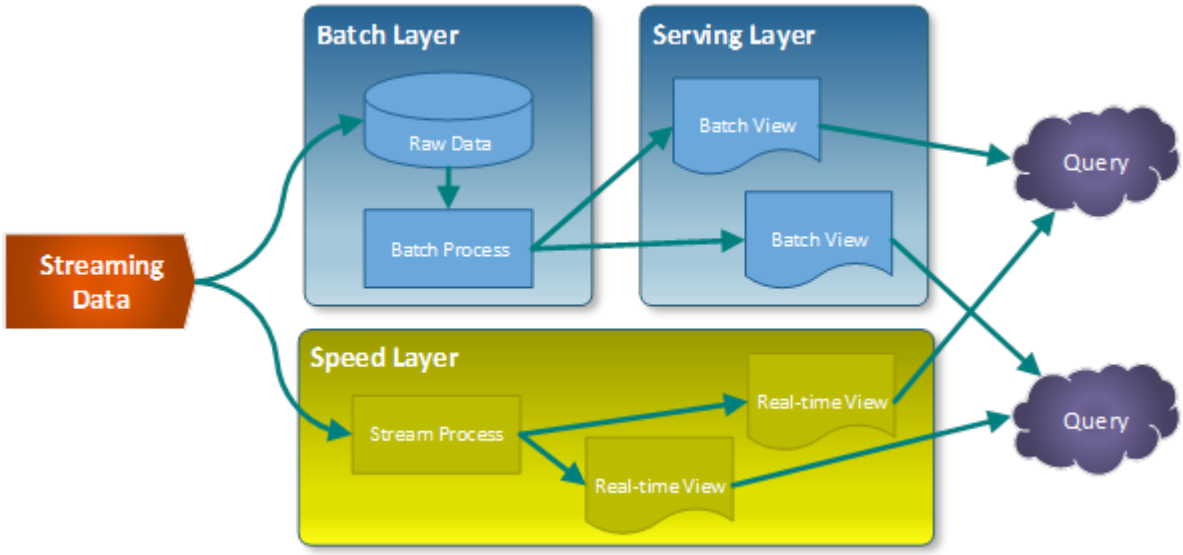
Nel mondo dei BigData esistono una moltitudine tecnologie e soluzioni differenti che possono essere adottate per la risoluzione di un problema. Di seguito analizzeremo quelle da noi individuate come le migliori per il raggiungimento dell'obiettivo del progetto cercando di spiegare sinteticamente le motivazioni che hanno portato a questa scelta.

Architettura Lambda VS Architettura Kappa

In letteratura vengono proposte due diverse architetture per sistemi di BigData Real-Time. L'architettura Lambda e l'architettura Kappa (detta anche DataFlow) [4].

Queste differiscono principalmente per la presenza o meno di un Batch Layer che affianca la computazione Real-Time che rende le due architetture adatte a differenti utilizzi.

Architettura Lambda

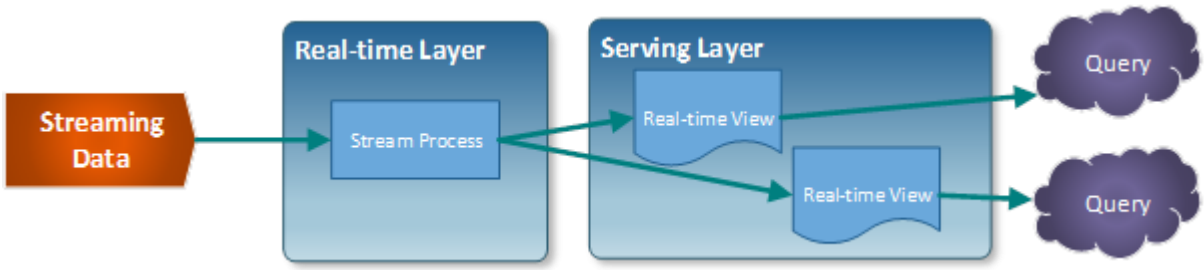


L'architettura Lambda ad oggi è una delle più utilizzate per il processamento dei dati in tempo reale progettata per fornire un sistema a bassa latenza, scalabile e *fault-tolerant* [5].

Questa è composta da un **batch layer** ed uno **speed layer** per la gestione dello streaming di dati ricevuto in input ed un **serving layer** per la risposta alle query degli utenti.

- **Batch Layer:** Memorizza I dati ricevuti in streaming ed esegue periodicamente delle computazioni batch per la creazione di viste che potranno essere aggiornate per correggere eventuali errori derivati da dati ritardatari o errori di computazione. Il batch layer risulta quindi fondamentale per garantire fault-tolerance e maggiore precisione nei nell'elaborazione dei dati.
- **Speed Layer:** Lo speed Layer ha il compito di creare delle viste in tempo reale che andranno a completare quelle già analizzate nel Batch Layer. Lo speed layer ha quindi il compito di ridurre la latenza permettendo eventuali errori dovuti a dati ancora non ricevuti ma elaborando i dati non ancora processati in batch.
- **Serving Layer:** Le viste elaborate dai due Layer precedenti vengono poi elaborate insieme dal serving layer che si occupa di rispondere alle query degli utilizzatori dell'architettura fornendo viste pre-compilate o modellandone di nuove.

Architettura Kappa



L'architettura Kappa al contrario della precedente non presenta uno Layer Batch ma punta ad analizzare esclusivamente lo stream di dati in tempo reale [6].

È quindi presente un Real-Time Layer per il processamento dello stream e un serving layer per le interrogazioni egli utenti.

L'idea di base infatti è quella di gestire l'elaborazione dei dati tramite una rielaborazione continua tramite un unico flusso, per questo è chiamata anche architettura **DataFlow.** Questo prevede quindi che il flusso possa essere riprodotto nella sua interezza o da punti specifici (checkpoint).

I vantaggi di questa architettura sono di snellire il processamento dei dati tramite un unico flusso eliminando la latenza dovuta al processamento Batch e snellisce le query che dovranno cercare i risultati unicamente nelle viste del livello streaming anziché dover unire quest'ultime a quelle del livello Batch.

Motivazioni della scelta architetturale:

Nel nostro caso era necessaria un'architettura in grado di gestire rapidamente i dati per mostrare delle statistiche in tempo reale. Per il tipo di analisi che si deve effettuare è tollerabile qualche errore di precisione ed è preferibile un'architettura più snella in grado di girare anche su un cluster locale o un singolo server per ridurre i costi e poter permettere all'azienda maggiori investimenti in altri campi.

Per queste motivazioni la scelta di un'architettura Lambda non era la soluzione ottimale in quanto il batch layer avrebbe appesantito di molto il sistema garantendo una maggiore fault-tolerance non necessaria e avrebbe richiesto una infrastruttura hardware più costosa che fosse in grado di tenere in storage la moltitudine di dati in ingresso. Invece con l'architettura Kappa ci consente un processo più leggero in grado di tenere in storage una versione aggregata dei dati risparmiando tantissimo spazio in memoria eliminando la necessità di dover scalare il sistema su più macchine o sul cloud ma consentendoci comunque di poterlo fare per migliorare le prestazioni generali.

La scelta per il nostro caso è ricaduta quindi su un'architettura kappa in grado di mostrare statistiche sui giochi attualmente più utilizzati, una distribuzione geografica dei player connessi ed un conteggio delle sessioni di gioco degli utenti.

Dataset: Steam API

Per effettuare queste analisi, sono ovviamente necessary i dati. Per ottenere questi dati abbiamo deciso di far ricorso alle API messe a disposizione dalla piattaforma **Steam** [7]. Rispettando la struttura, di seguito elencata, si potrà effettuare una richiesta WEB di tipo **GET** ai server della piattaforma, i quali risponderanno con un file in uno specifico formato, contenente le informazioni. Un **Pre-Requisito** fondamentale è la generazione di una chiave *univoca* [8] da utilizzare come **identificatore**, strettamente legata ad un account registrato ed autenticato. La richiesta da inviare può essere costruita come un link e commissionata direttamente dal browser, purché rispetti la seguente struttura: [9]

```
http://api.steampowered.com/interface_name/method_name/version/?key&params&format
```

- *interface_name* è il nome d'accesso della specifica libreria che si è interessati ad utilizzare;
- *method_name* è il nome del metodo da invocare;
- *version* la sua versione;
- *key* la chiave a 32 caratteri collegata ad un account, ottenibile tramite specifica richiesta;
- *params* sono i possibili parametri input da aggregare alla richiesta, separati da una *&* (appid='1234'&steamid='0123456789');
- *format* è il formato in cui desideriamo ricevere il file di risposta (JSON, XML, VDF), impostato automaticamente in JSON se non espresso diversamente.

Un elenco di interface_name e method_name, con le relative versioni, è disponibile nell'apposita documentazione [10].

Un esempio di come risulta una richiesta finale da inviare al server è la seguente

```
http://api.steampowered.com/ISteamUser/GetPlayerSummaries/v0002/?key=XXXXXXXXXXXXXXXXXXXXX&steamids=0123456789
```

La risposta sarà un JSON di rimando con le informazioni richieste sull'utente con l'id specificato e la seguente struttura:

JSON ANSWER:


```
{
  "response": {
    "players": [
      {
        "steamid": "76561197960435530",
        "communityvisibilitystate": 3,
        "profilestate": 1,
        "personaname": "Robin",
        "lastlogoff": 1532068695,
        "profileurl": "https://steamcommunity.com/id/robinwalker/",
        "avatar": "https://steamcdn-
a.akamaihd.net/steamcommunity/public/images/avatars/f1/f1dd60a188883caf82d0cbfcc
fe6aba0af1732d4.jpg",
        "avatarmedium": "https://steamcdn-
a.akamaihd.net/steamcommunity/public/images/avatars/f1/f1dd60a188883caf82d0cbfcc
fe6aba0af1732d4_medium.jpg",
        "avatarfull": "https://steamcdn-
a.akamaihd.net/steamcommunity/public/images/avatars/f1/f1dd60a188883caf82d0cbfcc
fe6aba0af1732d4_full.jpg",
        "personastate": 0,
        "realname": "Robin Walker",
        "primaryclanid": "103582791429521412",
        "timecreated": 1063407589,
        "personastateflags": 0,
        "loccountrycode": "US",
        "locstatecode": "WA",
        "loccityid": 3961
      }
    ]
  }
}
```


Streaming Messaging: Kafka

Per processare lo stream di dati ricevuto in input risulta necessario l'utilizzo di Apache Kafka che faccia da intermediario tra lo streamer e la piattaforma di elaborazione [11].

Apache Kafka è una piattaforma software open source per lo streaming sviluppata da Apache Software Foundation, scritta in Scala e Java. Il progetto mira a fornire una piattaforma unificata, ad alto throughput e a bassa latenza per gestire i feed di dati in tempo reale.

Questo ha l'obiettivo di connettersi a vari servizi esterni per l'importazione ed esportazione dei dati creando una coda di messaggi scalabile e fault-tolerant.

L'utilità di Kafka sta proprio nella sua facilità di integrazione tra i vari servizi che rende molto semplice la comunicazione tra lo streamer e il resto dell'architettura, consentendo inoltre ulteriori aggiunte future sia di input che di elaborazione (rendendo per esempio possibile il passaggio ad un'architettura lambda modificando in maniera minore l'intera infrastruttura, o l'aggiunta di ulteriori streamer senza la necessità di modifiche).

Inoltre, aggiunge un ottimo grado di tolleranza agli errori tenendo in memoria i dati nel caso l'architettura dovesse interrompere il proprio funzionamento o dovessero esserci dei ritardi in elaborazione i messaggi inviati dallo streamer non andrebbero persi.

Storage: MongoDB

Come viene spesso riportato in letteratura, la lista di nuove tecnologie e tecniche da imparare è in continua crescita. Tuttavia, è ugualmente impressionante la velocità con cui le tecnologie consolidate vengono rimpiazzate. Apparentemente da un giorno all'altro, tecnologie affermate sono messe in discussione da un repentino cambiamento di attenzione da parte dei programmatori. Il fenomeno è evidente nell'affermazione delle tecnologie **NoSQL** a scapito dei ben consolidati database relazionali. Anche se sembra che queste transizioni avvengano nel corso di una notte, la realtà è che possono passare anni prima che una nuova tecnologia divenga pratica comune. L'entusiasmo iniziale è guidato da un gruppo relativamente piccolo di sviluppatori e aziende. I prodotti migliorano con l'esperienza e, quando ci si rende conto che una tecnologia è destinata a rimanere, altri cominciano a sperimentarla. Ciò è particolarmente vero nel caso *NoSQL* poiché spesso queste soluzioni non vengono progettate come alternative a modelli di storage più tradizionali, ma intendono piuttosto far fronte a nuove necessità. Laddove storicamente i fornitori di database relazionali hanno sempre tentando di posizionare i loro software come soluzione universale per qualunque problema, NoSQL tende a individuare piccole unità di responsabilità per ognuna delle quali scegliere lo strumento ideale. Quindi uno **stack NoSQL** potrebbe contemplare un database relazionale, *MySQL* per esempio, *Redis* per ricerche veloci e *Hadoop* per le elaborazioni dati intensive. In parole povere *NoSQL* è essere aperti e coscienti dell'esistenza di modelli e strumenti alternativi per la gestione dei dati. In questo senso, la grande flessibilità di uno storage NoSQL rispetto ad un tradizionale database relazionale, ha influenzato notevolmente la nostra scelta progettuale di utilizzare una base di dati *document-based* come **MongoDB** [12]. Un ulteriore, fondamentale, motivazione è stata dettata dalla ben nota velocità della lettura di cui **MongoDB** è dotato, che abbiamo convenuto essere un requisito fondamentale per un sistema **Real-Time**, oltre alla semplicità di inserimento dei dati in ricezione, che ci verranno forniti già in formato **JSON** da parte dei server di Steam.

Real-Time Analysis: SparkStreaming

Apache Spark è un framework di cluster-computing open source. Originariamente sviluppato presso l'Università della California, l'AMPLab di Berkeley, il codice di Spark fu successivamente donato alla Apache Software Foundation. Tra i vari framework sviluppati per l'analisi di BigData risulta uno dei più completi, veloci ed affidabili, è composto da vari moduli tra i quali uno appositamente studiato per le computazioni in tempo reale: **SparkStreaming**. Il funzionamento di questo modulo sfrutta l'efficienza dell'elaborazione di Spark per eseguire trasformazioni in RDD di mini-batch. Questa soluzione permette quindi un'ottima elaborazione e un grado di tolleranza agli errori migliore dei concorrenti, infatti SparkStreaming garantisce la stessa fault-tolerance di Spark-core grazie alla replicazione degli RDD e dunque alla possibilità di rieseguire il mini-batch in caso di fallimento. Fornisce un'ottima processazione di stream statefull grazie ai checkpoint e la funzione *updateStateByKey()* ulteriormente migliorata nelle nuove versioni della libreria purtroppo non ancora compatibile con Python. Tramite questa è possibile effettuare un calcolo della durata delle sessioni tramite contatori. Per questi motivi è stato preferito ai concorrenti come *Storm, Flume, Trident Storm, Samza* o *Flink*.

Dynamic Query: SparkSQL

Successivamente all'analisi in tempo reale si rende necessaria la possibilità di ulteriori analisi più mirate. Per questo, utilizzando **Spark** come framework per lo streaming andremo ad utilizzarlo anche per interrogare il sistema in batch. In questo caso ci vengono forniti due moduli per l'analisi, uno basato su approccio Map-Reduce classico e uno in cui l'utente può interagire con il sistema tramite query SQL.

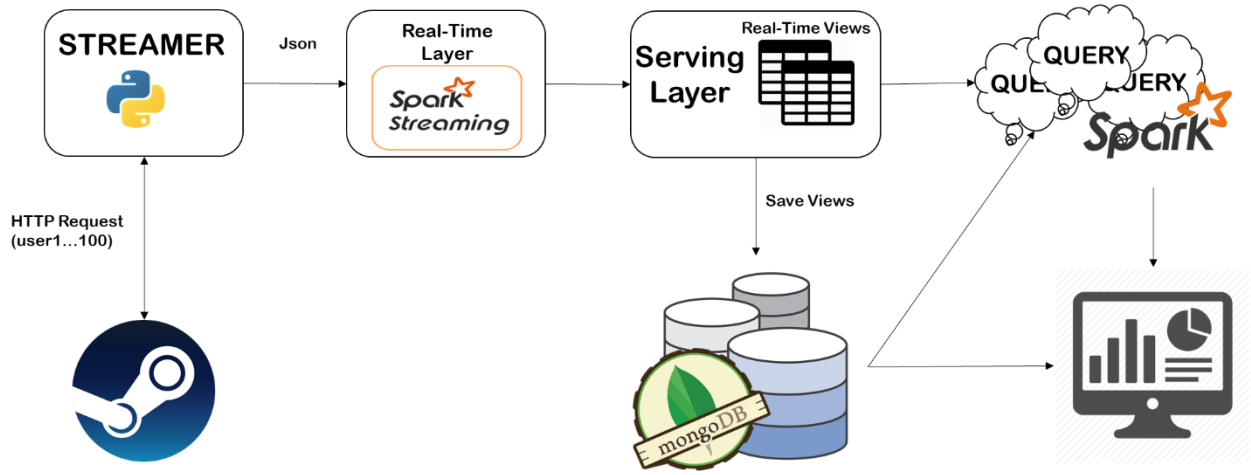
Tra questi due moduli è stato scelto **SparkSQL** che ci consente di effettuare delle query sui dati precedentemente elaborati tramite l'utilizzo di semplici query SQL consentendo una certa flessibilità nelle

interrogazioni possibili. Tramite la possibilità di parametrizzare alcuni variabili è possibile infatti scrivere una query sola nel codice e definire intervalli temporali o item specifici su cui filtrare quando si va ad eseguire la query. Questo modulo sfrutta anziché gli RDD visti in **SparkStreaming** un altro tipo di dati chiamati **DataFrame** questi risultano meno strutturati rispetto agli RDD e non sono tipizzati a tempo di compilazione. Per poter passare da RDD a Dataframe è quindi necessaria una piccola operazione di conversione tramite la quale sarà poi possibile effettuare query dinamiche.

Visualizzazione dei risultati: Apache2, JQuery, MatPlotLib

Per la visualizzazione dei risultati si è scelto di ricorrere ad una piccola applicazione web. **Apache2** [13] è stato utilizzato come infrastruttura di base su cui costruire la *WebApp*. La semplicità di installazione e la diffusione del prodotto e del relativo supporto On-Line sono state considerazioni fondamentali circa la nostra scelta nel suo utilizzo. Stesso dicasi per **jQuery** [14], una libreria basata su JavaScript che fornisce dei pratici strumenti per gestire eventi, visualizzare dati e una notevole raccolta di tool online. Per la generazione di grafici è stato utilizzato **MatPlotLib** [15] una delle più note librerie di *plotting 2D* per Python. Tale libreria si è rivelata particolarmente comoda per la sua capacità di graficare i dati estratti dai DataFrame.

Infrastruttura Finale



L'utilizzo delle varie tecnologie analizzate finora si può riassumere nell'architettura mostrata nell'immagine sovrastante.

Come possiamo notare avremo un'architettura lambda composta da uno **streamer python** che dialoga con i server di **Steam** tramite richieste **http.** Il file **json** ottenuto come risposta viene poi inserito nella coda di **Kafka** dalla quale viene estratto dal Real-Time Layer sviluppato tramite **Spark Streaming** dove vengono generate il conteggio della durata delle sessioni di gioco degli utenti insieme alla classifica dei giochi più utilizzati e la distribuzione geografica dei giocatori connessi in tempo reale. Tali dati vengono poi resi persistenti tramite **MongoDB**. Infine, tramite **SparkSQL,** è possibile effettuare delle query per ottenere statistiche sull'utilizzo dei giochi, correlazioni di gradimento e distribuzione geografica dei player all'interno di archi temporali scelti dall'utente, permettendo quindi anche la possibilità di effettuare analisi mensili, settimanali o giornaliere.

Di seguito analizzeremo con maggiore accuratezza il funzionamento delle singole componenti dell'infrastruttura nell'ordine definito dal **DataFlow**.

Streamer

Prima di addentrarsi nel funzionamento dello Streamer, ovvero quella parte del software che si occupa di creare il dataStream da analizzare, è necessario prendere in considerazione i prerequisiti necessari all'acquisizione dei dati tramite le API di Steam di cui abbiamo parlato nel capitolo delle tecnologie.

Prerequisiti

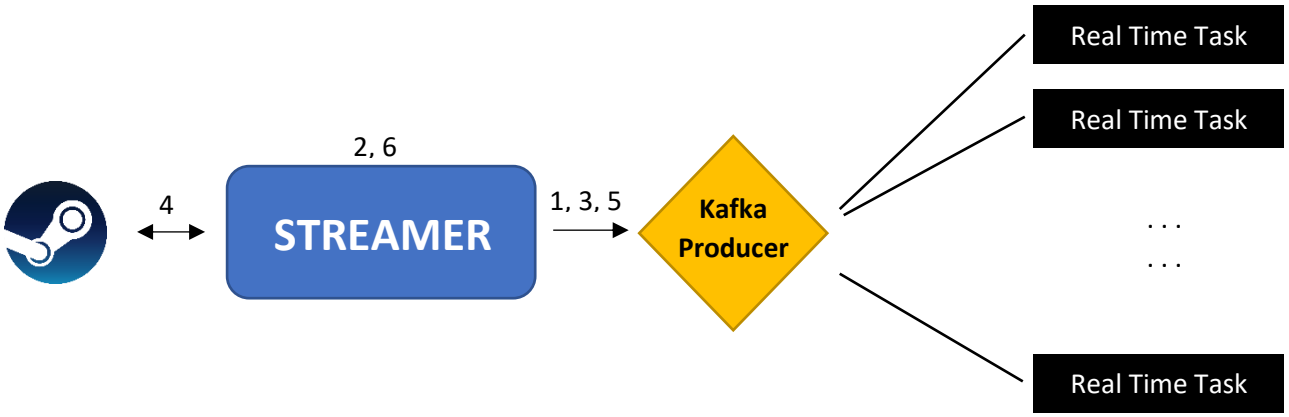
Una volta ottenuta l'API Key per poter effettuare le richieste si è studiata la metodologia per acquisire un Dataset funzionale al nostro scenario. Infatti, essendo il nostro obiettivo quello di analizzare le preferenze e la collocazione geografica degli utenti della nostra ipotetica società, era necessario trovare un pool di utenti adatti allo scopo. Dopo aver attentamente analizzato i metodi di accesso e di richiesta delle WEB API di Steam [9] ,è stata scelta una metodologia di accesso di **Neighbour Discovery**, abbinandovi una ricerca su grafo in ampiezza grazie ad un *Crawler*. La motivazione che ha mosso principalmente questa scelta è stata la necessità, dato il limite giornaliero di richieste (100'000 al giorno), di minimizzare il numero di volte in cui richiedere il servizio, saturandole con il numero massimo di **id utente** per massimizzare il **Throughput** di informazioni a carico del server di destinazione. È stato pertanto realizzato un crawler, in linguaggio **Python**, fornito di un id di partenza. Il crawler costruisce la richiesta richiedendo al server le informazioni su quell'utente, marcandolo come *"visitato"* e mettendo l'intera *"lista amici"* in una *pending-list*. Ogni volta che un nuovo utente viene analizzato viene marcato a sua volta come *visitato* e aggiunto ad una lista accettante nel caso rispetti determinati requisiti (ad esempio un profilo pubblico). La *pending-list* viene riempita mano a mano con la *lista amici* degli utenti. Il risultato finale è un dataset composto da una lista di *id utente*, che verrà ciclicamente acceduta per costruire le successive richieste da effettuare in fase di analisi.

Realizzazione dello Streamer

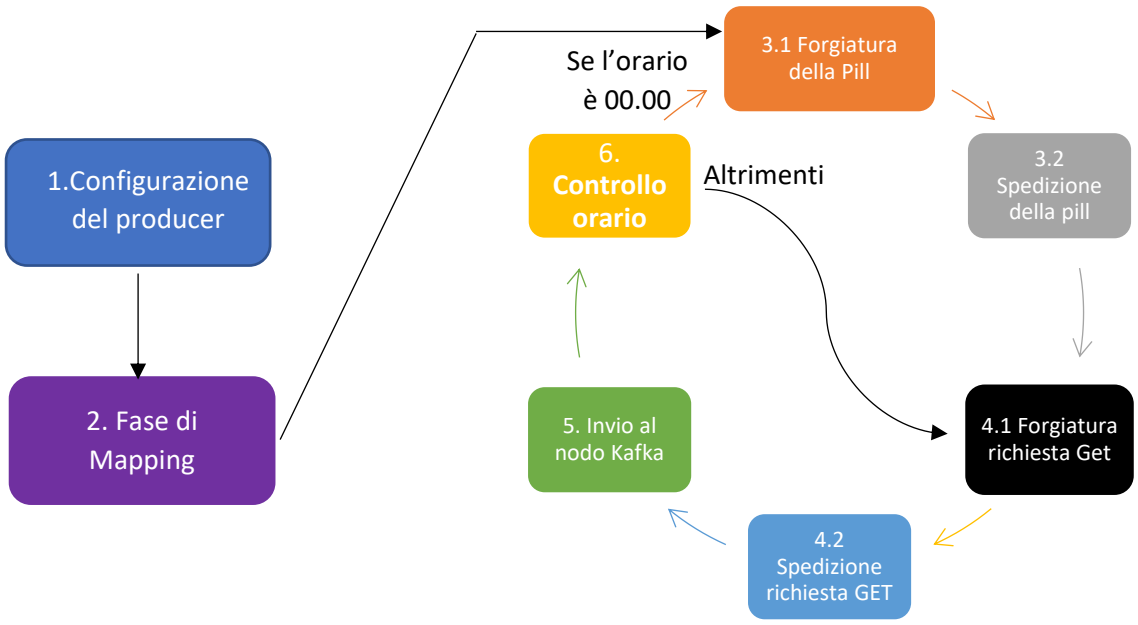
Nell'architettura e nella fase implementativa, la realizzazione dello **Streamer** ed il suo corretto funzionamento acquisisce un ruolo centrale nella richiesta e nell'inoltro dei dati al nodo di analisi **Real-Time** su cui sarà installato **Spark Streaming**. L'amministrazione delle richieste lo mette in una posizione intermedia tra i server di Steam ed il nostro livello di elaborazione, assumendo un comportamento simile a quello di un **Broker**. I suoi compiti sono, principalmente, i seguenti:

1. **Configurazione** di un Producer-Kafka
2. **Mappatura** id_richiesta-lista_utenti
3. **Forgiatura** e **Spedizione** di una **pill** contenente un **reset_code** per l'inizializzazione
4. **Forgiatura** e **Spedizione** della richiesta ai server di Steam
5. **Invio temporizzato** al nodo Kafka
6. **Controllo** sull'orario per l'invio di una **Reset_Pill** a 00.00 per il reset della cache e segnalare l'inizio di una nuova transazione

Lo schema architetturale può essere intuito come segue:



Per quanto riguarda invece il ciclo dei compiti sopraelencato che lo Streamer ha la responsabilità di eseguire, possiamo identificare la sequenza dei lavori da eseguire nel ciclo seguente:



Durante la prima iterazione, lo Streamer si occupa di forgiare un pacchetto, riconoscibile dal sistema come una **pill** tramite un **reset_code** specifico presente nei dati. In questo modo segnaliamo ai consumer l'inizio di una nuova transazione, che nel nostro caso d'uso sono associate alle singole giornate. Subito dopo aver mandato la prima pill, lo Streamer imposterà a False il flag d'inizializzazione, iniziando a forgiare le richieste da inoltrare ai server di **Steam**, accodando una serie di **id utente** in una stringa in maniera ciclica. Operando su un insieme limitato di **dati reali**, il ciclo comincerà nuovamente solo dopo aver inoltrato l'ultima richiesta presente nella mappa costruita in fase di lancio, aggiornando i dati precedentemente analizzati con le nuove informazioni. Una volta raggiunta la mezzanotte, il sistema forgerà nuovamente una pill di **reset** per avvertire il sistema che i dati che si stanno per ricevere fanno parte di una nuova transazione e di **non** sovrascrivere/aggiornare quelli ricevuti prima dell'arrivo della pill stessa.

Analisi Real-Time

L'obiettivo principale del progetto era riuscire a fare delle analisi in tempo reale. Nello specifico sono stati scelti e implementati tre job ognuno contenuto in un diverso file **Python**: *"getUserSessions.py", "getLocation.py" e "getBestGames.py"*. Come si può intuire dal nome i tre Job consistono nel calcolare la durata giornaliera delle sessioni di gioco degli utenti, ottenere una visualizzazione in tempo reale della provenienza geografica dei giocatori ed una classifica aggiornata dei giochi più utilizzati.

Per l'analisi dei dati in tempo reale è stato scelto il framework **Spark Streaming** ed è stato implementato tramite il linguaggio **Python** seguendo il paradigma **MapReduce**.

Job 1 – getUserSessions:

Il primo Job che andremo ad analizzare è quello che tratta il calcolo della durata delle sessioni dei giocatori. Analizzando i dati ricevuti dallo **Streamer** possiamo notare che il **json** che otteniamo in input non ci fornisce informazioni certe sulla durata della sessione ma unicamente se l'utente è connesso o meno e quale gioco sta giocando. Sapendo però che lo stream è temporizzato con intervalli di breve durata è possibile supporre che se un utente era connesso nell'istante t se nell'istante t+1 risulta ancora connesso questo abbia continuato a giocare nel tempo intercorso tra la ricezione del primo **json** al secondo. È quindi possibile contare questi intervalli tramite una variabile di accumulazione per ottenere un valore abbastanza preciso che rappresenti il tempo di connessione dell'utente. Nel nostro caso quindi sarà necessario effettuare il seguente calcolo: sapendo che l'analisi viene svolta su un campione di 3000 utenti e che le API consentono la richiesta di massimo 100 utenti alla volta il dato del nostro utente viene aggiornato una volta ogni 30 **json** ricevuti, dal momento che effettuiamo una richiesta http ogni 2 secondi, otterremo che ogni intervallo tra un aggiornamento e l'altro relativo ad un utente è di 1 minuto. Da questo possiamo quindi affermare di avere un margine di errore di 1 minuto a sessione, risultato decisamente accettabile per l'analisi da effettuare.

Data quindi questa premessa andremo a creare all'interno di **Spark** uno **StreamContext** connesso a **kafka** per ricevere i dati dello **Streamer** e procedere all'elaborazione. Ricevuto il **json** andrà caricato in un **RDD** per poter essere mappato.

```
kvs.map(lambda v: json.loads(v[1])) \
      .flatMap(lambda player: (player['response']['players'])) \
      .map(lambda x: (x['steamid'],clean(x)))
```

In queste righe è mostrato il codice per ottenere una mappa del **json** dopo il parsing. *Kvs* indica lo Stream di dati generato da **Kafka** mentre la funzione *clean* si occupa di eliminare dal RDD quei dati non utili al nostro scopo come i link delle immagini del profilo dei giocatori e simili.

Ottenuta quindi una mappa avente come chiave l'id del giocatore e come valore i dati relativi, è possibile iniziare il lavoro di analisi dei dati. Per poterlo fare è necessario ricorrere ad una delle funzioni di programmazione *stateful* di **Spark Streaming**, *updateStateByKey*. Tale funzione consente di mantenere uno stato arbitrario aggiornandolo continuamente con nuove informazioni. Per usare questo, sono necessari due passaggi: Definire lo stato iniziale, e definire la funzione di aggiornamento. In ogni batch, **Spark** applicherà la funzione di aggiornamento dello stato per tutte le chiavi esistenti, indipendentemente dal fatto che abbiano o meno nuovi dati in un batch. Se la funzione di aggiornamento restituisce *None*, la coppia chiave-valore sarà eliminata. Nel nostro caso quindi tale funzione prevede uno stato iniziale in cui gli utenti sono offline, e una funzione di aggiornamento che vada ad aggiornare lo stato dell'utente (il campo *PersonaState*: 0 Offline – altrimenti Online) e verifichi se la connessione è stata mantenuta. In quest'ultimo caso la funzione dovrà andare ad incrementare di 1 il conteggio della sessione dell'utente se il gioco a cui sta giocando è lo stesso a cui giocava prima, altrimenti aprire una nuova sessione per un gioco differente. Effettuata questa operazione

quindi il risultato sarà una mappa con chiave utente e valore una lista contenente le informazioni relative alle varie sessioni dell'utente in quella giornata. Giunti a questo ulteriori trasformazioni sono necessarie per salvare la mappa in **MongoDB** come un unico documento **json** in maniera da garantire il salvataggio dei dati in un'unica transazione come spiegheremo di seguito.

```
RDD.updateStateByKey(updatePlayerCount) \
        .map(lambda x:{'user':x[0],'sessions':x[1]}) \
        .map(lambda x: (1, [x])) \
        .reduceByKey(lambda x, y: x+y) \
        .map(lambda x:{'users':x[1]}) \
        .foreachRDD(saveToMongo)
```

A questo punto però sorge il problema dell'utilizzo di una struttura *stateful*. Infatti, i dati vengono continuamente aggiornati e il contatore viene incrementato indistintamente. Poiché il nostro obiettivo era quello di raccogliere i conteggi giornalieri, sarebbe stato necessario inserire un ulteriore parametro nella chiave relativo alla data. Questa soluzione però porta al tenere nell'RDD le informazioni dei giorni precedenti rallentando il sistema. La funzione *updateStateByKey* verrebbe comunque applicata a tutte le chiavi anche senza avere ulteriori dati per aggiornarle. La soluzione da noi trovata prevede quindi di salvare un **JSON** contenente i dati relativi alle sessioni relative ad una sola data su **MongoDB** in maniera tale da snellire **l'RDD** tenendo in memoria solamente i dati del giorno stesso. Per ottenere questo risultato è stato necessario modificare la funzione di update in maniera tale che dallo **Streamer** fosse possibile ricevere un messaggio terminatore chiamato "*Pill*" che possa essere interpretato da ***Spark*** come un reset di tutte le connessioni. Tale messaggio viene riconosciuto dal fatto che gli utenti sono segnati con il campo *PersonaState* = 999.

I dati ottenuti da questo Job verranno poi caricati da **MongoDB** ed analizzati tramite **SparkSQL** successivamente.

Job 2 – getLocation

Il secondo job tratta di effettuare un'analisi geografica in tempo reale della provenienza dei vari player connessi in quell'istante. In questo caso le operazioni iniziali per il ricevimento dello stream saranno identiche a quelle del primo Job. Anche in questo caso riceveremo gli aggiornamenti di un utente ogni minuto mantenendo lo stesso margine di errore del precedente task. In questo caso la programmazione *stateful* è necessaria per mantenere le informazioni di tutti i 3000 utenti poiché altrimenti visualizzeremo solamente i 100 che vengono ricevuti ogni richiesta. In questo caso quindi la funzione di update non andrà a incrementare un contatore ma unicamente ad aggiornare lo stato dell'utente da online a offline e viceversa.

```
def updatePlayerCount(currentCount, countState):

  if not currentCount:
    return countState
  else:
    x = currentCount[-1]

    if 'loccountrycode' in x:
      loccountrycode = x['loccountrycode']
    else:
      loccountrycode = "Unknown"
    if 'locstatecode' in x:
      locstatecode = x['locstatecode']
    else:
      locstatecode = "Unknown"
    if 'loccityid' in x:
      loccityid = x['loccityid']
    else:
      loccityid = "Unknown"
    if 'personastate' in x:
      personastate = x['personastate']
    else:
      personastate = "0"

    countState = (loccountrycode,locstatecode,loccityid,personastate)

    return countState
```

Dopo questa fase tramite un filtro sugli utenti online ed il paradigma **MapReduce** si potranno contare il numero di giocatori connessi per ciascun luogo. Abbiamo tre livelli di dettaglio per il luogo: loccountrycode, locstatecode e loccityid per comodità in questa fase viene calcolata solamente il livello di dettaglio più alto e più basso (country e city), lasciando il raggruppamento per singoli stati (utile prevalentemente per i giocatori USA) ad analisi successive.

Anche in questo caso dopo il reduce viene effettuata un ulteriore trasformazione per poter salvare i risultati su mongo con un'unica transazione.

```
filterx = kvs.map(lambda v: json.loads(v[1])) \
             .flatMap(lambda player: (player['response']['players'])) \
             .map(lambda x: (x['steamid'],x))\
             .updateStateByKey(updatePlayerCount)\
             .map(lambda x: filterOnline(x[1]))\
             .reduceByKey(lambda a, b: a + b) \
             .map(lambda x:{'country':x[0][0], 'connections':x[1], 'details':
                                          {'state':x[0][1], 'city':x[0][2]}})

countries = filter.map(lambda x:(x['country'], x['connections'])) \
                  .reduceByKey(lambda x,y: x+y) \
                  .map(lambda x: {'country':x[0], 'connections':x[1]}) \
                  .map(lambda x: (1, [x])) \
                  .reduceByKey(lambda x,y: x+y) \
                  .map(lambda x: {'locations':x[1]})

detailed = filter.map(lambda x: (1, [x])) \
                 .reduceByKey(lambda x,y: x+y) \
                 .map(lambda x: {'cities':x[1]})

countries.foreachRDD(saveCountriesToMongo)
detailed.foreachRDD(saveDetailedToMongo)
```

Job 3 - getBestGames:

Il terzo Job invece ha il compito di stilare una classifica in tempo reale dei giochi più utilizzati. Anche in questo caso la programmazione sarà necessariamente *Statefull* per mantenere i dati di tutti i 3000 utenti.

Dopo aver caricato il json questo job la funzione di update andrà ad effettuare un mapping da tenere aggiornato tra gli utenti e il gioco a cui stanno giocando o NoGame se non sono online oppure nel caso siano comunque connessi ma non stiano giocando a nessun gioco.

```
def updateGamesCount(currentCount, countState):

    if not currentCount:
        return countState
    else:
        x = currentCount[-1]

        if 'gameid' in x:
            gameid = x['gameid']
        else:
            gameid = "No Game"

        countState = gameid

        return countState
```

Da qui è necessaria un'operazione di MapReduce simile a quella vista negli esempi sul wordCount. Viene creata quindi una mappa tra i valori della mappa precedente e un "1" per poter effettuare il conteggio. Eseguendo quindi una reduceByKey andremo ad aggregare sull'id del gioco e sommeremo i valori "1" ottenendo il numero di giocatori per ogni gioco. Per ottenere poi una classifica i giochi vengono ordinati per numero di giocatori e convertiti per il salvataggio tramite transazione unica su MongoDB.

```
kvs.map(lambda v: json.loads(v[1])) \
    .flatMap(lambda player: (player['response']['players'])) \
    .map(lambda x: (x['steamid'],x))\
    .updateStateByKey(updateGamesCount)\
    .map(lambda x: (x[1],1))\
    .reduceByKey(lambda a, b: a + b)\
    .transform(lambda rdd: rdd.sortBy(lambda x: x[1], ascending=False))\
    .map(lambda x: {'appid':x[0], 'count':x[1]})\
    .map(lambda x: (1, [x])) \
    .reduceByKey(lambda x, y: x+y) \
    .map(lambda x:{'games':x[1]}) \
    .foreachRDD(saveToMongo)
```

Analisi Batch

Dal primo job Real-Time salviamo su MongoDB un json per ogni giorno contenente tutte le informazioni degli utenti e le loro sessioni relative a quella data. Tali dati ci consentono di effettuare delle ulteriori analisi di maggior interesse e maggior livello di dettaglio.

Per l'esecuzione di queste analisi si è scelto di continuare ad utilizzare lo stesso framework già utilizzato nello streaming sfruttando il suo modulo SparkSQL per effettuare delle query dinamiche per poter effettuare interrogazioni su diversi intervalli temporali.

In prima analisi è stato necessario convertire i Json provenienti dallo storage in DataFrame Spark per poter essere analizzati. Per questo passaggio è risultato comoda una conversione intermedia dal file Json a uno CSV, tramite questa conversione la creazione del dataframe risulta più rapida e meno soggetta ad errori di parsing.

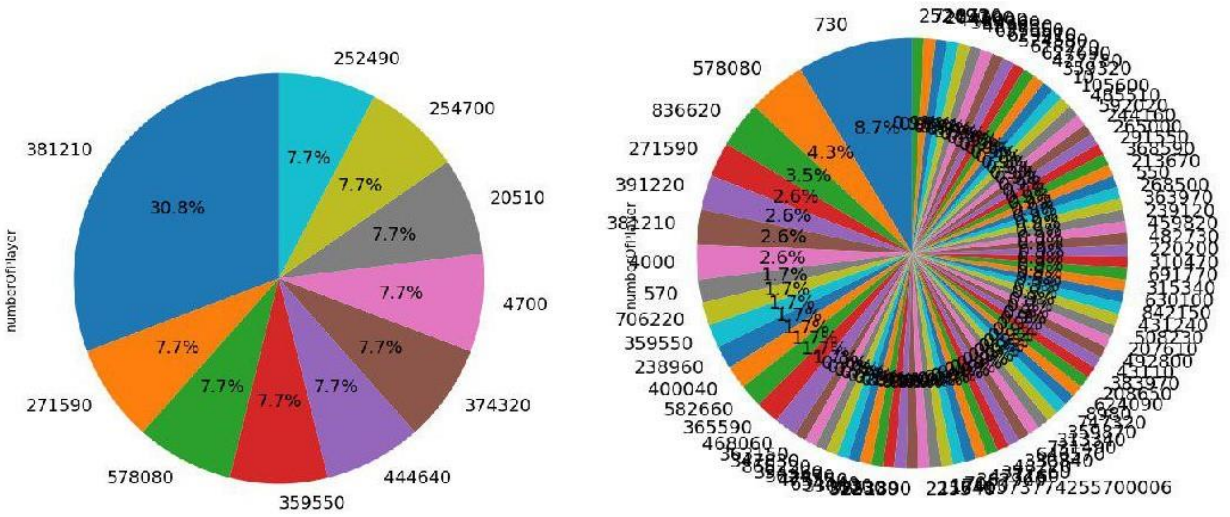
Per effettuare le query è necessario specificare un intervallo di date su cui si vuol fare l'interrogazione, *startDate* ed *endDate*.

```
query1 =

"SELECT gameid, COUNT(DISTINCT(user)) as numberOfPlayer, AVG(connectionTime) as AverageTime,
STDDEV(connectionTime) as STDDEV

FROM t1

WHERE sessionDate>=\"{0}\" AND sessionDate<=\"{1}\"

GROUP BY gameid

ORDER BY numberOfPlayer DESC"

.format(startDate,endDate)
```

La prima query si occupa di effettuare il conteggio dei singoli giocatori che hanno giocato a ciascun gioco indicando il tempo medio di connessione e la deviazione standard di quest'ultima. È stato scelto di mostrare questo ulteriore parametro perché abbiamo ritenuto indicativo avere un indice di dispersione statistico necessario per osservare di quanto i valori si discostino dalla media. Lo scarto quadratico medio è infatti uno dei modi per esprimere la dispersione dei dati intorno ad un indice di posizione.

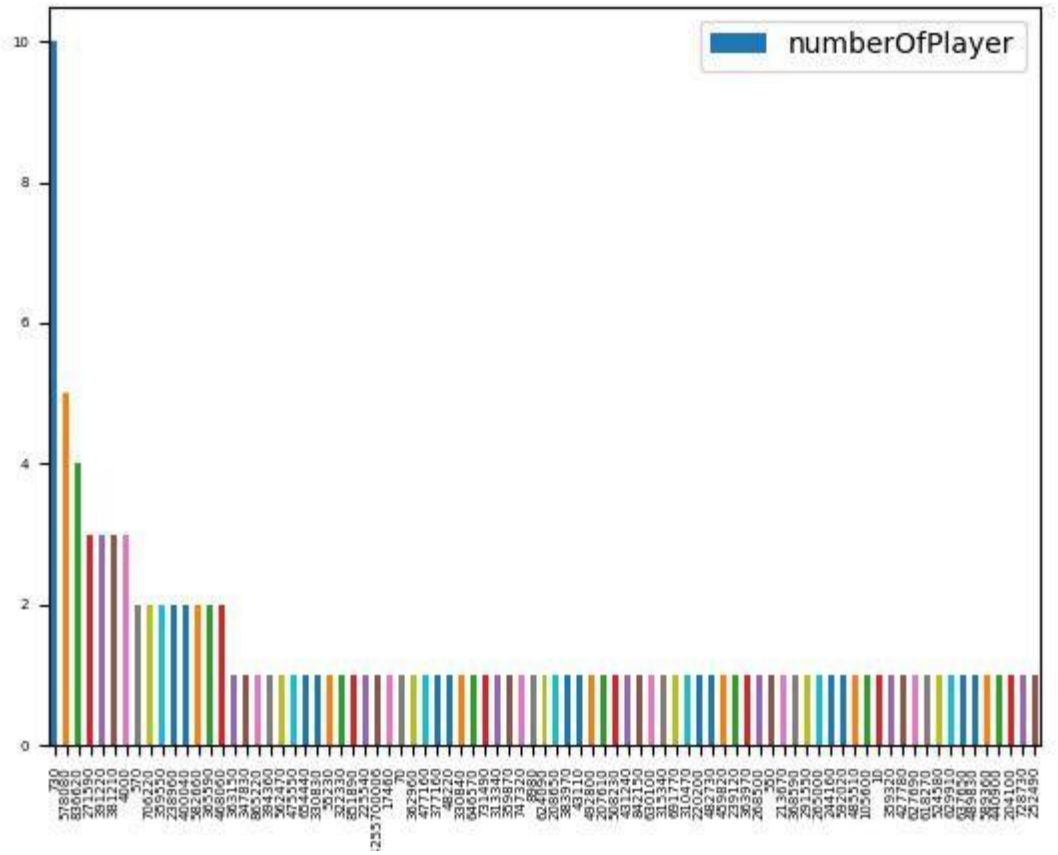
Da questa statistica è possibile risalire ai giochi più apprezzati e trarre conclusioni circa il successo dei nostri giochi o di quelli dei nostri competitor. È inoltre utile per vedere come reagisce l'utenza rispetto a determinate scelte commerciali dettate dai dati delle successive analisi.

I risultati ottenuti erano quindi da rappresentare in una maniera comoda alla visualizzazione. Abbiamo quindi deciso di mostrarli tramite dei grafici riassuntivi generati tramite la libreria Python MatPlotLib.



Inizialmente avendo effettuato i test con pochi dati avevamo scelto come rappresentazione dei giochi più usati un grafico a torta che mostrasse chiaramente il *Market Share* dei giochi. Purtroppo, come è possibile notare dal grafico di destra, con un alto numero di dati questo risulterà molto confusionario.

Per risolvere il problema siamo quindi passati ad un più comune grafico a barre che mostra in maniera chiara i dati anche per i valori minori.

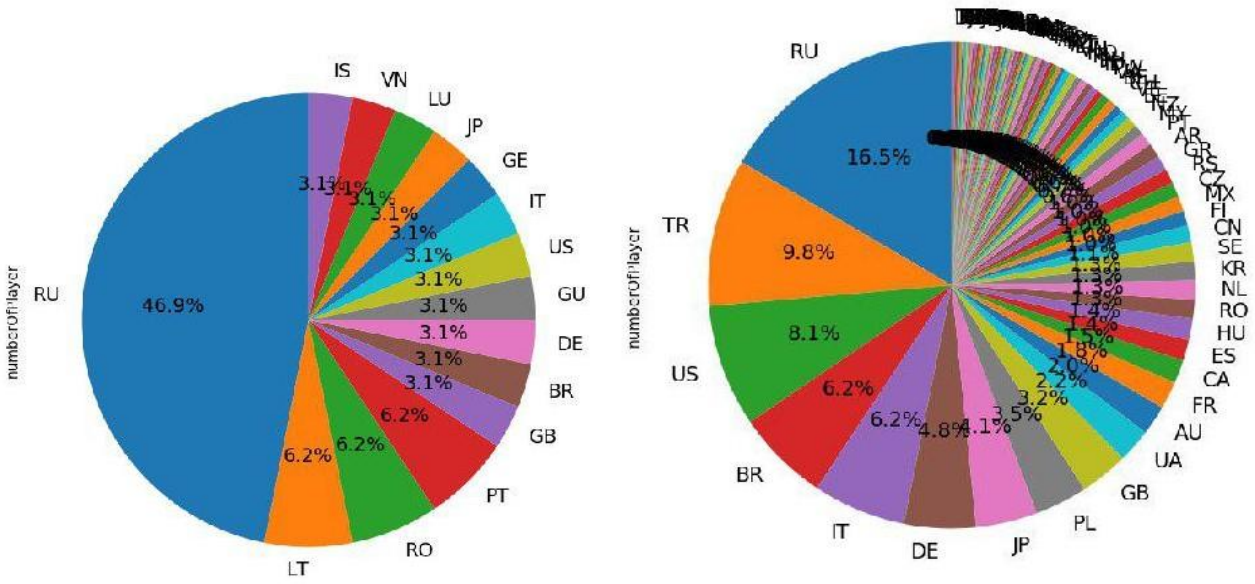


```
query2 =

"SELECT loccountrycode, locstatecode, loccityid, COUNT(DISTINCT(user)) as numberOfPlayer,
AVG(connectionTime) as AverageTime, STDDEV(connectionTime) as STDDEV

FROM t1

WHERE sessionDate>=\"{0}\" AND sessionDate<=\"{1}\" AND connectionTime>0

GROUP BY loccountrycode, locstatecode, loccityid

ORDER BY numberOfPlayer DESC"

.format(startDate,endDate)
```

```
query3 =

"SELECT loccountrycode, locstatecode, COUNT(DISTINCT(user)) as numberOfPlayer, AVG(connectionTime)
as AverageTime, STDDEV(connectionTime) as STDDEV

FROM t1

WHERE sessionDate>=\"{0}\" AND sessionDate<=\"{1}\" AND connectionTime>0

GROUP BY loccountrycode, locstatecode

ORDER BY numberOfPlayer DESC"

.format(startDate,endDate)
```

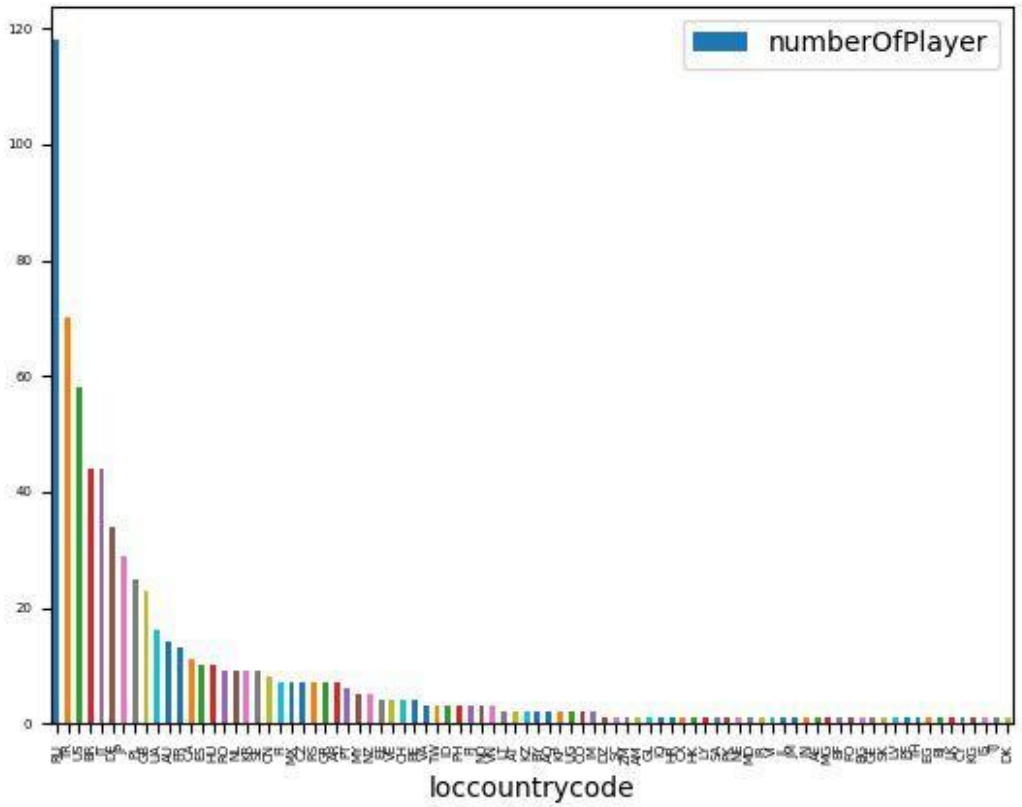
```
query4 =

"SELECT loccountrycode, COUNT(DISTINCT(user)) as numberOfPlayer, AVG(connectionTime) as
AverageTime, STDDEV(connectionTime) as STDDEV

FROM t1

WHERE sessionDate>=\"{0}\" AND sessionDate<=\"{1}\" AND connectionTime>0

GROUP BY loccountrycode

ORDER BY numberOfPlayer DESC"

.format(startDate,endDate)
```

Le query 2-3-4 sono simili tra loro quindi le analizzeremo insieme, queste effettuano un conteggio geografico dei singoli utenti connessi nell'intervallo di tempo specificato rispettivamente a livello di città, stato e nazione. Conoscendo quindi la disposizione geografica dei nostri utenti potremmo decidere di incrementare la pubblicità in aree specifiche, organizzare eventi in aree con il maggior numero possibile di utenti per avere maggior risonanza, o anche inserire nei giochi elementi delle culture di cui è composta principalmente la nostra utenza.



Anche in questo caso in questo caso si era scelta un grafico a torta che è risultato decisamente poco visibile nei test con molti dati.

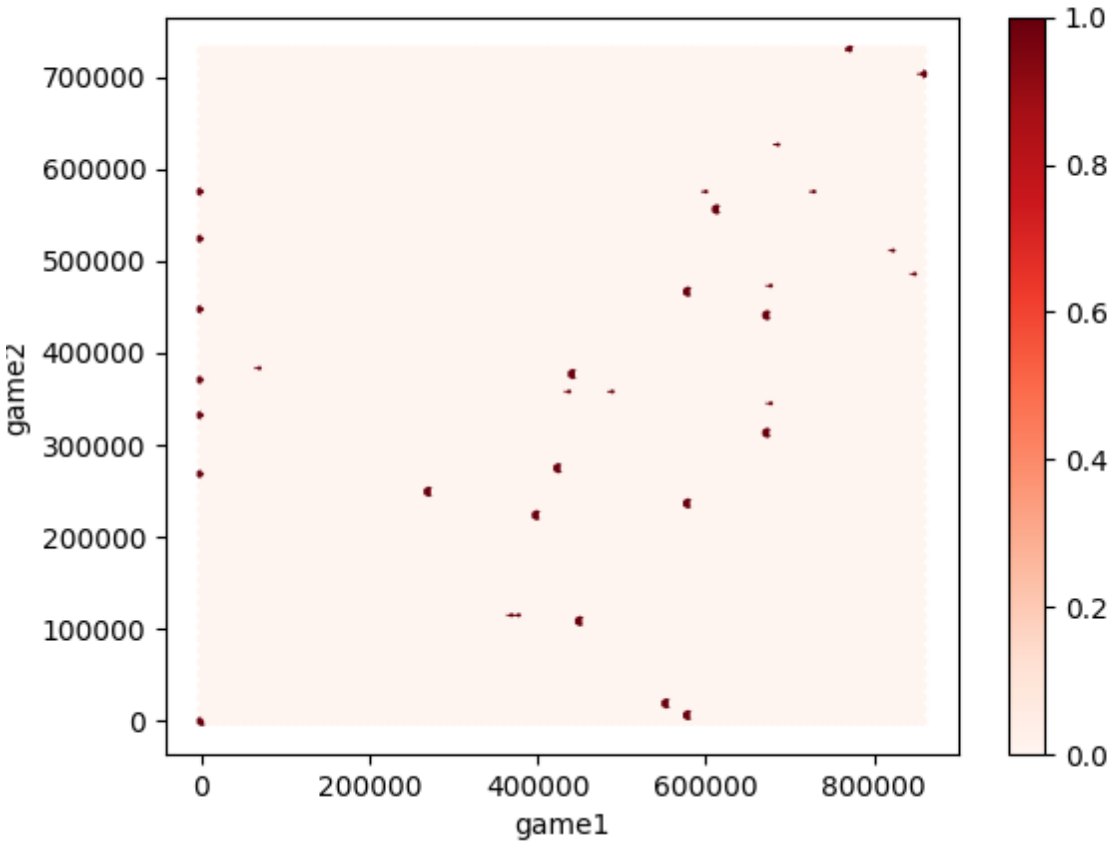
Anche in questo caso il grafico a barre risulta una soluzione che concede una miglior visualizzazione delle informazioni.



```
query5 =

"SELECT tab1.user, tab1.gameid as game1, tab2.gameid as game2

FROM t1 as tab1, t1 as tab2

WHERE tab1.gameid<>'None' AND tab2.gameid<>'None' AND tab1.sessionDate>=\"{0}\" AND
tab1.sessionDate<=\"{1}\" AND tab1.connectionTime>0 AND tab2.sessionDate>=\"{2}\" AND
tab2.sessionDate<=\"{3}\" AND tab2.connectionTime>0 AND tab1.user=tab2.user AND
tab1.gameid>tab2.gameid"

.format(startDate,endDate,startDate,endDate)

query6 =

"SELECT game1, game2, COUNT(DISTINCT(user))

FROM t2 WHERE game1<>'None' AND game2<>'None'

GROUP BY game1, game2

ORDER BY game1"
```

La query 5 ha il compito di creare una vista necessaria al funzionamento della query 6 che serve ad individuare, all'interno di un range di date richieste, i giochi maggiormente correlati tra loro. Ovvero quei giochi che se uno dei due piace ad un utente probabilmente apprezzerà anche l'altro. Nell'ambito del machine learning vi sono molti studi al riguardo, nel nostro caso ne abbiamo tratto una semplificazione che desse comunque un ottima visione d'insieme. Nel nostro caso quindi andremo a vedere le coppie di giochi usati dallo stesso utente (query 5) per poi andare a raggruppare sulle coppie di giochi contando il numero di player che hanno giocato entrambi. Da questa analisi risulta evidente come alcuni giochi siano correlati e con questa informazione è possibile pianificare partnership tra le software house, bundle di vendita dei giochi, buoni sconto per gli utenti che hanno uno dei due giochi della coppia per acquistare l'altro ecc...

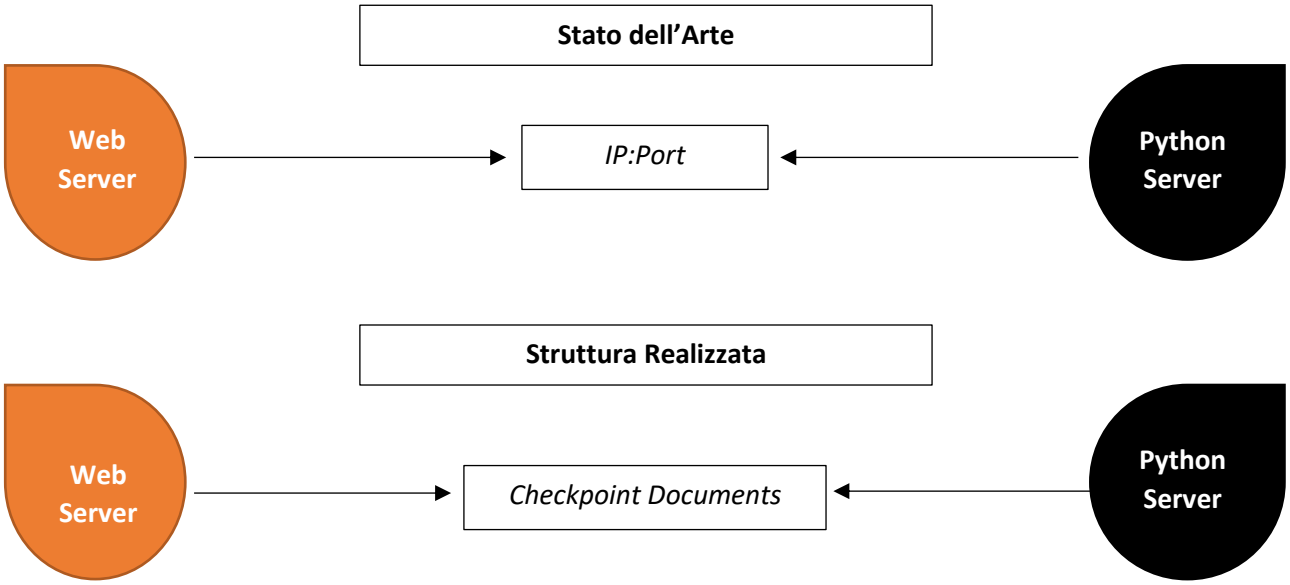
I risultati di questa query vengono poi rappresentati tramite il grafico sottostante ottenuto, come i precedenti, tramite l'utilizzo di matplotlib.



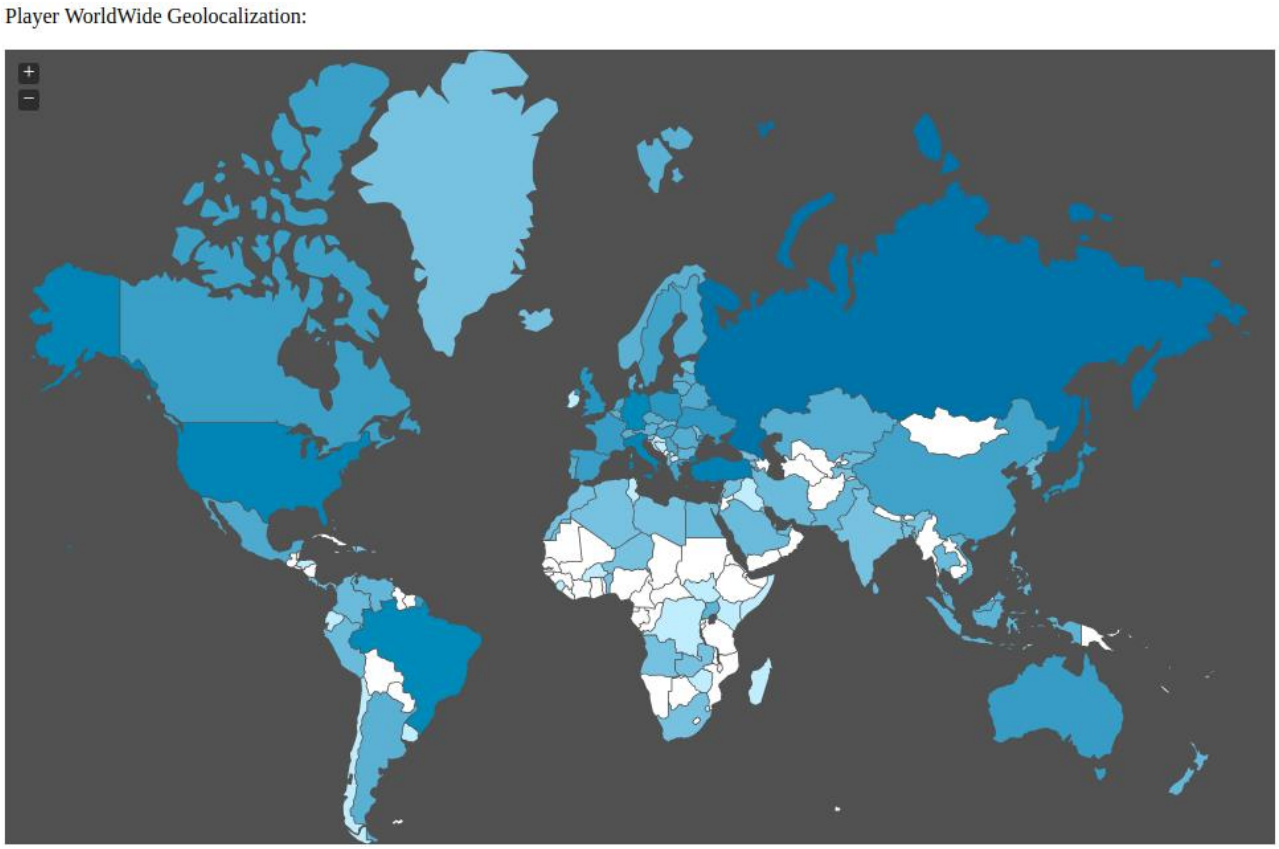
Questi grafici insieme ad una stampa in tabella di questi risultati vengono poi mostrati all'utente tramite una webApp appositamente implementata.

Visualizzazione dei risultati: WebApp

È stata infine realizzata una WebApp per la visualizzazione dei dati ottenuti. La progettazione iniziale prevede di proiettare sul browser i risultati ottenuti, e memorizzati in **MondoDB**, grazie a delle elaborazioni in **JavaScript**. Allo stato dell'Arte, il progetto architetturale dell'applicazione web è composto da un **Python Server** responsabile di effettuare una serie di **Query**, interrogando MongoDB per reperire tutti i documenti di una certa collezione che rispettino determinati dati in ingresso, come ad esempio un intervallo di date, o tutte le elaborazioni di una singola, specifica giornata. Una volta ottenuta la risposta da MongoDB, il Python Server metterebbe a disposizione il **JSON** ottenuto in una determinata coppia *Indirizzo_ip:Porta*, sulla quale farà richiesta il Web Server tramite uno script Javascript, elaborandone i dati proiettandoli sul browser tramite *framework* come **jQuery**. L'implementazione realizzata è tuttavia leggermente diversa: Anziché inviare i dati richiesti dal server Python direttamente su una coppia *indirizzo_ip:Porta*, questi vengono salvati in dei *checkpoint,* ovvero dei documenti temporanei che verranno poi acceduti da Javascript, ottenendo ugualmente le stesse informazioni. La motivazione principale del perché è stata scelta questa strada è dettata unicamente da una questione di **tempistica**. Sebbene la fase iniziale di sviluppo abbia fortemente virato verso un'implementazione **senza** *stadi intermedi* (ovvero i *checkpoint*), i diversi problemi di comunicazione tra le due tecnologie, infatti, si sono rilevati essere più **ostici** del previsto. Per questo motivo si è deciso di utilizzare questo *workaround* temporaneo. I due schemi seguenti sintetizzano la principale differenza tra la scelta architetturale iniziale e quella implementativa:



Una volta stabilita la connessione tra i due nodi, si è fatto uso di alcune funzioni Javascript per mostrare, ad esempio, la distribuzione geografica dei giocatori **online**, colorando i diversi stati del mondo con un'intensità diversa a seconda del numero di giocatori **attualmente online** provenienti da quel paese, come in figura:



La mappa precedente è stata realizzata tramite framework **jQuery**, è stata resa *interattiva*, mostrando il numero effettivo di giocatori attualmente connessi, solo passando il cursore sullo stato a cui si è interessati. La mappa cambia automaticamente l'intensità dei colori all'aumentare/diminuire delle connessioni

provenienti dallo stato interessato, dando un'idea piuttosto precisa della distribuzione geografica degli utenti presenti nel nostro dataset.

Un altro tipo di analisi che si può effettuare è la classifica **Real-Time** dei giochi più giocati attualmente su Steam, mostrata nella figura successiva. Avere una classifica di questo tipo può essere molto utili se si vuole capire quali sono le tendenze del momento. Essere a conoscenza di quali categorie di giochi sono preferite dagli utenti (eg. Strategia, Sparatutto, Sportivi, etc…) può essere un'indicazione cruciale sulle decisioni progettuali per lo sviluppo del prossimo gioco di un'azienda. Difatti, se nelle prime posizioni della classifica c'è una grande maggioranza di Sparatutto, probabilmente potrebbe essere una buona scelta sviluppare un gioco che appartenga a quella categoria. L'elenco mostrato in figura mostra solo i primi risultati dei titoli più giocati dagli utenti che abbiamo analizzato. Il primo record mostra come i giocatori collegati a Steam non sono necessariamente impegnati in qualche sessione di gioco, ma probabilmente sono solo *parcheggiati* nella home del client di Steam.

Best Games:

Game Id	Game Name	Players Online
No Game	"Steam Home"	2750
730	"Counter-Strike: Global Offensive"	31
570	"Dota 2"	11
578080	"PLAYERUNKNOWN'S BATTLEGROUNDS"	8
230410	"Warframe"	7
252950	"Rocket League"	4
271590	"Grand Theft Auto V"	4
4000	"Garry's Mod"	3
624090	"Football Manager 2018"	3
292030	"The Witcher 3: Wild Hunt"	3
359550	"Tom Clancy's Rainbow Six Siege"	3
238960	"Path of Exile"	2
706220	"Black Desert Online SA"	2
365590	"Tom Clancy's The Division"	2
582660	"Black Desert Online"	2
10	"Counter-Strike"	2
227300	"Euro Truck Simulator 2"	2

Abbiamo effettuato anche analisi di diverso tipo, rappresentati nelle figure successive, dove **AverageTime** è il tempo medio delle sessioni di gioco per ogni giocatore e **STDDEV** la deviazione standard:

Tempo medio per sessione

gameid	numberOfPlayer	AverageTime	STDDEV
None	2924	1.690467359050445	4.1666809131031926
730	73	5.271604938271605	5.727155370860415
578080	26	4.678571428571429	5.2567668660116444
570	24	6.0	5.959865770300536
359550	15	3.4	4.339190181194116
252490	12	3.4285714285714284	4.50152599278692
381210	11	3.5454545454545454	3.205109557055308
230410	9	5.666666666666667	6.020797289396148
271590	8	7.25	6.158617655657106
252950	8	5.666666666666667	6.264982043070835
4000	8	5.777777777777778	6.180165405913052
107410	5	3.6666666666666665	5.08592829940284
218620	4	1.75	0.5
624090	4	7.75	6.652067347825035
292030	4	10.5	6.3508529610858835
550	4	6.6	6.767569726275452

Nell'immagine **Player to Country** che segue, analizziamo il tempo medio delle sessioni di gioco e la deviazione standard di ogni giocatore, dividendolo per il paese di provenienza:

Player to Country

loccountrycode	numberOfPlayer	AverageTime	STDDEV
Unknown	243	6.216867469879518	5.871840100828178
RU	201	6.164750957854406	5.820360717347951
US	100	7.885135135135135	6.008524158780447
TR	94	5.540983606557377	5.700728590499546
IT	80	4.042553191489362	4.907564697544099
BR	60	7.746987951807229	6.0904631036503485
DE	53	5.369230769230769	5.627969301758714
JP	45	6.53030303030303	6.0056888337890415
GB	39	6.155172413793103	5.848381764898525
PL	37	6.020408163265306	5.949404017484886
UA	36	5.8125	5.818738073324615
CA	25	6.193548387096774	5.991212202544154
AU	23	7.611111111111111	6.161065667458038
FR	23	5.066666666666666	5.747163218273104
ES	17	5.173913043478261	5.890301535058544
SE	17	5.086956521739131	5.759679942798754

Volendo fare un'analisi più approfondita, possiamo scendere più nel dettaglio suddividendo i tempi delle connessioni, identificando l'area di provenienza dei diversi giocatori, all'interno dello stato da cui si stanno connettendo (Il valore *Unknown* indica che quel numero di giocatori non ha segnalato la regione di provenienza oppure che hanno un livello di privacy più elevato):

Player to State

loccountrycode	locstatecode	numberOfPlayer	AverageTime	STDDEV
Unknown	Unknown	243	6.216867469879518	5.87184010082818
RU	Unknown	56	6.972222222222222	5.959894654098336
TR	Unknown	37	5.088888888888889	5.384414470666164
RU	48	25	5.942857142857143	5.764714457205078
DE	Unknown	25	5.896551724137931	5.942250821665659
US	Unknown	24	7.972222222222222	5.930483531773351
TR	34	22	5.375	5.857170956434789
IT	Unknown	21	3.36	4.28057628519027
US	CA	21	7.851851851851852	6.074560469720003
BR	27	20	8.433333333333334	6.083801745468119
RU	66	18	5.863636363636363	5.9465512577856305
GB	Unknown	18	6.777777777777778	5.937710859953543
JP	40	17	6.391304347826087	6.073333145322356
BR	Unknown	15	8.285714285714286	6.1167218110917645
RU	47	15	5.0	5.44671154612273

Un'analisi ancora più approfondita della precedente è identificare la città dalla quale gli utenti si stanno connettendo. Questo potrebbe essere utile per un'ipotetica fidelizzazione dell'utenza. Se, ad esempio, emerge che una notevole fetta della nostra utenza proviene da una determinata città, si potrebbe effettuare la scelta progettuale di ambientare il prossimo titolo proprio in quella città. Viceversa, si potrebbe decidere di utilizzare, come ambientazione, una tra le città meno ricorrenti tra l'utenza, in modo da invogliare quei giocatori ad acquistare il suddetto titolo ed aumentare i margini di guadagno.

Player to City

loccountrycode	locstatecode	loccityid	numberOfPlayer	AverageTime	STDDEV
Unknown	Unknown	Unknown	243	6.216867469879518	5.8718401008281775
RU	Unknown	Unknown	56	6.972222222222222	5.959894654098336
TR	Unknown	Unknown	37	5.088888888888889	5.384414470666164
DE	Unknown	Unknown	25	5.896551724137931	5.942250821665659
US	Unknown	Unknown	24	7.972222222222222	5.930483531773351
RU	48	41460	21	5.678571428571429	5.722299275178553
IT	Unknown	Unknown	21	3.36	4.28057628519027
GB	Unknown	Unknown	18	6.777777777777778	5.937710859953545
TR	34	Unknown	18	5.964285714285714	6.039880688404261
BR	Unknown	Unknown	15	8.285714285714286	6.1167218110917645
UA	Unknown	Unknown	13	4.388888888888889	5.237184794826645
JP	Unknown	Unknown	13	7.6	6.184871185315615
RS	Unknown	Unknown	13	6.0625	6.233979467402824
PL	Unknown	Unknown	12	6.333333333333333	6.078847008469694
RU	66	42316	10	6.090909090909091	6.122982042347428
AU	Unknown	Unknown	10	7.6875	6.279264818963019
FR	Unknown	Unknown	9	3.727272727272727	5.100802075966271

Conclusioni e Sviluppi Futuri

In conclusione, il progetto è stato implementato come descritto e risultata funzionante, i dati mostrati sono stati ottenuti da dati reali raccolti dallo Streamer e analizzati tramite Spark. La visualizzazione dei risultati riesce a dare la giusta visibilità alle statistiche necessarie a prendere decisioni per il marketing e lo sviluppo. Il sistema generale risulta quindi pienamente utilizzabile per le sue funzioni.

Da questa base il progetto offre ampi margini per un ulteriore sviluppo, specialmente se affiancato da algoritmi di *machine learning* tramite i quali effettuare predizioni più precise sui singoli utenti al fine di effettuare pubblicità mirata o sconti su giochi scelti in base ai gusti dell'utente stesso.

Infatti, sarebbe molto interessante ampliare questo progetto con un algoritmo simile a quello utilizzato da Netflix per suggerire agli utenti i prossimi film e serie tv da vedere. [16]

Questo algoritmo è basato sul calcolare la similarità tra due utenti tramite le votazioni attribuite ai vari *item.* È possibile da questo stimare il giudizio di una persona su un *item* su cui non ha espresso la preferenza tramite il confronto delle valutazioni degli utenti con gusti simili.

Nel nostro caso potremmo quindi applicare l'algoritmo calcolando la similarità dei giocatori sul tempo speso nei vari giochi e consigliare ad un utente un gioco a cui non ha giocato ma che piace a chi gioca ai suoi stessi giochi.

È quindi possibile effettuare **user-profiling** ed ottenere statistiche su uno specifico utente e che possono esser messe in relazione con i risultati della nostra **query 6** che effettua un'analisi simile ma sull'intero pool di utenti.

Inoltre, per lo sviluppo di uno **user-profiling** ancora più accurato è possibile richiedere i dati alle API oltre ai giochi utilizzati quelli acquistati dall'utente e le recensioni sui giochi posseduti. In questo caso sarebbe però necessaria la richiesta di un'estensione del limite di 100.000 richiesta giornaliere.

Con questi dati sarebbe però possibile applicare l'algoritmo Market Basket Analysis [17] che ci consente di inviare richieste mirate agli utenti proponendogli magari pacchetti con vari giochi a prezzo scontato negli orari in cui sappiamo essere più incline ad effettuare acquisti.

Dal punto di vista della comunicazione, invece, il **Python-Server** può essere migliorabile portando a termine l'implementazione progettata nel capitolo: Visualizzazione dei risultati: WebApp. Come già citato in precedenza, l'architettura progettata vede il Server Back-End, su cui viene eseguito python, passare i dati ad una determinata coppia IP:Porta, sulla quale sarà in ascolto il motore JavaScript. Idealmente, non vorremmo avere nessuno stadio intermedio e che le informazioni vengano passate integralmente **Real-Time** tra i due nodi.

Sul lato della **Web-App** è possibile considerare l'idea di inserire degli input form finalizzati a passare i parametri di base per l'esecuzione delle query di ricerca. Questo significa che anziché inserire delle date di interesse, in maniera statica, al momento del lancio delle query rivolte a **SparkSQL**, è possibile rendere più dinamica la cosa rendendo l'interfaccia più user-friendly.

Riferimenti

[1] i. b. N. M. Michael Hausenblas & Nathan Bijnens, «http://lambda-architecture.net/,» 2017. [Online]. Available: http://lambda-architecture.net/.

[2] «kappa-architecture.com,» [Online]. Available: http://milinda.pathirage.org/kappa-architecture.com/.

[3] Microsoft, «Big data architectures,» 11 Novembre 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/.

[4] J. Kreps, «Questioning the Lambda Architecture - The Lambda Architecture has its merits, but alternatives are worth exploring.,» 2 Luglio 2014. [Online]. Available: https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

[5] N. Marz, «How to beat the CAP theorem,» 13 Ottobre 2011. [Online]. Available: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[6] J. García. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/ASPgems%20-%20Kappa%20Architecture.pdf.

[7] Steam, «Steam,» [Online]. Available: https://store.steampowered.com/.

[8] Steam, «Obtaining an Steam Web API Key,» [Online]. Available: https://steamcommunity.com/dev/apikey.

[9] «Steam API Wiki,» Valve, [Online]. Available: https://developer.valvesoftware.com/wiki/Steam_Web_API.

[10] S. Partner. [Online]. Available: https://partner.steamgames.com/doc/webapi.

[11] Apache, «Kafka,» [Online]. Available: https://kafka.apache.org/documentation/.

[12] K. Seguin. [Online]. Available: https://nicolaiarocci.com/mongodb/il-piccolo-libro-di-mongodb.pdf.

[13] «APACHE Http Server Project,» [Online]. Available: https://httpd.apache.org/.

[14] «jQuery: Write less, Do more.,» [Online]. Available: https://jquery.com/.

[15] M. d. team, «matplotlib,» 2012. [Online]. Available: https://matplotlib.org/.

[16] A. Blasetti, «Come Netflix capisce il film che volete vedere: Matematica, Machine Learning e Serie Tv,» 5 Marzo 2017. [Online]. Available: http://www.mathisintheair.org/wp/2017/03/come-netflix-capisce-il-film-che-volete-vedere-matematica-machine-learning-e-serie-tv/.

[17] S. Li, «A Gentle Introduction on Market Basket Analysis — Association Rules,» 25 Settembre 2017. [Online]. Available: https://towardsdatascience.com/a-gentle-introduction-on-market-basket-analysis-association-rules-fa4b986a40ce.

[18] Apache, «https://tez.apache.org/,» [Online]. Available: https://tez.apache.org/.