**UNIVERSITATEA BUCUREȘTI**

**FACULTATEA DE MATEMATICǍ ŞI INFORMATICǍ**

**SPECIALIZAREA INFORMATICĂ**

**Lucrare de licență**

**Analiza rețelelor de coautori folosind Spark**

**Coordonator ştiinţific Absolvent Conf.Dr. Șerbănuță Florin-Traian Ghidoveanu Mihai**

[**Analizarea rețelelor de coautori folosind Spark**](#_9wcw9o8sft29) **3**

[Motivația lucrării și studiul situației actuale](#_7r5ayks0u3wl) 3

[Structura unei rețele de coautori](#_2wwno4q038gl) 5

[Microsoft Academic Graph](#_b8jco8giwj2d) 8

[Microsoft Academic Knowledge API](#_94ocy27eaona) 8

[MAG în Azure Data Lake Store](#_oddp5y7apbag) 11

[Introducere în Apache Spark](#_f7rk4735osaj) 12

[De la Azure Data Lake Store către Apache Spark](#_sc7h5iyu2g29) 17

[Datele primare și agregate pentru construirea rețelei statice](#_xx1sfxoqsmzi) 19

[Algoritmul de construire al unei rețele statice](#_mey2swi5o5x8) 20

[Construirea rețelelor dinamice](#_7bac39uucuyz) 21

[Utilizarea rețelelor personale](#_loe82csp0vqo) 22

[Vizualizarea rețelelor personale](#_cwdvii3565kv) 23

[Scalabilitatea proiectului](#_agiz3x428y2u) 24

[Tehnologii folosite](#_g5xctwa8ppr0) 24

[Apache Spark și Databricks](#_fil8ysbnbdxv) 24

[Scala](#_6kiz1ydo8tjj) 25

[Microsoft Azure](#_vklbeco1pc8h) 27

[Databricks](#_cnn8sj7ng4bq) 27

[Bibliografie](#_h72hgtdlbodl) 28

[Citări](#_13dsdag6b38f) 28

# Analizarea rețelelor de coautori folosind Spark

## Motivația lucrării și studiul situației actuale

Rețelele personale sunt îndelung studiate de către sociologi pentru a descoperi tipare în modul în care oamenii își aleg prieteni sau colaboratori. O rețea de acest fel este ușor modelată sub forma unui graf, unde nodurile reprezintă oamenii și muchiile grafului reprezintă faptul că doi oameni au o relație de colaborare. Muchiile pot fi ponderate de o cuantificare a acestei colaborări. În cadrul unei rețele de copii de grădiniță, de exemplu, muchiile reprezintă relația de prietenie dintre doi copii, și pot fi ponderate de numărul jocurilor la care aleg să participe împreună fără influența profesorilor. Desigur, analiza acestor tipuri de rețele este utilă și în cuantificarea influențelor pe care unii colaboratori le pot exercita într-o rețea. Rețelele personale pot fi întocmite în toate mediile în care există oameni care colaborează unul cu celălalt, indiferent de forma de colaborare. Revenind la exemplul copiilor dintr-o grădiniță, se poate observa dacă copii din straturi sociale diferite (bogați, săraci, clasa medie) au o preferință de prieteni în același strat. Un alt studiu al rețelelor sociale poate observa tendințele adolescenților de a se apuca de fumat ținând cont de poziția lor în grupul de prieteni.[1] De asemenea, studiul rețelelor sociale poate fi folosit și pentru a analiza găștile criminale și tiparele după care au loc ucideri.[2] Lucrarea de față își propune să abordeze rețelele de coautori în cadrul cercetării. Rețelele de coautori au fost studiate și în trecut, însă rezultatele oferite de aceste analize sunt contradictorii, mai ales cele privind modul în care autorii își aleg colaboratorii. Unele lucrări argumentează că legăturile de colaborare vin în principal din homofilie, preferința cercetătorilor de a lucra cu persoane cu interese sau personalități similare[3][4]. Alte păreri favorizează cauza efectului Mathew. Acest efect reprezintă tendința cercetătorilor de a-și alege parteneri de lucru din cei care au deja succes și vizibilitate, pentru a se propulsa ei înșiși[5]. De asemenea, analizele de până acum nu au o opinie comună în ceea ce privește măsura în care numărul de citări al unui articol este influențat de numărul de coautori al acestuia, precum[6],care argumentează pentru un impact mare al coautoratului și [7], care susține opusul, cum că coautoratul ar avea doar o valoare marginală. Toate acestea fac cercetarea rețelelor de coautori un domeniu activ, în care un nou studiu poate avea un impact puternic. Astfel, aplicația noastră își propune să ofere înspre analiză astfel de rețele și să adauge acestor rețele o caracteristică care le lipsește studiilor de până acum. Cele două efecte, cel de homofilie și Mathew, sunt greu de separat în rezultatele actuale pentru că studiile acestea oferă doar o perspectivă statică asupra rețelelor. Ele oferă o viziune asupra a multe și diverse populații de cercetători, însă doar la un moment dat în timp. Proiectul nostru își propune să adauge o dimensiune longitudinală asupra acestor rețele, oferind posibilitatea de a analiza evoluția acestora în timp, idee propusă pentru studii vitoate în [8][9]. Folosind această abordare, ponderea efectelor și a influențelor coautorilor ar putea fi mult mai vizibilă. Spre exemplu, se poate urmări evoluția diverșilor caracteristici ale cercetătorilor unei rețele (e.g. numărul de citări ale articolelor lor sau indicele Hirsch) și se poate verifica corelația dintre creșterile valorilor acestor indicatori și adăugarea sau eliminarea unor coautori.

Proiectul nostru constă în alcătuirea acestor rețele personale prin modelarea informațiilor preluate dintr-o bază de date și expunerea acestor rețele pentru o viitoare analiză statistică. În plus, dată fiind mărimea considerabilă a bazei de date alese, aproximativ 175 de milioane de publicații și 210 de milioane de articole, ne asigurăm că viitoarele analize statistice vor fi relevante. Pentru a fi capabili să modelăm o cantitate atât de mare de date, am ales să folosim Spark, un framework de procesare distribuită, pe care îl vom descrie în detaliu în secțiunea referitoare la tehnologiile folosite. Deși rezultatele obținute vor fi specifice rețelelor de coautori, arhitectura aplicației poate fi aplicată oricărui tip de rețea personală, cu ușoare modificări pentru a putea acomoda noile surse de date și structura acestor date.

În continuare vom prezenta structura rețelelor personale obținute, arhitectura aplicației și modul în care am format rețelele.

## Structura unei rețele de coautori

Rețeaua de coautori este modelată, ca majoritatea rețelelor personale, sub forma unui graf. Fiecare nod al grafului reprezintă un cercetător, iar o muchie între două noduri reprezintă faptul că doi cercetători au colaborat la scrierea unor articole. Graful este orientat și simetric, dat fiind că o legătură de la cercetătorul *X* la cercetătorul *Y* vine de la sine și cu legătura inversă : dacă *X* colaborează cu *Y*, atunci este de la sine înțeles că și *Y* colaborează cu *X*.

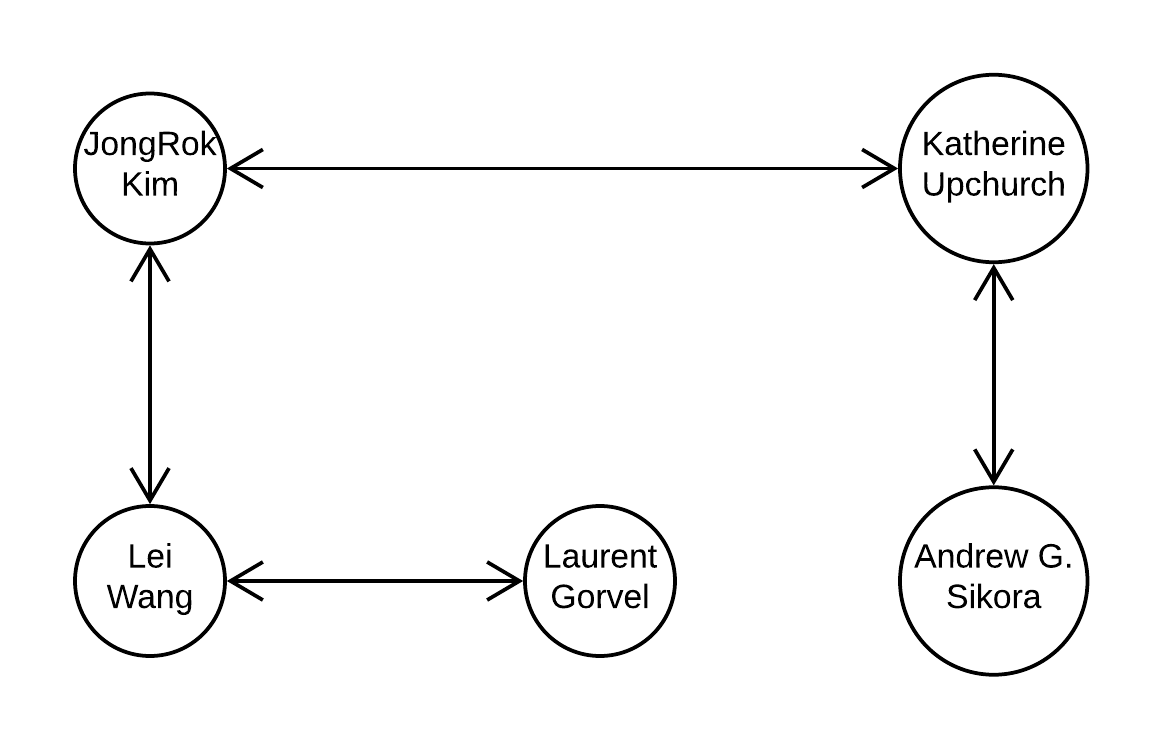


Fig. 1. Exemplu de rețea socială de cercetători

În afară de această semantică, fiecare nod și muchie conține date relevante pentru analiză. Pentru fiecare nod, reținem identificatorul unic al cercetătorului în baza de date, cât și date agregate despre acesta: numărul de publicații la care a participat, numărul de citări pe care le-au obținut aceste publicații și indicele Hirsch al autorului. Indicele Hirsch este o măsură de cuantificare atât a productivității, cât și a impactului unui autor. Acest indice a fost inventat în 2005 de către Jorge E. Hirsch și este definit astfel: un cercetător are un index Hirsch cu valoarea *h* dacă a publicat *h* lucrări dintre care toate au fost citate de cel puțin *h* ori. Este de menționat că acest indice este relevant doar în comparații referitoare la cercetători din același domeniu de studiu, deoarece convențiile de citare diferă de la un domeniu la altul.[10] De pildă, numărul de citări primite de articole din domeniul științelor naturale este de până la 6 ori mai mare decât cele primite în domeniul matematicii și al știnței calculatoarelor [11].

Rețele personale care pot fi extrase cu ajutorul aplicației noastre sunt de două tipuri: rețele statice, care reprezintă statusul unei rețele în momentul actual, și rețelele dinamice, care oferă informații despre evoluția statusului unei rețele în timp. Cele două tipuri diferă prin structura muchiilor care leagă două noduri în graf. Vom explica, pe rând, structura unei muchii pentru fiecare tip de rețea. Muchiile rețelelor statice rețin, pe lângă identificatorii unici ai celor doi autori care colaborează, și numărul de publicații pe care cei doi autori le-au scris împreună.

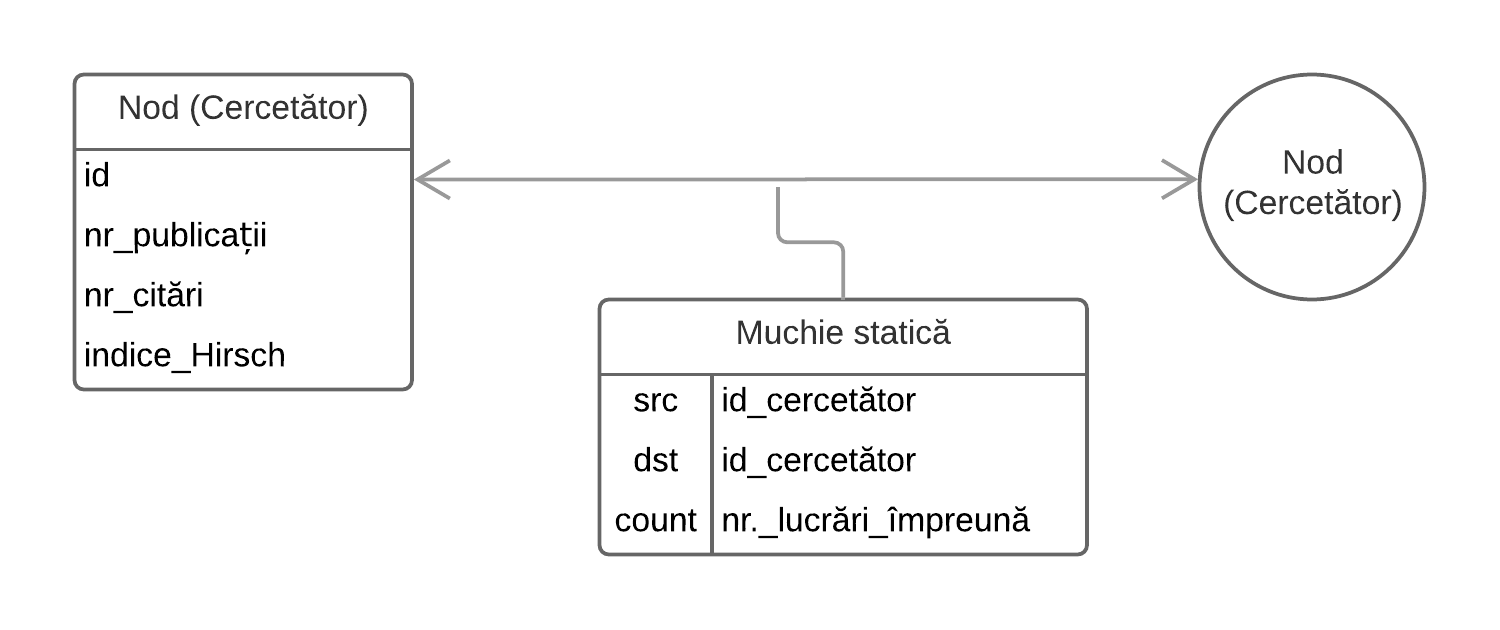


Fig. 2. Structura unei rețele personale cu muchii statice

Rețelele dinamice sunt mai complexe și modelează evoluția autorilor și a lucrărilor acestora pe parcursul unui interval de ani dat. Spre exemplu, o muchie a acestui tip de rețea va reține o structură care expune, pentru fiecare an din ultimii zece ani, numărul de citări al fiecărui articol la finalul acelui an și numărul de citări însumat al acestor articole în anul respectiv.

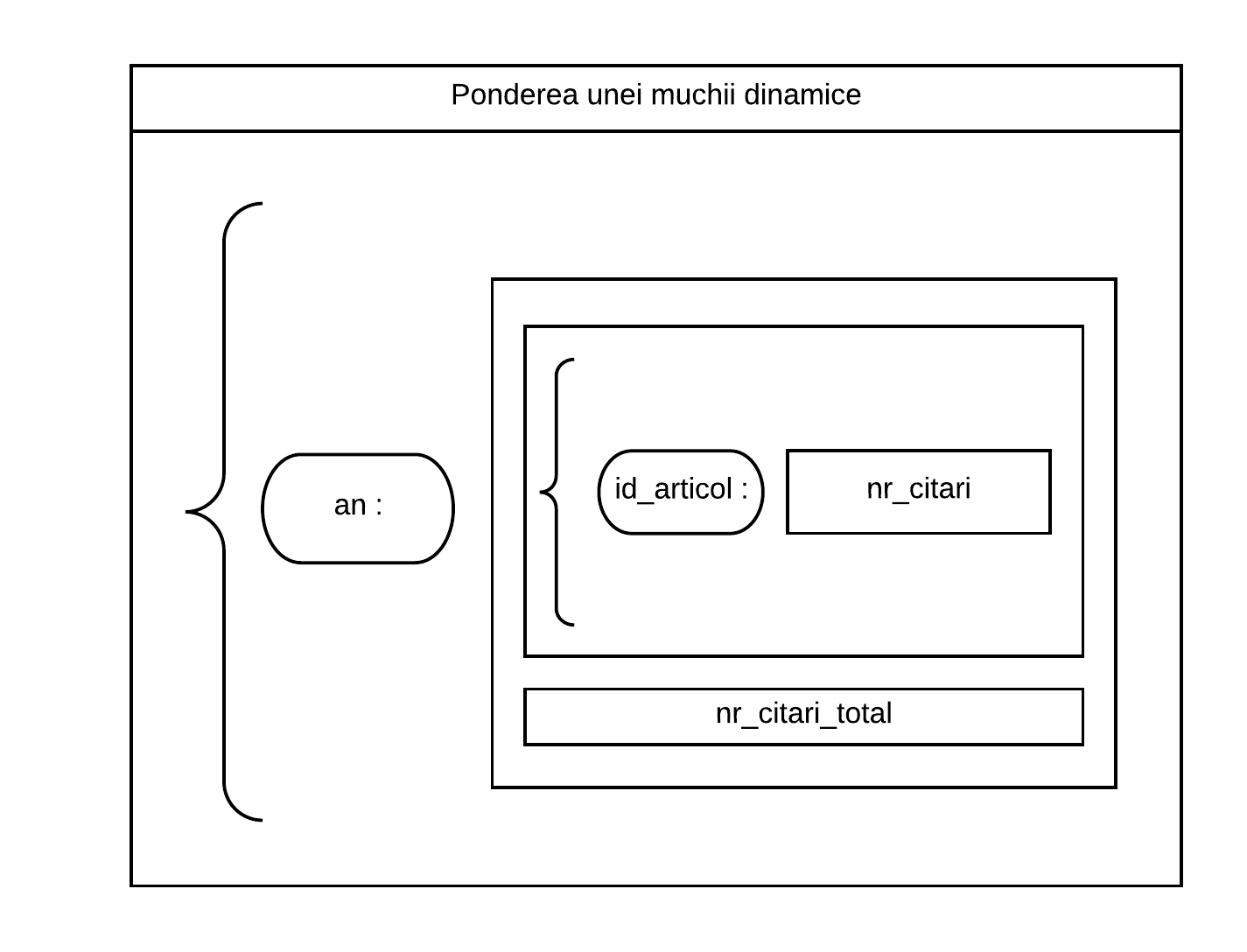


Fig. 3. Ponderea unei mucii dinamice

Datorită faptului că rețelele statice au mai puține date agregate decât cele dinamice, acestea sunt mai ușor de calculat și au o dimensiune mai redusă, ceea ce le face mai ușor de stocat și de expus grafic. Din această cauză, deși poate părea redundant faptul că informațiile din cele statice sunt incluse în informațiile din cele dinamice, ușurința lor de modelare le face excelente în scopul previzualizării. De exemplu, se pot urmări mai întâi factori precum dimensiunea rețelei și numărul mediu de publicații scrise în coutorat, și apoi a se decide dacă rețeaua în cauză merită să fie analizată mai amănunțit, folosind versiunea sa dinamică.

În cele ce urmează vom prezenta sursa noastră de date în modelarea acestor rețele, cât și modul în care le-am obținut și problemele întâlnite pe parcurs.

## Microsoft Academic Graph

Microsoft Academic Graph, prescurtat în continuare **MAG**, este o bază de date care conține informații despre jurnale, conferințe și publicații ștințifice, despre autorii acestora și despre diverse domenii de studiu ale cercetării. După cum spune și numele, informația este modelată sub forma unui graf cu enități ce modelează toate aceste activități de cercetare. Fiind preluate cu ajutorul motorului de căutare Bing, datele sunt updatate săptămânal și sunt folosite de către Microsoft în servicii precum Microsoft Academic, Cortana, Word sau însăși Bing.[12,13]

La momentul actual, baza de date are o dimensiune de *388 GB*, ceea ce o face să conțină o cantitate îndeajuns de mare de informație pentru a fi relevantă statistic. Acest factor, împreună cu prezența informațiilor despre citările fiecărui articol, fac din Microsoft Academic Graph o alegere potrivită pentru scopul nostru, extragerea rețelelor personale de coutori.

Pentru a prelua datele din MAG, am avut la dispoziție două opțiuni: să folosim API-ul Microsoft Academic Knowledege sau să importăm întreaga baza de date în Azure Data Lake Store. Inițial, am încercat să folosim API-ul, dar datorită a ce am aflat pe parcurs, am schimbat abordarea către Azure Data Lake Store.

## Microsoft Academic Knowledge API

API-ul constă în patru puncte de acces de tip REST pentru preluarea informației din MAG:

* *interpret*
* *evaluate*
* *calcHistogram*
* *graph search*

Metoda *evaluate* este principala folosită, împreună cu o sintaxă specială, disponibilă la [14], pentru a interoga graful, iar celelalte trei sunt folosite fie pentru a ușura sau rafina căutarea (*interpret*, respectiv *graph traversal*), fie pentru a oferi date statistice despre atributele entităților din graf(*calcHistogram*).

Metoda *interpret* este o metodă auxiliară care transformă un text în limbaj natural în sintaxa specială folosită de *evaluate*. În urma experimentării cu API-ul, am observat că metoda *interpret* dă rezultate greșite în multe dintre cazuri, de aceea am ales să folosim *evaluate* pentru a căuta informații în baza de date. Deși a fost nevoie de timp în plus pentru a înțelege și învăța sintaxa acestor cereri, ele au dat rezultate mult mai exacte decât cele oferite de *interpret*. [13]

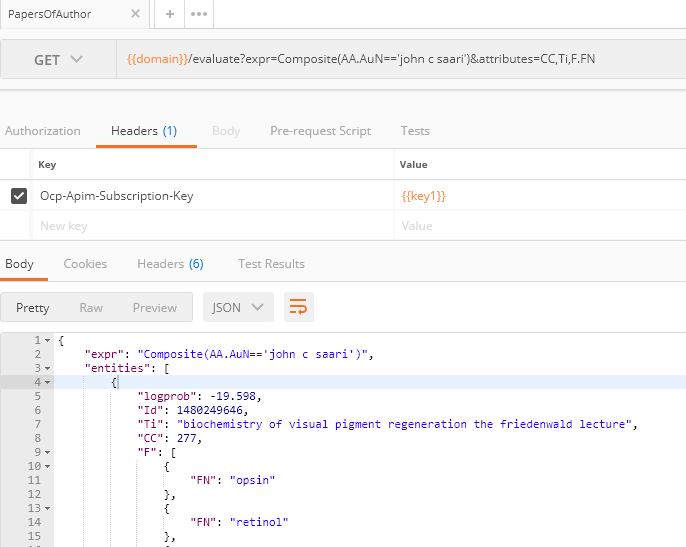


Fig. 4. Exemplu de utilizare a metodei Evaluate a API-ului

Existența acestor metode predefinite și ușurința de a accesa API-ul ne-au făcut să optăm pentru această modalitate de a accesa datele din MAG, pentru început. Însă nu mult mai târziu am dat peste câteva probleme majore care ne-au făcut să ne îndreptăm atenția spre cealaltă metodă de preluare a datelor. În continuare vom prezenta aceste probleme întâlnite.

Pentru a înțelege și experimenta modul de funcționare al API-ului, am folosit *Postman*, un program folosit pentru a crea și a testa cereri HTTP. În urma acestor experimente, am observat că metoda *evaluate* putea întoarce cel mult 1000 de entități odată, ceea ce creea nevoia de multe cereri HTTP pentru a extrage cantitatea mare de date. Folosind atributul *offset* al metodei, puteam să obținem primele 1000 de entități, apoi următoarele 1000 și tot așa. Dar pentru că cererile HTTP eșuau spontan și repetat când foloseam metoda cu atributul *offset* setat la peste un milion, a fost nevoie de introducerea unui nivel de tratare al erorilor. Când atributul ajungea la acest număr, request -ul urma să fie repetat de 3 ori, la un interval de câteva secunde, înainte de a fi considerat eșuat.

În plus, numărul mare de entități cerea stocarea lor într-un mod distribuit, astfel că am fi avut nevoie să integrăm structurile de date din Apache Spark cu request-urile la API și apoi să stocăm datele în fișiere pentru a evita reluarea request-urilor de fiecare dată când rulam aplicația. Pentru a putea integra acest modul al aplicației noastre cu Spark, a fost scris în Scala și am folosit librăriile *Play-WS* pentru cererile HTTP. Dar acest modul nu a mai fost folosit mai departe. Toate detaliile discutate mai sus ar fi dus la o complexitate mărită a nivelului de tratare al erorilor aplicației, la un timp îndelungat de extragere a datelor și la eventuale întârzieri neașteptate din cauza erorilor API-ului.

În al doilea rând, am ajuns la concluzia că, folosind metoda *evaluate* putem obține date doar despre două milioane de publicații ștințifice. În același timp, Microsoft Academic, deci și baza de date MAG peste care este construit, oferă informații despre peste 175 de milioane de publicații, la momentul actual. Astfel, tot potențialul Microsoft Academic Graph ar fi rămas nefolosit. [15]

Datorită acestor inconveniențe, am decis să folosim Azure Data Lake Store, sau **ADLS**, pentru a obține datele din MAG. Deși mai greu de configurat inițial, s-a dovedit că prin această modalitate puteam accesa în întregime baza de date și o puteam accesa fără întârzieri și erori, direct din API-ul Spark, fără să mai fie nevoie să folosim aplicația greoaie care se baza pe request-uri HTTP. Deși setările necesare pentru a importa baza de date în ADLS au durat mai mult și a fost necesar să primim aprobarea echipei Microsoft, rezultatele au fost cu mult mai satisfăcătoare față de folosirea API-ului.

## MAG în Azure Data Lake Store

ADLS este un serviciu de stocare în cloud, oferit de către Microsoft Azure. Serviciul asigură durabilitatea și siguranța datelor stocate prin copierea acestora. Redundanța astfel creată face posibilă recuperarea datelor în cazul erorilor neașteptate. De asemenea, serviciul poate stoca fișiere de orice mărime și de orice tip, de la câțiva kilobytes la petabytes. ADLS este conceput și optimizat pentru a expune date spre procesare și analiză, exact scopul proiectului nostru. De asemenea, sistemul de fișiere pe care îl folosește, *ADL*, este un sistem distribuit perfect compatibil cu Hadoop Distributed File System, acesta fiind compatibil la rândul lui cu Spark, framework-ul pe care îl folosim la procesarea distribuită necesară proiectului nostru. [16]

Pentru a ne putea conecta la ADLS, trebuie să folosim sistemul de autentificare Azure Active Directory, sau **Azure AD**. Pentru acest sistem de autentificare avem două opțiuni: autentificare de tip *end-user*, sau *service-to-service*. Am ales să folosim a doua variantă dat fiind faptul că este metoda preferată de Microsoft pentru a încărca datele în ADLS, dar și pentru că este varianta care permite accesul automatizat la ADLS, fapt convenabil pentru aplicația noastră Spark pe care o vom descrie în capitolele următoare. Autentificarea *service-to-service* are ca rezultat faptul că operațiile de citire și scriere în ADLS vor fi intermediate de o aplicație Active Directory Web, numită în continuare **ADW**, care primește drepturile necesare de operație asupra sistemului de fișiere. ADW este disponibilă în cloud și, după adăugarea drepturilor de acces necesare, alte aplicații se pot conecta la ea cu următoarele credențiale: id-ul și cheia secretă a ADW, cât și id-ul contului Azure care găzduiește ADW. După cum spuneam mai sus, echipa Microsoft folosește aceste credențiale pentru a se conecta la aplicația noastră ADW și a încărca, săptămânal, o copie actualizată a MAG-ului în serviciul nostru de stocare.[17][18] De aici, datele vor fi preluate de componenta Spark a aplicației noastre și vor fi transformate în rețelele personale dorite.

## Introducere în Apache Spark

Apache Spark este un framework folosit pentru procesarea Big Data, și reprezintă în același timp și principala tehnologie folosită în proiectul nostru pentru obținerea rețelelor personale. Performanțele sale și de ce am ales să folosim acest framework sunt detalii ce vor fi discutate pe larg în secțiunea despre tehnologii a lucrării de față. Dar pentru moment, pentru a ușura înțelegerea metodologiei folosite în acest proiect, vom explica care sunt principalele structuri de date folosite în Spark și principiile centrale ale acestui framework. Toate acestea au fost folosite extensiv în obținerea rețelelor personale și au jucat un rol important în întocmirea acestui proiect.

În genere, o aplicație Spark este reprezentată de o componentă *driver* care apelează funcția *main()* a programului și execută pe nodurile de tip *worker* ale unui cluster, operațiile în paralalel care apar. Comunicarea dintre *driver* și nodurile *worker* este intermediată de un *Cluster Manager*, care orchestrează împărțirea sarcinilor în interiorul cluster-ului. *Cluster Managerul* poate fi manager-ul de sine stătător din Spark, sau orice manager compatibil, cum ar fi Mesos, Yarn, sau Kubernetes. [19][20] Aplicația noastră folosește Azure Databricks drept *cluster manager*.

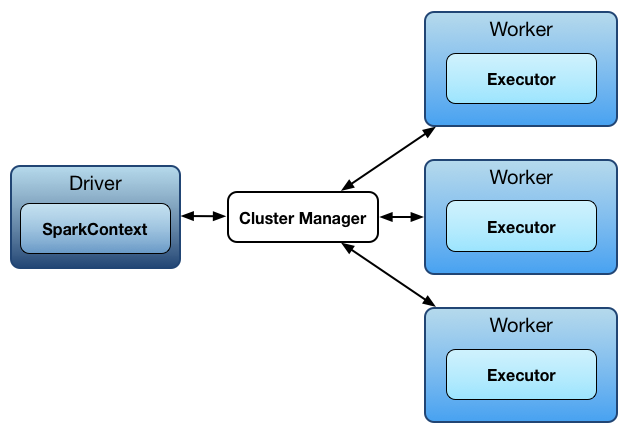


Fig.4. Arhitectura Apache Spark

Principala structură de date folosită de Spark este un **RDD** (Resilient Distributed Dataset), o colecție imutabilă și distribuită de elemente către nodurile cluster-ului, care suportă diverse operații în paralel. RDD-urile pot fi create având ca sursă un fișier dintr-un sistem de fișiere distribuit sau o secvență de elemente generată în *driver*. Folosind un mecanism de *cache*, un RDD poate fi salvat în memorie pentru a putea fi reutilizat cu ușurință de către sarcinile programului. De asemenea, după cum spune și termenul "Resilient", acest tip de colecție se recuperează automat în urma erorilor neașteptate ale cluster-ului.

Există două tipuri de operații aplicabile unui RDD: transformările și acțiunile. O transformare crează un nou RDD, cu elemente modificate plecând de la un RDD deja existent, în timp ce o acțiune execută calcule cu elementele unui RDD și întoarce valoarea obținută programului *driver*. Un mare avantaj al Spark este faptul că transformările sunt mereu executate în mod leneș. Aceasta înseamnă că procesarea elementelor și calculele necesare nu au loc la executarea transformării, ci doar când RDD-ul transformat este supus unei acțiuni. Până în acel moment, orice transformare se adaugă unui plan de execuție, fără a fi de fapt aplicată. Acest lucru ne permite să ne verificăm codul mult mai ușor și să modelăm cantități mari de date, precum sursa noastră MAG, fără ca nodurile să execute operațiile de fiecare dată. O acțiune cere ca un rezultat să se întoarcă la *driver*, și atunci planul de execuție al RDD-ului trebuie pus în aplicare pentru a avea un rezultat tangibil. Un dezavantaj al acestui proces este că toate transformările din plan trebuie executate mereu, de la cap la coadă, de fiecare dată când acțiunea este apelată, chiar dacă este aceeași. Dar dacă alegem să persistăm RDD-ul în memorie, atunci problema este rezolvată, rezultatele transformărilor fiind reținute după ce au fost executate prima oară. [20]

Alte abstractizări importante în Spark sunt Dataset și DataFrame, care împart același API în implementarea lor. Acestea sunt și ele colecții imutabile și distribuite de elemente, precum RDD-ul, dar beneficiază de optimizări mai puternice în spate. Fac parte din Spark SQL, un modul care permite execuția de cereri SQL către sursele de date distribuite. Acest modul folosește caracteristici funcționale din Scala precum *pattern-matching* și *quasiquotes* pentru a optimiza cererile Spark. Dataset vine cu o optimizare în plus, în privința tipului de date al entității pe care o conține. Pentru a serializa obiectele din Dataset, este folosit un *Encoder* specializat, specific tipului obiectului dat spre serializare. Acest *Encoder* este reprezentat de cod generat dinamic și transformă obiectul într-o secvență de biți a cărei format îi permite să poată facă unele operații precum filtrarea și sortarea fără să mai deserializeze obiectul.[21] [22]

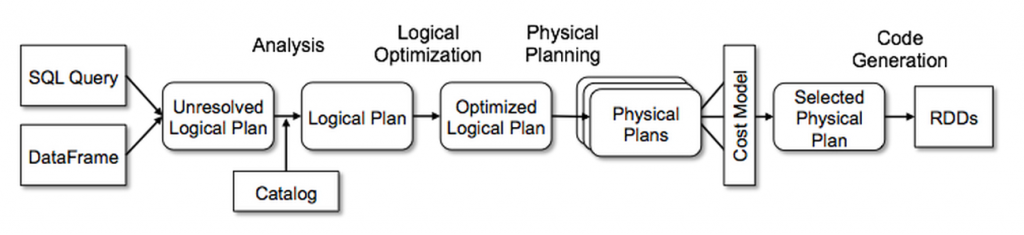


Fig.5. Plan de execuție și optimizare în Spark SQL

După cum am spus, DataFrame-ul și Dataset-ul împart același API. Mai exact, DataFrame-ul este definit ca un Dataset care nu ține cont de tipul elementelor din interior. Elementele dintr-un DataFrame sunt de tipul *Row*, iar un *Row* poate conține oricâte atribute de orice tip, cu nume asociat fiecărui atribut. Se poate observa cu ușurință că această structură este foarte similară cu un tabel SQL și, de altfel, metode specifice SQL-ului (*select*, *join*) pot fi aplicate atât DataFrame-ului cât și Dataset-ului. Aceste colecții oferă o interfață facilă de a interacționa cu datele structurate și, împreună cu cererile Spark SQL, folosesc același motor de executare, ceea ce permite trecerea ușoară de la un una la alta. [23]

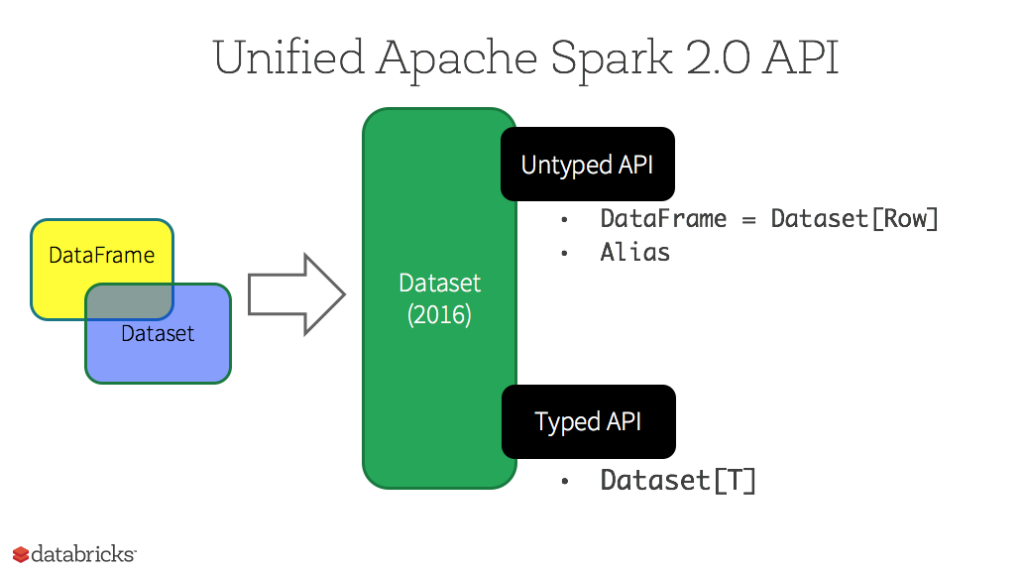


Fig. 6. Unificarea API-urilor Dataset și DataFrame

Deși RDD-ul este abstractizarea de bază a Spark, este și cea mai veche, celelalte două colecții fiind construite peste aceasta. De aceea, este recomandat ca RDD-urile să fie folosite doar pentru operații distribuite de nivel jos, cum ar fi salvarea colecțiilor distribuite în fișiere sau manipularea partițiilor ocupate de date în nodurile *worker*. Spre deosebire de RDD, API-ul Dataset este recomandat pentru oprații cu o abstractizare mai înaltă, precum analizarea și procesarea datelor într-un mod interactiv, ceea ce se potrivește mai bine nevoilor proiectului nostru. În plus, dat fiind că Dataset-urile sunt construite peste RDD-uri, cele două colecții pot fi transformate cu ușurință din una în cealaltă. [23]

De-a lungul lucrării, vom preciza caracteristicile Spark folosite pentru a întocmi diverse acțiuni în stadiile de prelucrare, modelare și procesare a datelor. Astfel, informațiile de mai sus vor fi benefice în a înțelege de ce am ales anumite metode și de ce anumite abordări funcționează.

## De la Azure Data Lake Store către Apache Spark

Pentru a putea citi datele din ADLS, am autorizat componenta noastră Spark folosind același tip de autentificare *service-to-service* către ADW, prezentată mai sus. Cele trei credențiale au fost adăugate în configurațiile *Spark Session* și *Spark Context*, două entități care dau mai departe configurațiile către principalele structuri de date folosite în Spark, *RDD*, *Dataframe* și *Dataset*.

Baza de date MAG din ADLS are o structură asemănătoare cu o bază de date relațională, în ciuda semanticii sale de graf. Datele sunt reprezentate sub forma unor fișiere text, care conțin informații în formă tabulară: câte o enitate pe linie, cu atributele fiecărei entități despărțite de caracterul tab(‘\t'). Fiecare fișier conține informații despre un anumit tip de entitate. De exemplu, avem fișierul *Authors.txt*, cu informații despre autori, *Papers.txt*, cu date despre publicații, și *PapersAuthorsAffiliations.txt*, care servește drept o tabelă de legătură între autori, articole și parteneriatele autorilor.

Pentru a putea modela datele cu ușurință, am încărcat fișierele text în memorie, am segmentat liniile după caracterul separator '\t' și am creat câte o clasă de obiecte pentru fiecare entitate reprezentată de un fișier.

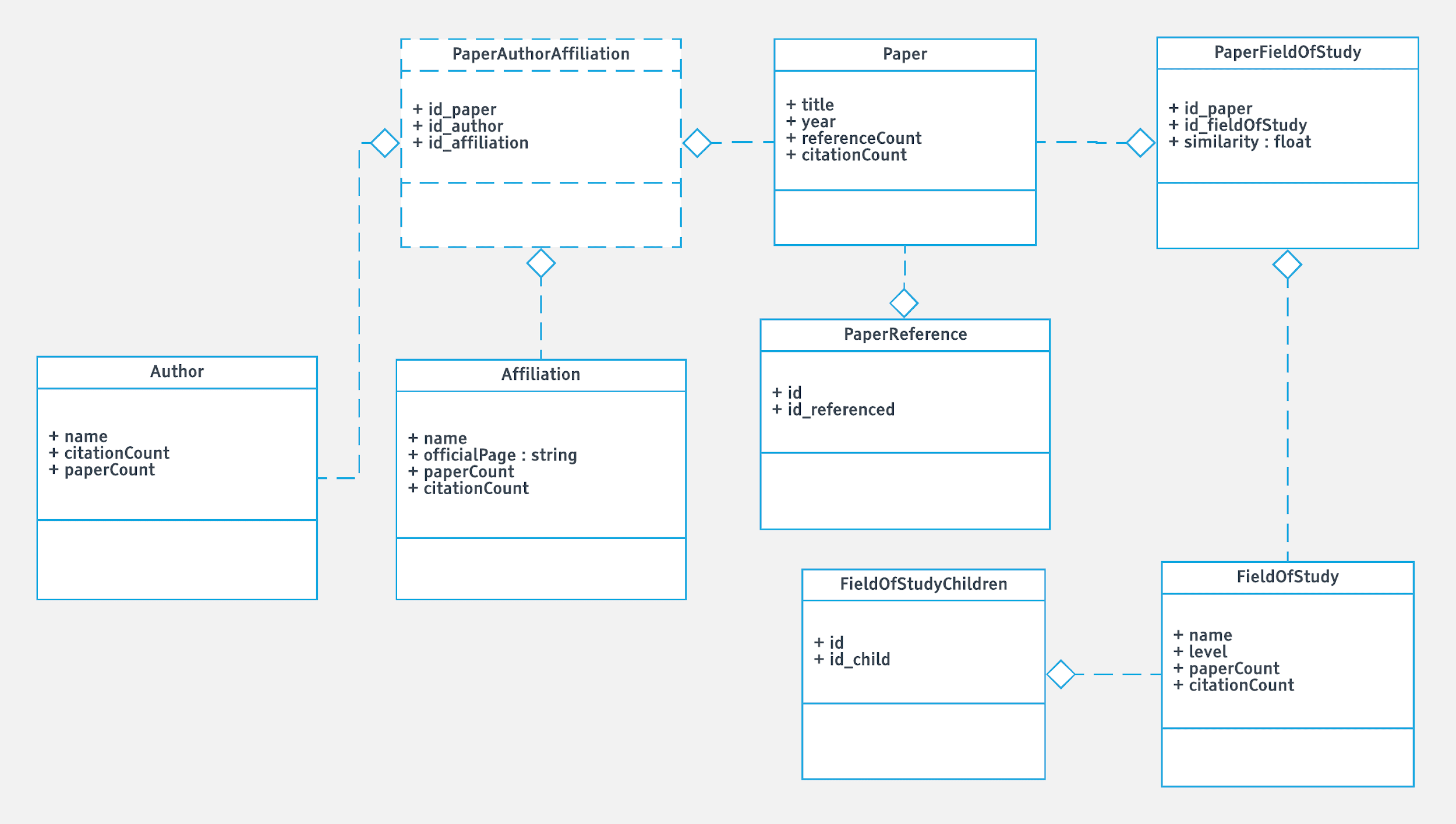


Fig. 7. Schema claselor folosite pentru a modela baza de date MAG în Spark

Dată fiind dimensiunea mare a unor fișiere, de la 10 *GB* la aproape 70 *GB*, am stocat toate obiectele extrase dintr-un fișier într-un *Dataset* corespunzător clasei de obiecte, distribuit în cluster-ul nostru. Astfel, fiecare fișier din baza de date are un Dataset corespunzător cu același nume în aplicația noastră.

Deținând informația despre tipul obiectelor, procesările pe care le facem asupra colecției vor beneficia de optimizările *Encoder*-ului. În plus, evaluările leneșe specifice Spark ne dau posibilitatea de a efectua transformări peste aceste fișiere de dimensiuni mari într-un timp scurt, urmând să așteptăm doar atunci când apelăm o acțiune asupra Dataset-ului. Dacă acest tip de evaluări ar fi lipsit, etapa de programare a aplicației ar fi fost mult mai anevoioasă. Pentru a testa fiecare nouă transformare, ar fi trebuit să așteptăm procesarea a zeci de *GB* de date. Sau să testăm toate transformările odată cu acțiunea, ceea ce ar fi mimat evaluarea leneșă, dar ar fi făcut mult mai greu să identificăm erorile logice ale codului.

## Datele primare și agregate pentru construirea rețelei statice

Pentru a putea crea rețeaua personală a unui cercetător, trebuie să putem obține următoarele date primare :

* Articolele scrise de către un anumit cercetător
* Autorii unui anumit articol
* Cercetătorii care activează într-un anumit domeniu de studiu

Am definit funcții care obțin aceste date folosind Dataset-urile entităților principale din MAG, precum *Authors*, *Papers* și *FieldsOfStudy*, legate printr-o metodă *join* cu Dataset-urile de legătură *PaperAuthorAffiliation* și *PaperFieldsOfStudy*. Am denumit funcțiile în mod reprezentativ, *authorsOfPaper*, *papersofAuthor* și *authorsOfField*.

Deși nu participă în algoritmul de construire a rețelei, avem nevoie și de date agregate referitoare la cercetători, pentru a fi expuse în rețea pentru viitoarele analize statistice:

* Numărul de citări al autorului
* Numărul de publicații al autorului
* Indicatorul Hirsch al autorului

Primele două sunt deja prezente în baza de date MAG ca atribute în tabelul *Authors*, deci foarte ușor de obținut. Vom descrie în continuare algoritmul folosit pentru a calcula indicatorul Hirsch. Mai întâi, obținem toate publicațiile autorului și le ordonăm descrescător după numărul de citări obținute. În această așezare, indicele Hirsch este poziția ultimei publicații pentru care numărul de citări este mai mare sau egal cu numărul de ordine al poziției sale. De exemplu, să considerăm un autor cu patru publicații, fiecare având 60, 32, 10, respectiv 3 citări. Index-ul Hirsch al acestui cercetător este 3, pentru că lucrarea de pe poziția 4 are doar 3 citări( mai puține decât indicele poziției, 4), iar lucrarea de pe poziția 3 are 10 citări (mai multe decât indicele poziției, 3). [10]

Algoritmul este ușor de implementat, folosind funcția primară de mai sus, *papersOfAuthor*. Articolele autorului extrase astfel sunt apoi sortate, după numărul de citări al fiecăreia, valabil ca un atribut al entității. Având publicațiile astfel sortate, ne rămâne doar să urmăm regula de mai sus pentru a calcula indicele dorit.

Având la dispoziție datele necesare, suntem gata de a construi o primă rețea personală a coautorilor unui cercetător.

## Algoritmul de construire al unei rețele statice

După cum am discutat în secțiunea “*Structura unei rețele personale”,*  o astfel de rețea este reprezentată de o listă de noduri și o listă de muchii. Pentru a afla coautorii unui cercetător, vom prelua toate publicațiile lui, și pentru fiecare din acestea vom alege ceilalți autori ai publicației înafară de el. Toate acestea sunt efectuate folsoind funcțiile din secțiunea de mai sus.

Este posibil ca un cercetător să scrie mai multe publicații în colaborare cu un altul, așa că algoritmul urmărește, pentru fiecare publicație, două cazuri: coautorii noi descoperiți și cei deja descoperiți. Pentru un coautor care nu a mai fost găsit printre autorii unei publicații până atunci, adăugăm două noi muchii direcționate între el și cercetătorul "sursă" al rețelei, ponderate cu 1 pentru că până acum au o singură publicație la care au colaborat amândoi.

În cel de-al doilea caz, pentru un coautor care a fost descoperit înainte, nu mai este nevoie să adăugăm o nouă muchie, ci doar să le actualizăm pe cele vechi incrementând numărul publicațiilor scrise împreună cu 1.

Dorim ca rețeaua personală a unui cercetător să evidențieze întreaga "familie" de colaborare, și de aceea, aplicăm recursiv algoritmul de mai sus pentru toți coautorii "sursei", de data aceasta avându-i pe fiecare dintre ei ca "sursă". Ulterior, vom calcula coautorii coautorilor și tot așa, până când nu mai descoperim noi coautori.

Pentru a implementa aceasta, vom folosi o listă care să rețină cercetătorii ce vor fi "surse". Inițial, în listă va fi cercetătorul pentru care este construită rețeaua, iar mai apoi, fiecare coautor nou descoperit va fi adăugat în această listă. După ce terminăm de parcurs publicațiile "sursei", o ștergem pe aceasta din listă și repetăm tot algoritmul pentru următoarea "sursă" din listă. Vom continua să iterăm acest algoritm până când lista de "surse" va fi goală.

## Construirea rețelelor dinamice

Pentru calculul rețelelor personale dinamice, vom avea nevoie de aceleași date primare ca și rețelele statice, cu o mențiune în plus: - Numărul de citări primite de un articol într-un anume an

Am creat o funcție care, folosind Dataset-urile *Papers* și *PaperReferences*, numără câte dintre toate articolele apărute într-un an dat citează un articol dat. Funcția folosește aceleași metode *join* și *filter* ca mai sus. Folosind această informație primară, am calculat următoarele date agregate: - Numărul total de citări pe care le deține un articol într-un anume an - Indicatorul Hirsch al unui autor într-un anume an

Observăm că rezolvarea primei probleme duce la calculul simplu al celei de-a doua. Folosind același algoritm ca și pentru rețelele statice, putem calcula acest indicator pentru orice autor odată ce avem o listă cu numărul total de citări ale fiecărui articol al său într-un anume an. Așa că vom explica în detaliu doar tehnica folosită pentru prima problemă.

Utilizând funcția primară definită mai sus, putem calcula numărul de citări primit de un articol în fiecare an de la publicarea sa până în prezent. Având aceste numere, obținem numărul cumulat de citări într-un anumit an însumând toate citările la anul publicației până în acel an. Pentru că numărul de citări cumulate într-un an depinde mereu de numărul de citări cumulate în anul anterior, observăm că funcția noastră are proprietatea Markov.[25]. Pentru a evita repetarea acelorași calcule de mai multe ori, funcția noastră întoarce o secvență cu numărul de citări cumulate pentru fiecare an de la publicația articolului până în prezent. Iar în continuare, de fiecare dată când avem nevoie de datele agregate în discuție, este suficient să accesăm această secvență, nemaifiind nevoie să reapelăm funcția. Pentru a beneficia de timp de acces mic, am modelat secvența sub forma unei tabele de dispersie, indexată după an.

Construirea unei rețele dinamice este foarte asemănătoare cu cea a unei rețele statice, construirea muchiilor fiind singura diferență. Pentru fiecare coautor nou găsit, calculăm tabela de dispersie prezentată mai sus pentru fiecare articol de-al său scris în colaborare cu cercetătorul "sursă". Apoi, modelăm această tabelă conform structurii discutate în *Structura unei rețele personale*, și o adăugăm ca pondere a muchiei dintre coautor și sursă. Ca în cazul rețelelor statice, întreg algoritmul este repetat recursiv pentru a adăuga în rețea toată "familia" de colaborare a "sursei".

## Utilizarea rețelelor personale

Scopul proiectului de față este obținerea rețelelor personale, ci nu analizarea lor statistică, acest fapt fiind lăsat la latitudinea viitorilor utilizatori. Pentru aceștia, am întocmit această secțiune pentru a exemplifica cum aceste rețele pot fi extrase din mediul Azure sau chiar analizate în continuare folosind Spark.

Creatorii Apache Spark au întomit o librărie pentru procesări distribuite de grafuri, numită GraphFrames. Este ușor de văzut că, dat fiind că rețelele personale extrase de noi iau forma unui graf, această librărie li se potrivește perfect. Librăria oferă diverși algoritmi specifici grafurilor, precum *PageRank*, calcularea celui mai scurt drum, numărarea triunghiurilor sau găsiri de tipare. Un GraphFrame se construiește dintr-un DataFrame de noduri, care trebuie să conțină un atribut identificator unic, și un DataFrame de muchii, care să conțină două atribute cu identificatorul unic al nodului sursă, respectiv nodului destinație. Dat fiind că rețelele noastre vin sub forma unor Dataset-uri cu entități corespunzătoare, transformarea acestora în DataFrame-uri se face folosind o conversie standard din Spark. [26]Acestea sunt metode de analiză structurală, mai degrabă decât statistică.

Pentru cea din urmă se poate folosi programul Siena, disponibil și ca o librărie pentru limbajul R (RSiena). Acest program este conceput de mai mulți cercetători, printre care și Tom Snijders, și se concentrează pe analiza statistică a rețelelor sociale. Siena explorează rețelele longitudinal, de-a lungul unei perioade de timp, iar modelele sale de noduri sunt structurate ca un lanț Markov, afișând informația cumulată de la un moment la altul.[27] Astfel, modul în care am construit rețelele le face compatibile cu acest program.

Pentru a putea fi analizate de Siena, rețelele trebuie stocate local. Ceea ce nu este o problemă, pentru că, din mediul Databricks, odată ce am obținut rețelele, le putem scrie într-un fișier în ADLS, de unde le putem descărca cu un singur click.

## Vizualizarea rețelelor personale

De asemenea, poate fi util să creezi vizualizări ale rețelelor personale, pentru a le prezenta înainte sau după analiza statistică. Dată fiind modelarea lor sub forma unui graf, este convenabil să le reprezentăm în format GEXF(Graph Exchange XML Format). Acesta este un format similar XML-ului în care informații despre nodurile, muchiile și datele agregate ale acestora pot fi salvate. În această formă, grafurile pot fi încărcate în diverse programe de vizualizare. Unul din acestea este Gephi, o aplicație open-source pentru modelarea și analizarea rețelelor. Folosind acest program, utilizatorii pot partiționa rețelele după anumite atribute, pot controla mărimea și culoarea nodurilor cât și a muchiilor. [28] În plus, dezvoltatorii pot folosi rețelele în acest format pentru a oferi rețelele spre analiză într-un client web, folosind librăria de JavaScript *Linkurious*. Librăria este construită pentru grafuri cu până la miliarde de noduri, deci rețelele noastre personale vor putea fi procesate fără probleme. Totuși *Linkurious* este o librărie complexă care oferă mai multe funcționalități, precum filtrare, securitatea informațiilor și detectarea anomaliilor. Pentru cei care vor doar să obțină o vizualizare interactivă, recomandăm librăria de JavaScript *D3* (Data-Driven-Documents). Această librărie este open-source și este folosită pentru vizualizări diverse, de la histograme la rețele, deci are un grad mare de libertate.

## Scalabilitatea proiectului

## Tehnologii folosite

### Apache Spark și Databricks

Date fiind cele *388 GB* ale bazei de date MAG, am avut nevoie de un mediu capabil să stocheze cantități mari de date și tehnologii capabile să le proceseze într-un mod eficient. Astfel, am ales să folosim Apache Spark ca tehnologie principală a proiectului nostru.

Apache Spark este un proiect open-source care și-a găsit începuturile în lucrarea de doctorat a lui Matei Zaharia, la Universitatea din California, în 2009, și doar un an mai târziu a fost făcut un proiect open-source sub licența BSD. În 2013 a devenit parte a Apache Software Foundation, sub licență Apache, iar în 2014 era deja printre aplicațiile de top ale acestei organizații. În același an, compania Databricks, sub conducerea lui Zaharia, a înscris un nou record mondial folosind Spark în sortarea Big Data. [29]

Competiția consta în sortarea a 100 *TB* de date, sub forma a unui bilion de entități de câte 100 de bytes. Rulând pe 207 mașini virtuale EC2, Spark a sortat toate datele în 23 de minute. Comparativ, recordul anterior era stabilit de un framework-ul distribuit de top la ora aceea, Hadoop MapReduce, cu o sortare în 72 de minute folosind 2100 mașini fizice. Dat fiind acest rezultat, Spark este acum considerat succesorul lui Hadoop, totuși fără să îl înlocuiască pe deplin pe acesta. De asemenea, deși nu există o competiție oficială de sortare în *PB*, Databricks a sortat 1 *PB* de date(10 trilioane de entități) în sub 4 ore pe 190 de mașini EC2, în timp ce Hadoop a reușit să le sorteze 16 ore pe 3800 de mașîni.[30]

Întrebarea naturală este cum reușește Apache Spark să ofere rezultate cu atât de mult mai rapide ? De obicei cele mai “scumpe” operații în privința timpului și a resurselor sunt operațiile de citire și scriere. Spark reușește să minimizeze dățile în care aceste operații sunt executate, procesând cât se poate de multe date în memorie. De asemenea, Databricks vine cu un mediu specializat de rulare Spark în cloud și oferă optimizări speciale pentru scrierea și citirea din fișiere prin runtime-ul DBIO. Acest motor oferă un timp de execuție până la de 5 ori mai bun comparativ cu Apache Spark independent în cloud-ul AWS. [31]

Datorită acestor performanțe de viteză superioare, Apache Spark în mediul Databricks a devenit o alegere potrivită aplicației noastre Big Data. Aceste tehnologii ne asigură scalabilitatea aplicației pentru a proce și mai multe date, ceea ce în timp devine o nevoie dat fiind că baza de date MAG este actualizată săptămânal cu aproximativ 5-10 *GB* de date.

### Scala

O alegere dificilă de făcut pentru proiectul nostru a fost selectarea limbajului de programare folosit. Cum am specificat și mai înainte în lucrare, limbajul ales de noi a fost Scala. În continuare vom prezenta de ce am ales acest limbaj în comparație cu variantele disponibile.

Apache Spark oferă API-uri pentru Java, Python, Scala și R [29], ceea ce ne-a lăsat cu o gamă variată de unde putem alege. Am dorit să folosim un limbaj cât mai expresiv, care să ne crească productivitatea. De asemenea, a fost de dorit ca limbajul să aibă destulă funcționalitate deja implementă pentru a putea preprocesa fișierele text cu ușurință: ne referim, de exemplu, la operațiile de segmentare a liniilor din fișierele MAG după caracterul *tab*.

Python, Scala și R sunt mai expresive și mai productive decât prima, dar ultimul din ele e orientat mai mult spre analiză statistică. L-am exclus pe acesta dintre posibilități datorită faptului că am optat pentru un limbaj de programare mai general, nu orientat spre o nișă, pentru a putea rezolva în el orice probleme apar pe parcurs, fără a mai schimba limbajul. Această caracteristică s-a dovedit utilă atunci când am scris modulul de comunicare cu API-ul Microsoft Knowledge. Scrierea de request-uri HTTP într-un limbaj nișat precum R ar fi fost mult mai anevoioasă. Într-adevăr, modulul menționat nu a fost folosit mai departe în dezvoltarea proiectului, dar acesta servește drept un exemplu al problemelor care pot apărea pe parcurs și care pot fi rezolvate mult mai ușor într-un limbaj de programare general.

Cu aceste două opțiuni date la o parte, am rămas cu Scala și Python: amândouă expresive, puternice și productive. Inițial, datorită faptului că am avut mai multă experiență cu limbajul Python, acesta a fost alegerea noastră principală. Aveam disponibilă și o librărie prin care ne puteam conecta la sistemul de fișiere din Azure Data Lake Store și încărca, descărca, citi și scrie fișiere printr-o interfață facilă, foarte asemănătoare cu cea prin care Python comunică cu sistemele de fișiere în general. Deși nu a fost folosită în acest proiect, recomandăm utilizarea ei pentru script-uri simple de descărcare locală sau încărcare a fișierelor, în cazul în care se dorește automatizarea acestei sarcini.

În ciuda acestor avantaje, am descoperit câteva beneficii de care doar limbajul Scala dispunea, ceea ce a dus la alegerea noastră finală. În primul rând, Apache Spark este un framework scris chiar el în Scala, ceea ce are următoarele două mari implicații.

În primul rând, API-ul Scala este mai rapid decât cel de Python. Deoarece nu este nevoie de o librărie de tip *binding* care să facă legătura între două limbaje de programare diferite, framework-ul are mai puține lucruri de făcut pentru a executa cererile noastre, de unde timpii de executare sunt mai mici.

În al doilea rând, lipsa librăriilor *binding* înseamnă și faptul că nu există librării care să trebuiască întreținute. Altfel spus, știm sigur că ultima versiune de Scala va fi mereu disponibilă API-ului Scala, în timp ce celelalte limbaje de programare ar putea avea câteva întârzieri. De exemplu, la momentul actual, Dataset-urile din Spark, structura de date principală în proiectul nostru, nu sunt disponibile în Python.

Astfel, toți acești factori ne-au făcut să alegem Scala, un limbaj care ne-a oferit productivitate, expresivitate, flexibilitate și a maximizat numărul de funcționalități Spark de care dispunem.

[@cite : spark sql și alte surse ale limbajelor]

### Microsoft Azure

Dat fiind mediul Big Data al aplicației noastre și cantitatea mare de date pe care o pune la dispoziție MAG, am realizat repede că executarea pe un singur calculator a proiectului ar fi foarte dificilă. De aceea, am căutat o modalitate de a accesa un cluster și a-i folosi resursele.

Prima noastră opțiune, și cea pe care am ajuns să o folosim, a fost programul Microsoft Azure4Research Award. Prin acest program, Microsoft răsplătește pe cei care vor să facă cercetare în domeniile Big Data sau Inteligență Artificială cu resurse în cloud-ul Azure pentru a fi folosite timp de un an în proiectul lor. [24] Am aplicat la acest program cu o propunere în care ne prezentăm intențiile și cerințele tehnice de care avem nevoie (spațiu de stocare, putere de procesare, servicii Azure), pe baza cărora echipa Microsoft poate să accepte sau nu propunerea și estimează numărul de credite primite pentru a putea cumpăra resurse Azure.

Faptul că proiectul nostru a fost acceptat de acest program a însemnat foarte mult, în primul rând pentru că asta e ceea ce l-a făcut posibil. Prin Azure, avem acces la un putere de procesare și stocare care altfel nu ne-ar fi fost la îndemână.

### Databricks

Pentru a rula Spark în cloud am avut mai multe opțiuni: Databricks, HDInsight și containere Kubernetes. Ele sunt enumerate aici în ordinea crescătoare a dificultății de configurare a serviciului. Kubernetes oferă o libertate foarte mare, dar și complică lucrurile dat fiind că ar fi trebuit să configurăm noi întreg cluster-ul pentru Spark. HDInsight oferă Hadoop în cloud, peste care poate fi adăugat Spark. În cazul acestuia, cluster-ul este administrat automat, ceea ce oferă o mult mai ușoară configurare.

Însă ceea ce ne-a făcut să alegem Databricks sunt optimizările pe care le aduce în executarea Spark în cloud. Databricks este o companie creată de Matei Zaharia, unul dintre creatorii principali ai Spark, care oferă *Spark as a Service* companiilor și persoanelor fizice. Folosind Databricks, utilizatorii nu mai trebuie să se îngrijească de administrarea cluster-ului pe care rulează Spark și se pot concentra exclusiv pe procesarea și analizarea datelor. Ca toate produsele Azure, plata pentru serviciile Databricks se face bazată pe cât timp ai utilizat aceste servicii. Altfel spus, doar momentele în care cluster-ul rulează sunt momente în care utilizatorii percep taxe.

De asemenea, Databricks vine cu opțiunea de cluster auto-scalabil. În acest fel, am putut stabili un număr minim și unul maxim de noduri ale cluster-ului, iar acesta va oscila între numărul de noduri pornite în funcție de necesitățile evaluărilor si operațiilor folosite în Spark. Astfel, Databricks oferă un mediu ușor configurabil, rapid și economic de rulare Spark în interiorul cloud-ului.

## Concluzii

## Bibliografie

* Tehnologii folosite
  + Databricks [Performanțele Databricks](https://people.csail.mit.edu/matei/papers/2015/vldb_spark.pdf) [Introducere în Databricks și exemple de notebook-uri](https://docs.azuredatabricks.net/_static/notebooks/azure/gentle-introduction-to-apache-spark-azure.html)
  + Spark [Performanțe](https://opensource.com/business/15/1/apache-spark-new-world-record) [Avantajele Spark față de Hadoop](https://www.quora.com/What-are-resilient-distributed-datasets-RDDs-How-do-they-help-Spark-with-its-awesome-speed) [Documentația Spark](https://spark.apache.org/docs/latest/index.html) [Dataframe, Dataset sau RDD](https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html) [Cum este optimizat Spark SQL și Dataframes](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html) [Proiectul Tungsten](https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html)
  + Azure [Arhitecturi Big Data](https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/big-data) [Tehnici de design pentru aplicații Cloud](https://docs.microsoft.com/en-us/azure/architecture/guide/)

## Referințe

[1] (n.d.). Peer Group Structure and Adolescent Cigarette Smoking: A ... - Jstor. Retrieved June 19, 2018, from <https://www.jstor.org/stable/pdf/2137204.pdf>

[2] (n.d.). Murder by Structure: Dominance Relations and the Social ... - Jstor. Retrieved June 19, 2018, from <https://www.jstor.org/stable/10.1086/597791>

[3] (n.d.). Birds of a Feather: Homophily in Social Networks - Annual Reviews. Retrieved June 17, 2018, from <https://www.annualreviews.org/doi/10.1146/annurev.soc.27.1.415>

[4] (2016, October 27). Homophily in coauthorship networks of East European sociologists .... Retrieved June 17, 2018, from <https://www.nature.com/articles/srep36152>

[5] (2014, June 10). The Matthew effect in empirical data - NCBI - NIH. Retrieved June 17, 2018, from <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4233686/>

[6] (n.d.). Investigating different types of research collaboration ... - Springer Link. Retrieved June 18, 2018, from <https://link.springer.com/article/10.1007/s11192-011-0343-8>

[7] (n.d.). Predicting author h-index using characteristics of the co-author network. Retrieved June 18, 2018, from <https://link.springer.com/article/10.1007/s11192-012-0933-0>

[8] (2016, October 27). Homophily in coauthorship networks of East European sociologists .... Retrieved June 18, 2018, from <https://www.nature.com/articles/srep36152>

[9] (n.d.). Fragmented Romanian Sociology: Growth and Structure ... - NCBI - NIH. Retrieved June 18, 2018, from <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4237370/>

[10] (n.d.). h-index - Wikipedia. Retrieved June 18, 2018, from <https://en.wikipedia.org/wiki/H-index>

[11] (n.d.). Adler , Ewing , Taylor : Citation Statistics - Project Euclid. Retrieved June 17, 2018, from <https://projecteuclid.org/euclid.ss/euclid.ss/1255009002>

[12] (2015, June 5). Microsoft Academic Graph - Microsoft Research. Retrieved June 18, 2018, from <https://www.microsoft.com/en-us/research/project/microsoft-academic-graph/>

[13] (2017, March 27). Academic Knowledge API for Microsoft Academic Graph | Microsoft Docs. Retrieved June 18, 2018, from <https://docs.microsoft.com/en-us/azure/cognitive-services/academic-knowledge/home>

[14] (2017, March 27). Query expression syntax in the Academic Knowledge ... - Microsoft Docs. Retrieved June 17, 2018, from <https://docs.microsoft.com/en-us/azure/cognitive-services/academic-knowledge/queryexpressionsyntax>

[15] (2010, September 1). Microsoft Academic Search - Microsoft Research. Retrieved June 18, 2018, from <https://www.microsoft.com/en-us/research/project/microsoft-academic-search/>

[16] (2018, March 26). Overview of Azure Data Lake Store | Microsoft Docs. Retrieved June 18, 2018, from <https://docs.microsoft.com/en-us/azure/data-lake-store/data-lake-store-overview>

[17] (2018, May 29). Service-to-service authentication: Data Lake Store with Azure Active .... Retrieved June 18, 2018, from <https://docs.microsoft.com/en-us/azure/data-lake-store/data-lake-store-service-to-service-authenticate-using-active-directory>

[18] (n.d.). Microsoft Academic Graph (MAG) Azure Data Lake ... - GitHub Pages. Retrieved June 18, 2018, from <https://microsoftdocs.github.io/MAG/Getting-MAG-On-ADLS.html>

[19] (n.d.). Spark Architecture · Mastering Apache Spark - Jacek Laskowski. Retrieved June 18, 2018, from <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-architecture.html>

[20] (n.d.). RDD Programming Guide - Spark 2.3.1 Documentation - Apache Spark. Retrieved June 18, 2018, from <https://spark.apache.org/docs/latest/rdd-programming-guide.html>

[21] (n.d.). Spark SQL and DataFrames - Spark 2.3.1 ... - Apache Spark. Retrieved June 18, 2018, from <https://spark.apache.org/docs/latest/sql-programming-guide.html>

[22] (2015, April 13). Deep Dive into Spark SQL's Catalyst Optimizer - The Databricks Blog. Retrieved June 18, 2018, from <https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>

[23] (2016, July 14). DataFrames, RDDs, and Datasets: A Tale of Three Apache Spark APIs. Retrieved June 18, 2018, from <https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html>

[24] "An Overview of Microsoft Academic Service (MAS) and Applications ...." Accessed June 19, 2018. <https://www.bibsonomy.org/bibtex/216d3eb50639a33f5763b43837d0c0d53/jaeschke>.

[25] (n.d.). Markov property - Wikipedia. Retrieved June 18, 2018, from <https://en.wikipedia.org/wiki/Markov_property>

[26] (n.d.). Overview - GraphFrames 0.5.0 Documentation. Retrieved June 18, 2018, from <https://graphframes.github.io/>

[27] (n.d.). Siena Homepage - Oxford University Statistics - University of Oxford. Retrieved June 18, 2018, from <https://www.stats.ox.ac.uk/~snijders/siena/>

[28] (n.d.). Publications - Gephi. Retrieved June 18, 2018, from <https://gephi.org/users/publications/>

[29] (n.d.). Apache Spark - Wikipedia. Retrieved June 18, 2018, from <https://en.wikipedia.org/wiki/Apache_Spark>

[30] (2015, January 15). Apache Spark: 100 terabytes (TB) of data sorted in ... - Opensource.com. Retrieved June 18, 2018, from <https://opensource.com/business/15/1/apache-spark-new-world-record>

[31] (n.d.). Databricks Runtime - Databricks. Retrieved June 18, 2018, from <https://databricks.com/product/databricks-runtime>