Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Um estudo de integração de dados heterogêneos

João Paulo Ferreira Rodrigues

JUIZ DE FORA
DEZEMBRO, 2018

Um estudo de integração de dados heterogêneos

João Paulo Ferreira Rodrigues

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Victor Ströele de Andrade Menezes

JUIZ DE FORA

DEZEMBRO, 2018

Um estudo de integração de dados heterogêneos

João Paulo Ferreira Rodrigues

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Victor Ströele de Andrade Menezes
Professor Doutor

Mario Antonio Ribeiro Dantas
Professor Doutor

Luiz Felipe Carvalho Mendes
Professor Mestre

JUIZ DE FORA
05 DE DEZEMBRO, 2018

Aos pais, pela paciência e apoio.

À Juliana por toda força dada.

Resumo

Com o advento da internet e da era do Big Data, se tornou crucial para as organizações analisar e extrair conhecimento de um extenso volume de dados. Com tudo, o fato de, na maior parte das vezes, os mesmos se originarem de fonte da dados diferentes, torna a análise de dados heterogêneos um grande desafio a ser superado. Foram desenvolvidas diversas pesquisas nessa área, gerando algumas ferramentas capazes de executar essa função, como, o SQRE, o ARGO e o CloudMdsQL. Este trabalho propõe uma arquitetura baseada no Apache Spark para viabilizar a integração de dados e, fazendo uso de uma API, deixar transparente para o usuário a integração das fontes heterogêneas, diferentemente das propostas anteriormente citadas. A solução proposta foi implementada utilizando Spark, e uma API construída em Python, conectando uma base de dados em Neo4J, com as relações entre atores, diretores e filmes, e uma base de dados em PostgreSQL, contendo informações como gastos, faturamentos e popularidade dos filmes. Foi implementado também, na API, métodos com objetivos de analisar os dados e extrair informações utilizando a integração das duas fontes de dados. Para avaliar a proposta, foram conduzidos Cenários de Uso, onde os usuários utilizam do sistema proposto para obter informações sobre atores e filmes. Com a avaliação, verificou-se que a arquitetura proposta apresentou os resultados esperados, se mostrando uma alternativa viável para a tarefa de integração de dados. Além de confirmar o Spark como uma ferramenta poderosa de integração de dados, principalmente devido as abstrações presentes nativamente na mesma. Assim, entendemos que a proposta é uma forma eficiente de integrar e analisar uma variedade abrangente de tipos de dados.

Palavras-chave: Integração de Dados, Spark, Banco de Dados, API, Neo4j, PostgreSQL, Dados Heterogêneos.

Abstract

With the advent of the Internet and Big Data era, it is crucial to analyze and extract knowledge from a large volume of data. However, the fact that, in most cases, they originate from the different data source, makes a heterogeneous data analysis a great challenge to be overcome. Several functions have been adjusted in the area, generating some functions management tools, such as SQRE, ARGO and CloudMdsQL. This work presents a base based on Apache Spark for the feasibility of a data integration, and makes use of an API, allowing the user to integrate heterogeneous sources, unlike the previously mentioned proposals. The proposal was implemented using Spark, and a Python-built API, connecting a Neo4J database with entries, directors and movies, and a PostgreSQL database containing information such as spending, billing, and movies. It has also been implemented in the API for data analysis purposes and to extract information using an integration of the two data sources. To obtain a proposal, Use Scenarios were conducted, where users use the system to obtain information about actors and films. With an evaluation, it was verified that the proposed matrix presented expected results, being a viable alternative for a task of data integration. In addition, we can see as a powerful data integration tool mainly due to abstractions present natively in it. Thus, we understand that it is an efficient way to integrate and analyze a variety of data types.

Keywords: Data Integration, Spark ,Database, API, Neo4j, PostgreSQL, Heterogeneous Data.

Agradecimentos

Primeiramente à Deus, que em toda minha jornada se fez presente e iluminou meus caminhos.

Aos meus pais, Lacir e Simone, por todo carinho, amor e compreensão, por serem meu porto seguro e por me darem a oportunidade de realizar os meus sonhos.

A minha irmã, Gabriela, que mesmo com as implicâncias de uma irmã mais nova, sempre me fez sorrir.

A minha namorada e melhor amiga Juliana, cuja todas as palavras do universo não conseguem descrever a importância dela nessa conquista. Agradeço a todos esses anos ao meu lado e por ser a luz na escuridão dos momentos difíceis.

Ao meu sogro e minha sogra, por me receberem tão bem em sua família e, me considerando como um filho, se tornando minha família em terras juizforanas.

Aos todos meus amigos pelos momentos de felicidade e descontração, em especial, Bruno, Dennis e Rafael, que estiveram desde o princípio dessa jornada ao meu lado.

Às minhas avós, Dodora e Nazaré, e ao meu avô Geraldo, por todas as orações e apoio recebidos.

A minha madrinha Néria e minhas bisavós Geralda e Conceição (*in memorian*), por serem meus anjos da guarda e cuidarem de mim lá de cima.

Aos meus tios, primos, e familiares, que sempre estiveram do meu lado e, mesmo distantes, contribuíram para conclusão dessa etapa. Em especial ao meu primo e padrinho Rafael, por ser o principal apoiador, e culpado, por seguir na área de exatas.

Ao professor Victor Stroele pela orientação, paciência e comprometimento a esse trabalho, o qual, a sua conclusão, deve-se muito a ele. Gostaria de agradecer também suas aulas de Banco de Dados, que fizeram definir minha carreira.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"Quando vocês souberem qual é exatamente a pergunta, vocês saberão o que significa a resposta.".

Douglas Adams (O Guia do Mochileiro das Galáxias)

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

DCC	Departamento de Ciência da Computução
UFJF	Universidade Federal de Juiz de Fora
SGBD	Sistema de gerenciamento de banco de dados
BDOG	Banco de dados orientado a grafos
RDD	Dados Distribuídos Resilientes
UC	Universidade da Califórnia
REST	Representational State Transfer
API	Interfaces de Programação de Aplicativos
SQRE	Scalable Query Rewriting Engine
JSON	JavaScript Object Notation

1 Introdução

Diariamente são gerados cerca de 2,5 quintilhões de bytes, equivalentes a 2,5 bilhões de gigabytes, e em sua extensa maioria, estima-se 90% são de dados não estruturados (SIVARAJAH et al., 2017). A IBM projeta para os próximos 2 anos um aumento de 200% sobre este valor (LAU et al., 2016) e Gantz e Reinsel (2012) afirmam que em 2020 esse número chegará a 40 Zettabytes, o que corresponde a 40 trilhões de gigabytes.

Uma grande dificuldade enfrentada pelas organizações, ao trabalhar com uma extensa quantidade de dados, é o fato dos mesmos serem originados a partir de fontes heterogêneas e independentes entre si (HALEVY; RAJARAMAN; ORDILLE, 2006). O desafio de integração de dados é frequentemente enfrentado por diversos tipos de aplicações, como no progresso de pesquisas cientificas com a unificação de dados gerados por diversos pesquisadores autônomos (HALEVY; RAJARAMAN; ORDILLE, 2006) e na descoberta de novos medicamentos, integrando resultados de diferentes pesquisas e áreas, possibilitando a obtenção de novos conhecimentos, como relatado pelo Searls (2005).

Em 2009, surge na Universidade da Califórnia um projeto com a finalidade de criar um mecanismo para unificar o processamento de dados distribuídos, denominado Apache Spark (ZAHARIA et al., 2016). O modelo de programação do Spark se baseia no paradigma do MapReduce, todavia, adiciona o conceito de "Conjuntos de Dados Distribuídos Resilientes"(RDDs), possibilitando o compartilhamento de dados e a junção de diferentes fontes, como Hadoop, MySql, Cassandra, dentre outros (ZAHARIA et al., 2016).

1.1 Problema

O Big Data, nos últimos anos, se tornou um conceito essencial para as organizações. A necessidade de melhoria em tomadas de decisões, desenvolvimento de produtos, gerenciamento da produção e marketing direcionado, trouxe o uso de dados como base para obter vantagens competitivas no mercado.

Porém, como apresentado anteriormente, a predominância de dados heterogêneos e de fontes independentes dificultam a implementação de soluções baseadas em Big Data. Aplicações como redes sociais, serviços de armazenamento em nuvem, sistemas de gerenciamento de empresas, exigem não somente uma grande capacidade de armazenar dados, mas também uma facilidade em correlacionar e processar as informações para auxiliar a tomada de decisão através dos conhecimentos obtidos, como mostrado na Figura 1.1.

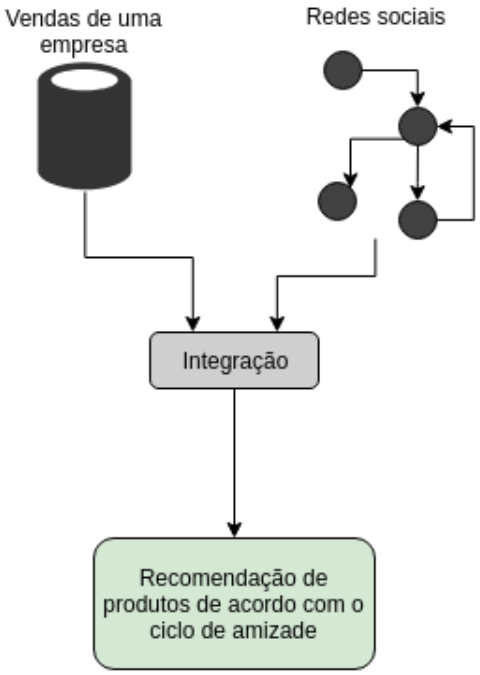

Figura 1.1: Exemplo de um cenário de integração de dados

O problema a ser tratado neste trabalho é: *Como integrar fontes de dados heterogêneas, de forma transparente para o usuário, permitindo que ele tenha acesso a informações consolidadas de diferentes repositórios de dados?*

1.2 Justificativa

Com o aumento da demanda por análise de dados e o crescimento de dados cada vez mais heterogêneos, novas ferramentas vem surgindo com o objetivo de, com a integração de dados, gerar mais conhecimento para seus usuários.

Algumas soluções em Bancos de Dados, buscam extrair dados de fontes diversas e centralizá-los em um único modelo, de forma que as consultas possam ser feitas de

maneira centralizada. Porém, essa solução exige que a rotina de extração dos dados seja executada periodicamente para manter os dados atualizados no modelo centralizado.

Neste sentido, são necessárias soluções que acessem os dados em suas fontes de origem e faça a integração dos mesmos em tempo de execução, sem a necessidade de rotinas que mantenham os dados atualizados. O Hadoop[1] e o Apache Spark[2], surgem como possíveis soluções para resolução desses problemas.

O Hadoop, por exemplo, oferece um sistema de baixo nível, o que o torna uma poderosa ferramenta para análise de dados, por possibilitar adaptabilidade ao cenário de uso e a programação de quaisquer processamentos requeridos pelo programador. Porém, o foco desse tipo de instrumentos é o armazenamento de dados, tornando a análise, muitas vezes, não tão eficiente.

O Apache Spark, por sua vez, é uma solução focada no processamento de dados, com o diferencial de ter várias funções nativas, como SparkSQl e uma biblioteca de aprendizado de máquinas. Além disso, ele carrega os dados em memória previamente, tornando a análise muito mais rápida e eficiente.

1.3 Objetivos

Considerando o exposto acima e com o propósito de auxiliar a integração de dados heterogêneos, este trabalho busca implementar uma API REST baseada no Apache Spark, com o objetivo de abstrair a coleta de informações através de dados obtidos em fontes independentes e heterogêneas.

Como objetivos específicos, podemos citar:

- Configurar um servidor utilizando o Apache Spark;
- Implementar uma API REST utilizando a linguagem Python;
- Configurar a API para trabalhar com o Spark;
- Conectar a API, utilizando o Spark, às fontes de dados;
- Implementar os métodos, que farão as consultas, a partir da API;

[1] https://hadoop.apache.org/
[2] https://spark.apache.org/

1.4 Organização do trabalho

Este texto segue a seguinte organização. O Capítulo 2, apresenta os conceitos relacionados à proposta deste trabalho, como Big Data, Ciência de dados, Spark, bancos de dados orientados a grafos e API REST. O Capítulo 3 apresenta os trabalhos relacionados à integração de dados, utilizando outras abordagens e ferramentas diferentes deste trabalho. O Capítulo 4 descreve a arquitetura proposta para a integração de dados heterogêneos e uma implementação baseada nessa arquitetura. O Capítulo 5 apresenta cenários de utilização da API criada e a análise dos resultados obtidos nesses cenários. E por ultimo, o Capítulo 6 expõe as considerações finais deste trabalho.

2 Fundamentação teórica

Esse capítulo apresenta alguns conceitos fundamentais sobre os quais o trabalho será embasado, onde será feita uma revisão da literatura sobre a teoria.

Serão abordados, nas próximas seções, conceitos como: Big Data, Spark e suas características, banco de dados orientado a grafos e API REST.

2.1 Ciência de dados

Com o avanço acelerado de novas tecnologias de processamento e armazenamento de dados, as empresas enxergaram uma oportunidade de usá-los para tomarem decisões mais assertivas ao longo do tempo. Porém, surgiu um novo problema, como analisar e extrair conhecimento de um grande conjuntos de dados (SONG; ZHU, 2016).

Ao longo dos anos, houve um considerável aumento no volume de dados produzidos e armazenados pela humanidade. Esse fenômeno, denominado de dilúvio de dados (SIVARAJAH et al., 2017), deu origem a era do *Big Data*, termo utilizado para representar os desafios gerados por essa extensa quantidade. Somente o grande volume não define *Big Data*, é necessário que os dados obedeçam certas características, batizadas de 5V's. São elas (SONG; ZHU, 2016):

- **Volume**: Quantidade de dados em grande escala;
- **Velocidade**: Rapidez na criação, processamento, análise e armazenamento de dados;
- **Variedade**: Fontes, tipos e formas de manipulação de dados heterogêneas;
- **Veracidade**: Dados confiáveis, de qualidade e incertezas nos dados;
- **Valor**: Retorno do investimento obtido pela descoberta de conhecimento analisando os dados;

Apesar da projeção que esse mercado movimentará US$ 46,34 bilhões até 2018, 85% da *Fortune 500 Organizations* não obterão vantagens competitivas explorando *Big Data* (SONG; ZHU, 2016). As empresas ainda encontram uma grande dificuldade devida a heterogeneidade das fontes de dados utilizadas. A citar; sensores, serviços web, comércios eletrônicos e outros (LAU et al., 2016), que geram dados heterogêneos contendo desde textos a conteúdos multimídias (SIVARAJAH et al., 2017).

Para solucionar esta nova adversidade, surgiu a Ciência de Dados. Substancialmente, essa ciência estuda conjuntos de princípios de extração e conhecimento de dados (PROVOST; FAWCETT, 2013). Entretanto, segundo Song e Zhu (2016), vários autores a definem de maneira diferente, sendo, ainda de acordo com esses autores, a apresentada por Dhar (2013) - "Ciência de dados é o estudo da extração generalizável de conhecimento a partir de dados" - a mais indicada.

A Ciência de Dados é pautada em três pilares: dados, tecnologia e pessoas. Os dados são representados por: dados estruturados, como banco de dados relacionais e não relacionais, quanto dados não estruturados, como planilhas e dados de redes sociais. A tecnologia refere-se as ferramentas usadas, como Sistemas de Gerenciamento de Bancos de Dados (SGBD), inteligência computacional e computação em nuvem. Já as pessoas são os envolvidos diretamente no tratamento e uso dos dados como: cientistas da computação, estatísticos e analistas de negocio (SONG; ZHU, 2016).

A aplicabilidade dessa ciência é vasta e amplamente utilizada por empresas de diversos setores. Empresas de marketing utilizam essas ferramentas para direcionamento de publicidade, varejistas usam para analisar comportamento de clientes e gerenciamento de cadeia de suprimentos e empresas do ramo financeiro atribui classificações de crédito e detectam fraudes usando esses instrumentos, por exemplo (PROVOST; FAWCETT, 2013).

2.2 Apache Spark

Um grupo de pesquisa da Universidade da Califórnia (UC), em Berkeley, identificou a necessidade de usuários de sistemas de arquivos distribuídos, mais especificamente o Hadoop, em executar aplicações mais complexas na manipulação dos dados. Porem, o MapReduce,

paradigma usado no Hadoop, apresentava limitações que impediam esses tipo de utilização (ZAHARIA et al., 2012).

O Spark foi criado, por esse grupo, com a proposta de estender o MapReduce facilitando o compartilhamento de informações e dados entre operações paralelas referentes as suas etapas. Os sistemas de arquivos distribuídos, para realizar esses compartilhamentos, precisam escrever os dados em um sistemas de arquivos, adicionando assim um elevado custo devido a replicação de dados em disco e também velocidade de gravação em discos rígidos que não possuem o melhor desempenho (principalmente que não são discos como SSD). O Spark, para transpor essa deficiência, propõe uma abstração a qual os dados são, através de consultas, armazenados na memória e com tolerância a falhas, pois possibilita obter os dados perdidos através dos dados base no disco. Com isso, o Spark viabiliza aplicações mais rápidas, diminuindo em até 40x o tempo de leitura e gravação dos dados, comparando com os sistemas de arquivos distribuídos (ZAHARIA et al., 2012).

Outras vantagens, geralmente destacadas, dessa ferramenta são: a facilidade de programação devido a possibilidade de programar em Python, Java, Scala e R (SHANAHAN; DAI, 2015) e ter diversos tipos de componentes já integrados ao seu núcleo, facilitando ainda mais o desenvolvimento de aplicações, são eles (ZAHARIA et al., 2016):

- SparkSQL: Disponibiliza o uso de SQL para análise e consultas de dados no Spark;
- GraphX: Possibilita o uso de processamento dos dados em grafos;
- MLlib: Biblioteca com diversos algoritmos de aprendizado de máquina já implementados;
- Spark Streamming: Permite a análise do fluxo de dados em tempo real;

2.2.1 MapReduce

Usado para processar um grande volume de dados, o MapReduce é um modelo de implementação bastante utilizado visando: a paralelização em clusters, tratamento de falhas e otimização do uso de discos e rede. Em um artigo, Dean e Ghemawat (2008), afirmam que devido a facilidade desse modelo, mais de 10 mil aplicações já foram implementadas

pelo Google utilizando-o. Esse modelo divide o processamento em duas etapas: Map e Reduce.

A etapa de Map é responsável por enviar o código a ser executado para os diversos nós, onde cada nó executara e responderá um conjunto de tuplas de chave e valor referentes aquele programa enviado.

A etapa de Reduce recebe o conjunto de chave e valor vindo da etapa de Map e agrupa os resultados de todos os nós pela chave de cada tupla. Como mostra a Figura 2.1.

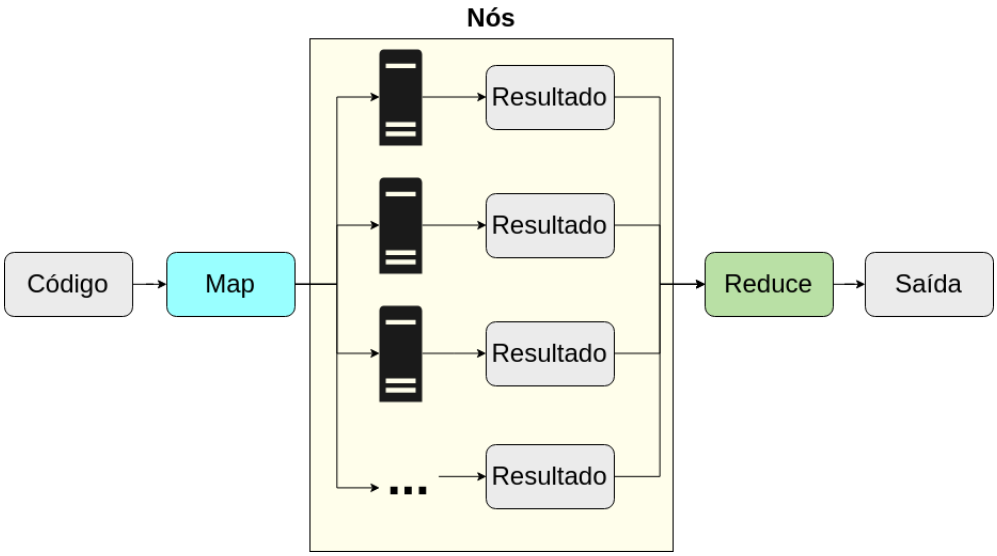

Figura 2.1: Funcionamento do MapReduce

2.2.2 Conjunto de dados distribuído resiliente (RDD)

Uma das principais diferenças entre o Spark e os Sistemas de Arquivos Distribuídos, são os RDDs. Essa abstração representa um conjunto de objetos que, de maneira distribuída, estão presentes em vários clusteres e são facilmente recriados caso uma partição seja perdida (ZAHARIA et al., 2010).

Os elementos desse conjunto não tem a necessidade de estarem armazenados fisicamente. Um *handle* de um RDD possui informações para computá-lo usando dados em armazenamentos confiáveis, ou seja, mesmo se alguns nós falharem, é possível reconstruí-los (ZAHARIA et al., 2010). Assim, como mostra a Figura 2.2, o RDD contém uma coleção, geralmente armazenada em memória, de objetos originários de vários nós de

maneira distribuída e os códigos para que possam ser gerados novamente em caso de falhas.

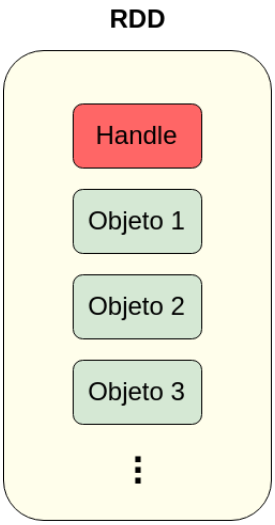

Figura 2.2: Arquitetura de um RDD

Os RDDs, podem ser construídos de quatro formas diferentes: a partir de um sistemas de arquivos compartilhados; paralelizando uma coleção de dados, dividindo-a em várias partes e enviando-as para os vários nós; através de um outro RDD existente, executando uma transformação de RDD A em um RDD B por meio de um função definida pelo programador; ou alterando a persistência de um RDD existente (ZAHARIA et al., 2010).

O padrão do RDD é as partições serem criadas sob demanda e descartadas após o uso, porém o programador pode alterar a persistência do conjunto por meio de *Cache*, fazendo que o RDD não seja excluído após o uso para ser reutilizado depois, ou salvando os dados em um sistemas de arquivos distribuídos, para serem utilizados por futuras operações (ZAHARIA et al., 2010).

2.3 Banco de Dados Orientado a Grafos

O surgimento do modelo relacional, com sua simplicidade e poder expressivo, proporcionou um grande avanço para pesquisas e desenvolvimento relacionado à banco de dados, tornando-o amplamente utilizado. Porém, com o passar do tempo e surgimento de novas demandas, uma grande quantidade de novos formalismos foram propostos. Uma dessas

demandas foi a necessidade de representar estruturas hierárquicas e a partir dela foi criado o banco de dados orientado a grafos (BDOG) (TUIJN; GYSSENS, 1996).

Com a popularização das redes sociais e serviços de nuvem novos paradigmas relacionadas a banco de dados vem ganhando popularidade. Os bancos NoSQL, apelidados de "não somente SQL", oferecem melhores desempenhos e escalabilidade em determinados cenários devido a sua estrutura não relacional, com tudo, para alcançar essas melhorias, muitas vezes sacrificam a disponibilidade, a tolerância à partição de dados ou a consistência. Os BDOG ganharam força entre os bancos NoSQL por oferecerem uma linguagem mais adequada para consultas, diferentemente de outros bancos que possuem armazenamento de chave-valor (HOLZSCHUHER; PEINL, 2016).

A principal diferença entre esse tipo de banco com os bancos de dados relacionais se dá ao fato de que, nos BDOG a topologia dos dados carregam informação, tornando-a tão relevante quanto o próprio dado. Sendo assim, tipicamente as consultas são por navegação entre os nós do grafo avaliando se percorrem um caminho que satisfaz determinadas propriedades especificas (REUTTER; ROMERO; VARDI, 2017).

Os dados nas redes sociais, especialmente, são extremamente interconectados, como rede de pessoas, tags, atividades e comentários. Esse cenário geraria muitas relações muitos para muitos em um modelo relacional, onde as junções necessárias para manipular esses dados, seriam bastante complexas e de baixo desempenho. Todavia, essas relacões são facilmente expressas em um BDOG, que são projetados visando justamente atender esses cenários, além de entregar um alto desempenho ao manipulá-lo (HOLZSCHUHER; PEINL, 2016).

2.4 API REST

O termo *Representational State Transfer* (REST) foi definido por Roy Fielding para categorizar uma arquitetura de software que, por sua vez, são um conjunto de regras a serem seguidas para que o software apresente as características estabelecidas pelo estilo definido. Baseado em outras 12 arquiteturas diferentes, o REST, visa oferecer maior usabilidade, simplicidade, escalabilidade e extensibilidade aos sistemas distribuídos de hipermídia. (LI; CHOU, 2011).

Os Web Services, são servidores especificamente criados para atender uma aplicação, onde, as mesmas, utilizam de interfaces de programação de aplicativos (APIs) para se comunicar com esses serviços. Desta forma, as API's, possibilitam as interações entre programas, permitindo maior facilidade em trocar informações (MASSE, 2011).

O REST se estabeleceu como um estilo arquitetural comum em API's. Uma API REST provem a interconexão de recursos com organização e melhor usabilidade, tornando o serviços mais atrativo para o uso dos desenvolvedores que desejam consumir alguma API (MASSE, 2011).

3 Trabalhos relacionados

3.1 SQRE

A necessidade de integrar e analisar dados de fontes heterogêneas tornou esse tema bastante recorrente em pesquisas relacionada à Big Data. Sendo assim, outras abordagens já foram propostas por outros pesquisadores, utilizando diferentes ferramentas para solucionar este problema.

Em sua pesquisa, Hai, Quix e Zhou (2018), descrevem a dificuldade de pesquisar em Data Lakes (repositório onde os dados brutos são inseridos em seu formato original), por não oferecerem um esquema unificado para os dados. Motivados pela complexidade de analisar dados de atores contidos em um banco baseado em grafos (Neo4J) e informações sobre filmes e elencos presentes em um banco MongoDB, propuseram um sistema, utilizando o Spark, que traduz a consulta em diferentes linguagens, como JSONiq e SQL, em uma "representação lógica independente dos sistemas subjacentes". Ao final do trabalho, foi apresentado o SQRE (*Scalable Query Rewriting Engine*), apresentado na Figura 3.1, que traduz consultas em JSONiq para consultas suportadas em vários sistemas relacionais e NoSQL, facilitando pesquisas em dados heterogêneos mantendo a escalabilidade das mesmas. Porém não oferece uma maior abstração visando o usuário final.

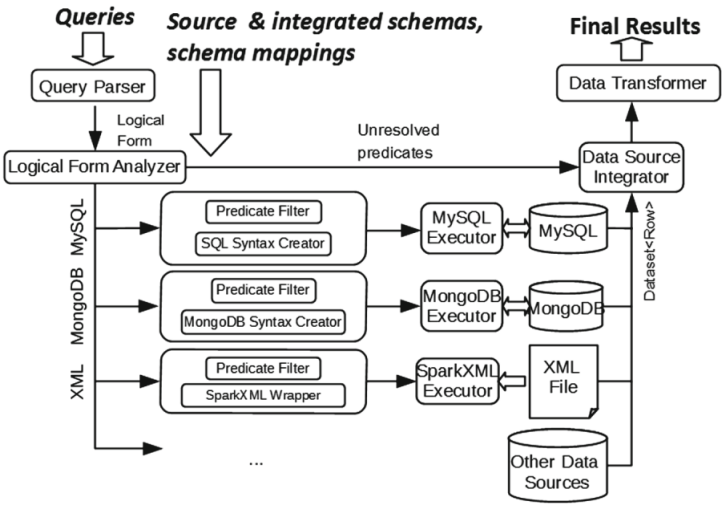

Figura 3.1: Representação do SQRE
Fonte: Hai, Quix e Zhou (2018)

3.2 ARGO

Com a ideia de usar sistemas de chave-valor em bancos relacionais, Chasseur, Li e Patel (2013) propuseram o ARGO. Com as transformações dos bancos relacionais no modelo de chave e valor, torna-se possível a integração dos dados de bancos NoSQL e SQL e facilita a análise, pois a consulta de chave-valor é relativamente mais simples. Porém, há uma custo alto de processamento nessa transformação o que torna essa alternativa inviável para analisar grandes volumes de dados.

3.3 CloudMdsQL

Outra abordagem para esse problema é o CloudMdsQL, proposto por Kolev et al. (2016). Consiste em um sistema baseado na nuvem que possibilita a integração de diversos tipos de fontes de dados e unifica a consulta em uma linguagem que constrói uma consulta através de sub-consultas das linguagens especificas de cada fonte, como no exemplo da Figura 3.2. Toda via, essa ferramenta tem como objetivo explicitar a heterogeneidade dos dados, portanto não funciona como uma abstração para várias fontes de dados, fazendo que suas consultas sejam mais complexas.

CloudMdsQL query

```
T1(x int, y int)@db1 = (
  SELECT x, y FROM A
)
T2(x int, z string)@db2 = {*
  db.B.find( {$lt: {x, 10}},
          {x:1, z:1, _id:0} )
*}
SELECT T1.x, T2.z
FROM T1, T2
WHERE T1.x = T2.x
  AND T1.y <= 3
```

Figura 3.2: Representação de uma consulta no CloudMdsQL
Fonte: Kolev et al. (2016)

3.4 JEN

Um outro trabalho relacionado à integração de banco de dados foi apresentado por Tian et al. (2015). O JEN é capaz de unir sistemas de arquivos distribuídos com banco de dados relacionais, permitindo a junção dos dados das duas fontes. Apesar de empregar técnicas para minimizar o fluxo de dados e ter um bom desempenho quando as junções são feitas no lado do sistema de arquivos distribuídos, essa solução se diferencia da proposta neste trabalho por apresentar limitações na diversidade de fontes de dados possíveis de serem conectados.

Após análise da literatura, a Tabela 3.1 foi criada para exibir as diferenças entre os trabalhos relacionados e o trabalho proposto.

Tabela 3.1: Diferenças entre os trabalhos relacionados e o trabalho proposto

Trabalhos	Uso do Spark	Diversidade de fontes	Abstrações para o usuário	Escalabilidade
SQRE	X	X		X
ARGO		X		
CloudMdsQL		X		X
JEN				X
Proposta	X	X	X	X

4 Proposta

Este capítulo descreve o conceito da proposta apresentada, os passos que foram seguidos para instalar as ferramentas necessárias e implementar a API, sendo essa a meta deste trabalho.

A arquitetura proposta para a integração de dados explicitadas neste trabalho tem como pontos positivos: o acesso aos dados sempre atualizados, interface amigável e independente de linguagem de programação, a possibilidade de conectar e integrar diversos tipos de fontes de dados e a escalabilidade por utilizar o Spark.

4.1 Arquitetura conceitual

Com o intuito de executar consultas em bases de dados heterogêneas, este trabalho propõe uma arquitetura utilizando Apache Spark. O uso dessa tecnologia possibilita que os dados sejam integrados em tempo de execução e, com isso, tem-se a garantia de que eles estão sempre atualizados. O Apache Spark tem como característica se conectar à fonte de dados heterogêneas simultaneamente. Na estrutura, por ele utilizada, o Spark se conecta nas fontes de dados e fornece para o usuário uma interface para interação, podendo essa ser em Scala, Python, Java ou R, como mostra a Figura 4.1. Contudo essa interface, muitas vezes, não é amigável. O fato de ser uma ferramenta com tantas possibilidades, tornam-se complexas até tarefas mais simples.

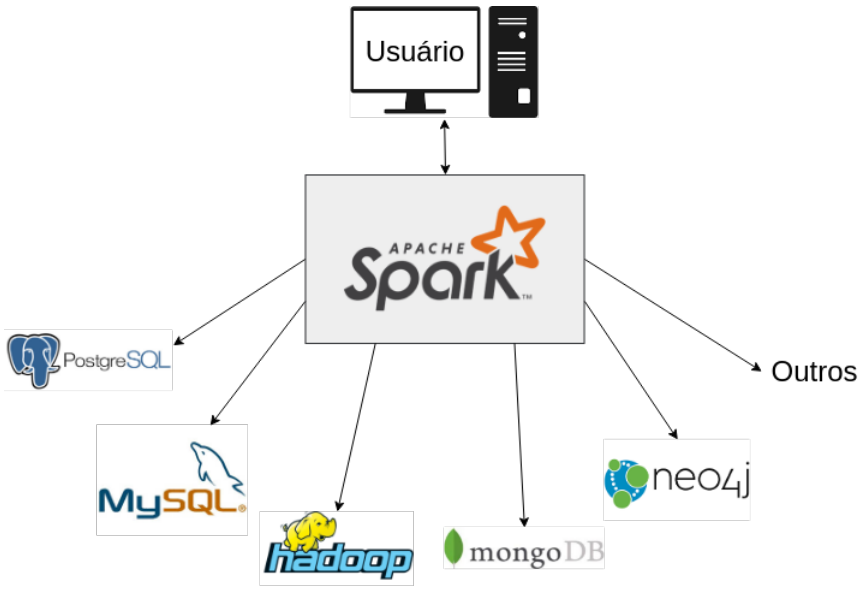

Figura 4.1: Representação da estrutura do Spark

Buscando simplificar o acesso aos dados, esse trabalho visou a criação de uma API *open source*, executada dentro do Spark (Figura 4.2), com o objetivo de deixar mais transparente, para o usuário final, as consultas que são previamente implementadas. Com a API, a introdução de novas fontes de dados ficou mais acessível, assim como a criação de novas buscas, facilitando o acesso a essa ferramenta. Por ter sido disponibilizada com a licença de código aberto, a comunidade e outros pesquisadores podem reutilizar a API, adicionando novos dados e métodos.

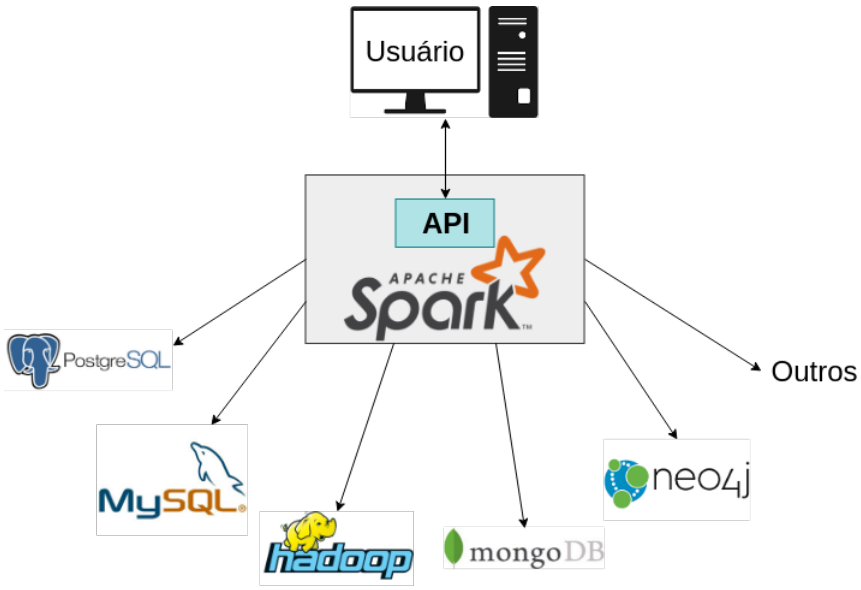

Figura 4.2: Representação da estrutura proposta

Para a integração das informações, cada fonte de dados foi conectada ao Spark que, por sua vez, realizou as devidas consultas em cada fonte. Os resultados dessas consultas são retornados na forma de um RDD. Essas RDD foram transformadas em *dataframes*, que são uma abstração em formato de tabela disponível no SparkSQL. Tendo essas tabelas, foi possível uni-las através dos ids, iguais em todas as tabelas (Figura 4.3).

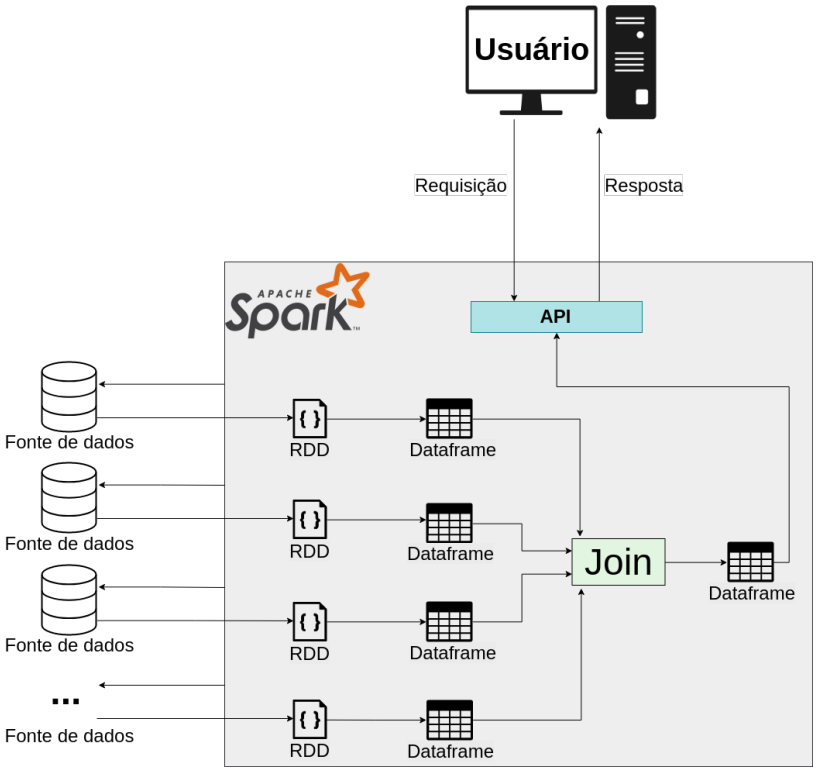

Figura 4.3: Representação da proposta deste trabalho

4.2 Desenvolvimento da solução

Para o desenvolvimento deste trabalho, foi utilizado uma máquina com o processador Intel(R) Core(TM) i5-7600K de 3.80GHz, uma memória de 8Gb DDR4 com frequência de 2400mhz e um HD de 1 Tb de 7200 rpm. O sistema operacional escolhido foi o Linux, mais precisamente a distribuição Ubuntu na versão 18.04.1 LTS com 64 bits.

Foi utilizado também o Python 2.7.15rc1, Java 8, o Neo4j na versão 3.4.10 da Edição Community e o PostgreSQL na versão 10.6 com o PgAdmin4. Foi escolhido o Apache Spark na versão spark-2.3.2-bin-Hadoop2.7, pois já vem com o Hadoop 2.7 como sistema de arquivos. Para acessar as URL's da API, foi utilizado o Google Chrome na

versão 70.0.3538.77 de 64 bits.

Primeiramente, foi instalado, na máquina anteriormente descrita,o Python, o Java e o Neo4J. O banco foi configurado e logo após foi importado à ele o conjunto de dados cineasts_12k_movies_50k_actors[3], que contém relacionamentos entre filmes e atores que atuaram neles, diretores e filmes que eles dirigiram e usuários e os filmes que classificaram, como mostra a Figura 4.4.

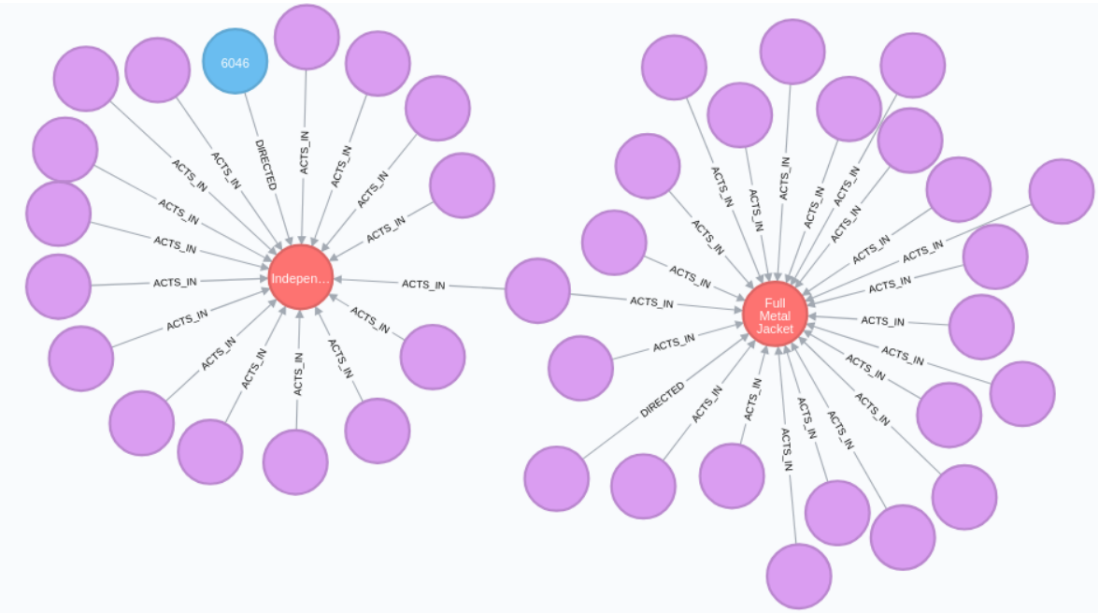

Figura 4.4: Representação dos dados no Neo4J

O PostgreSQL também foi instalado e populado com os dados presentes em movies_metadata.csv[4], contendo informações como: valor arrecadado, avaliação dos usuários, popularidade e gastos de 45465 filmes, como visto na Figura 4.5. Estes foram carregados com o uso da ferramenta *Import* do PgAdmin4 em uma tabela com as mesmas colunas do arquivo em CSV, como mostra a Figura 4.6.

[3]https://neo4j.com/developer/example-data/
[4]https://www.kaggle.com/rounakbanik/the-movies-datasetmovies_metadata.csv

	id character	imdb_id character va	budget character va	original_title character varying	popularity character vary	revenue character va	vote_average character varyi
1	862	tt0114709	30000000	Toy Story	21.946943	373554033	7.7
2	8844	tt0113497	65000000	Jumanji	17.015539	262797249	6.9
3	15602	tt0113228	0	Grumpier Old Men	11.7129	0	6.5
4	31357	tt0114885	16000000	Waiting to Exhale	3.859495	81452156	6.1
5	11862	tt0113041	0	Father of the Bride Par...	8.387519	76578911	5.7
6	949	tt0113277	60000000	Heat	17.924927	187436818	7.7
7	11860	tt0114319	58000000	Sabrina	6.677277	0	6.2
8	45325	tt0112302	0	Tom and Huck	2.561161	0	5.4
9	96357	tt0113276	0	Headless Body in Topl...	0.001346	0	0.0
10	287811	tt0369295	0	Boys Life 4: Four Play	0.066548	0	5.0

Figura 4.5: Representação dos Dados no PostgreSQL

```
CREATE TABLE public.movie_metadata
(
    adult character varying COLLATE pg_catalog."default",
    belongs_to_collection character varying COLLATE pg_catalog."default",
    budget character varying COLLATE pg_catalog."default",
    genres character varying COLLATE pg_catalog."default",
    homepage character varying COLLATE pg_catalog."default",
    id character varying COLLATE pg_catalog."default",
    imdb_id character varying COLLATE pg_catalog."default",
    original_language character varying COLLATE pg_catalog."default",
    original_title character varying COLLATE pg_catalog."default",
    overview character varying COLLATE pg_catalog."default",
    popularity character varying COLLATE pg_catalog."default",
    poster_path character varying COLLATE pg_catalog."default",
    production_companies character varying COLLATE pg_catalog."default",
    production_countries character varying COLLATE pg_catalog."default",
    release_date character varying COLLATE pg_catalog."default",
    revenue character varying COLLATE pg_catalog."default",
    runtime character varying COLLATE pg_catalog."default",
    spoken_languages character varying COLLATE pg_catalog."default",
    status character varying COLLATE pg_catalog."default",
    tagline character varying COLLATE pg_catalog."default",
    title character varying COLLATE pg_catalog."default",
    video character varying COLLATE pg_catalog."default",
    vote_average character varying COLLATE pg_catalog."default",
    vote_count character varying COLLATE pg_catalog."default"
)
```

Figura 4.6: Tabela criada no PostegreSQL

Feito isso, o Spark foi instalado e devidamente configurado e foi feito o download do postgresql-42.2.5.jre6.jar para pasta jar do mesmo. Assim, com todas as ferramentes já disponíveis na máquina, foi criada a API em Python, com o auxilio da biblioteca Flask, onde os métodos com análise desses dados foram criados (Figura 4.7). Ao inicializar a API, a conexão com o Neo4J e o PostgreSQL é estabelecida e a tabela do banco relacional é transformada em um dataframe, como mostrado no código, em Python, na Figura 4.8

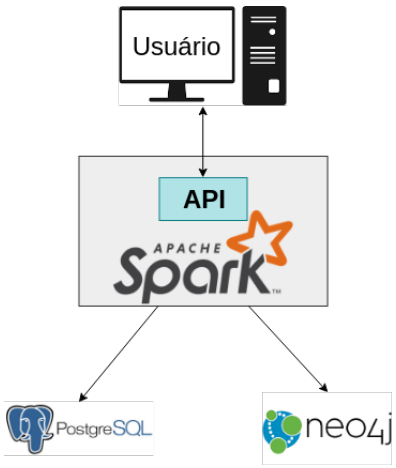

Figura 4.7: Representação da estrutura implementada

```
1  # Conexao com o Neo4J
2  db = GraphDatabase("http://localhost:7474", username="neo4j",
       password = "admin")
3
4  # Conexao com o PostgreSQL
5  sc = SparkContext(appName = "movies")
6  sqlContext = sql.SQLContext(sc)
7  url = 'postgresql://localhost:5432/postgres'
8
9  # Criacao do DataFrame com os Dados da tabela movie_metadata
10 properties = {'user': 'postgres', 'password': 'postgres'}
11 dfpostgres = DataFrameReader(sqlContext).jdbc(url='jdbc:%s' % url
       , table='movie_metadata', properties=properties)
```

Figura 4.8: Implementação das conexões com os bancos de dados e a transformação da tabela do PostgreSQL em dataframe

Os métodos criados são acessados via GET, alguns possuem nomes como parâmetros e todos retornam um JSON com os resultados obtidos e seguem o mesmo escopo: Uma consulta é realizada no Neo4j, o resultado é transformado em um dataframe, é feito um join com o dataframe criado a partir do postgreSQL, são feitos os devidos agrupamentos e agregações, é montado um JSON e o mesmo é retornado (Figura 4.9). Um exemplo está explicitado no código, em Python, exposta na Figura 4.10.

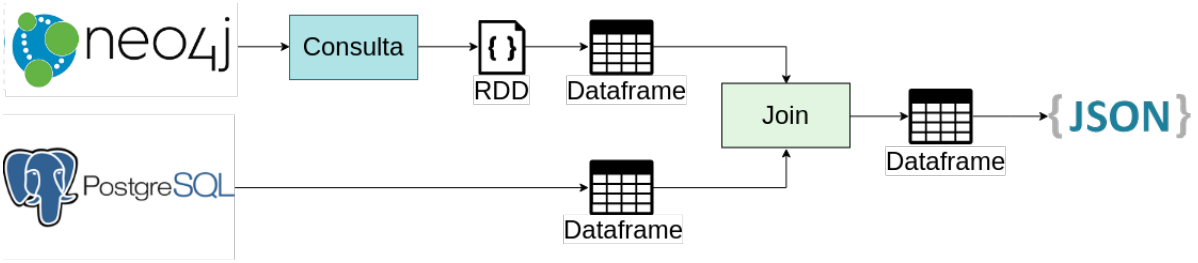

Figura 4.9: Escopo do funcionamento dos métodos

```
1  @app.route('/averagevotemoviesofactor/<string:actor_name>',
       methods=['GET'])
2  def average_vote_movies_of_actor(actor_name):
3      # Construcao da consulta no Neo4J
4      q = 'MATCH (a:Actor)-[r:ACTS_IN]->(m:Movie) WHERE a.name= "'
           + actor_name+'" RETURN m.imdbId'
5      # Realizacao da consulta
6      results = db.query(q, returns=(str))
7      rdd = sc.parallelize(results)
8
9      # Transformacao em um DataFrame
10     schema = StructType([StructField('imdbId', StringType(), True
           ),])
11     df = sqlContext.createDataFrame(rdd, schema)
12
13     # Juncao dos dados do Neo4J e do PostgreSQL
14     dfj = df.join(dfpostgres, df.imdbId == dfpostgres.imdb_id)
15     dfj = dfj.agg({'vote_average': 'mean','imdbId': 'count'})
16
17     # Construcao e retorno do JSON
18     mean = {}
19     mean["vote_average"] = dfj.rdd.map(tuple).take(1)[0][1]
20     mean["count_movies"] = dfj.rdd.map(tuple).take(1)[0][0]
21     return json.dumps(mean, indent=4, separators=(',', ': '))
```

Figura 4.10: Implementação do método average_vote_movies_of_actor(actor_name)

Ao todo foram criados 8 métodos, sendo eles:

- Método: average_vote_movies_of_actor(actor_name)
  - URL: /averagevotemoviesofactor/actor_name
  - Descrição: Retorna a média das avaliações recebidas por filmes em que o ator participou

- Método: average_revenue_movies_of_actor(actor_name)
  - URL: /averagerevenuemoviesofactor/actor_name

  - Descrição: Retorna a média dos faturamentos obtidos por filmes em que o ator participou

- Método: average_vote_movies_of_director(director_name)
  - URL: /averagevotemoviesofdirector/director_name
  - Descrição: Retorna a média das avaliações recebidas por filmes que foram dirigidos pelo diretor

- Método: average_revenue_movies_of_director(director_name)
  - URL: /averagerevenuemoviesofdirector/director_name
  - Descrição: Retorna a média dos faturamentos obtidos por filmes que foram dirigidos pelo diretor

- Método: average_profit_movies_of_director(director_name)
  - URL: /averageprofitmoviesofdirector/director_name
  - Descrição: Retorna a média dos lucros obtidos pelos filmes que foram dirigidos pelo diretor

- Método: popularityActors()
  - URL: /popularityActors
  - Descrição: Retorna o id, o nome e a soma da popularidade dos 10 atores mais populares, ou seja, os que obtiveram a maior soma de popularidade em filmes que participaram

- Método: best_profit_ad_partnership()
  - URL: /bestprofitadpartnership

  - Descrição: Retorna o id do ator, o nome do ator, id do diretor, o nome do diretor e a soma dos lucro, da melhor parceria entre ator e diretor, ou seja a que obteve a maior soma de todos lucros dos filmes em que os dois participaram

- Método: best_critical_aa_partnership()
  - URL: /bestcriticalaapartnership
  - Descrição: Retorna o id do primeiro ator, o nome do primeiro ator, id do segundo ator, o nome do segundo ator, a média das avaliações obtidas por filmes em que os dois participaram simultaneamente e quantos filmes aturam juntos, da melhor parceria entre atores, ou seja, dentre as maiores médias de avaliações, as com maiores números de filmes

A API foi inicializada utilizando o spark-subimit, um comando do Spark o qual aceita um código Python possibilitando o uso das ferramentas do Spark em um programa. Todo o código e instruções para uso foram disponibilizado em: ⟨https://github.com/joaopaulorodrigues/Spark-API/⟩.

5 Cenários de Uso

Para a avaliação deste trabalho, foi realizado de cenários de uso, com o objetivo de testar e afirmar o funcionamento da API desenvolvida, descrita na Sessão 4.2. Um cenário de uso de um sistema, descreve as etapas a qual um usuário deverá passar ao utilizar o sistema. Neste sentindo, as seguintes etapas foram seguidas. Primeiro foi escolhido o método a ser utilizado, depois, caso necessário, foi escolhido o parâmetro a ser passado, em seguida o método é chamado e, por ultimo, são exibidos os resultados.

A seguir são descritos os cenários escolhidos para a avaliação da a API, onde, passada a URL, foi retornado um JSON com o resultado. Para melhor visualização dos resultados, foi adicionado ao Chrome o plugin JSONView (2016).

5.1 Avaliação dos atores

Um usuário deseja analisar a avaliação de alguns atores. Para tal, ele utilizou o sistema proposto neste trabalho, visto que, através da solução desenvolvida, é possível encontrar dados de diferentes fontes de forma centralizada em um único ambiente.

Para acessar a API, o usuário precisou selecionar o método apropriado para a sua consulta que, neste caso, foi o método *average_vote_movies_of_actor()*. Como o mesmo necessita do nome de um ator como parâmetro, o usuário definiu ator "Robert De Niro". Quando o usuário executa o método no navegador, é retornado para ele um JSON com a média de avaliações do ator "Robert De Niro"de 6,70 em 50 filmes analisados, como visto na Figura 5.1.

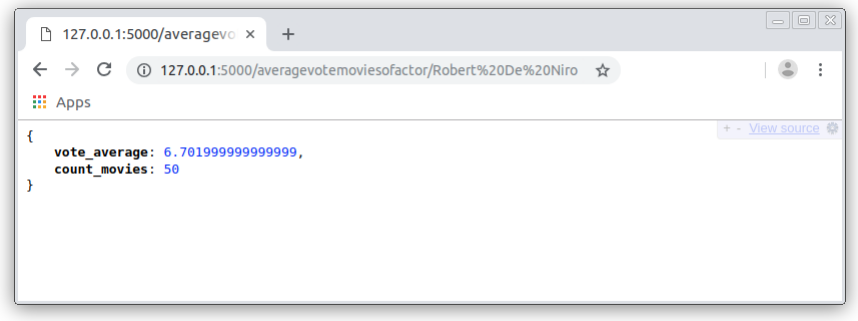

Figura 5.1: Funcionamento do método average_vote_movies_of_actor()

Seguindo com sua análise, o usuário fez outras consultas, modificando o parâmetro de entrada, para recuperar as médias dos outros atores que ele desejava analisar. Os resultados obtidos pelo usuário foram adicionados a Tabela 5.1. Considerando esses resultados, o usuário pode concluir que, para os atores selecionados, o "Charlie Chaplin"é o melhor avaliado.

Tabela 5.1: Resultados obtidos pelo usuário na análise das avaliações dos atores

Nome	Média das avaliações	Total de filmes analisados
Charlie Chaplin	7.13	10
Al Pacino	6.69	30
Robert De Niro	6,70	50
Meryl Streep	6.58	33
Adam Sandler	5.79	23
Scarlett Johansson	6.38	24

5.2 Atores mais rentáveis

Desejando avaliar, dos atores, as médias de faturamento em seus filmes, um usuário, fazendo uso da API desenvolvida, utilizou o método average_revenue_movies_of_actor(), passando como parâmetro o ator "Robert De Niro". Como resultado, retornado pelo método, o usuário observou que esse ator arrecada, em média, 76.226.965,24 dólares por filme em 50 filmes em que o mesmo atuou, como mostrado na Figura 5.2.

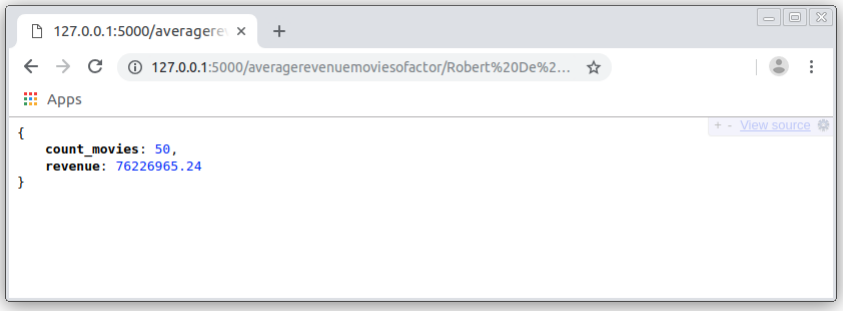

Figura 5.2: Funcionamento do método average_revenue_movies_of_actor()

Ainda com a intenção de analisar os faturamentos alcançados pelos atores, o usuário utilizou o mesmo método, porém com outros parâmetros, explicitando os resultados na Tabela 5.2. Assim, chegou a conclusão que, dos atores pesquisados, o Adam Sandler detém a maior média de faturamento por filme.

Tabela 5.2: Resultados obtidos pelo usuário na análise dos faturamentos dos atores

Nome	Média das faturamento	Total de filmes analisados
Charlie Chaplin	2.651.918,1	10
Al Pacino	75.655.089,86	30
Robert De Niro	76.226.965,24	50
Meryl Streep	78.221.074,18	33
Adam Sandler	103.462.075,74	23
Scarlett Johansson	79.294.836,33	24

5.3 Avaliação dos diretores

No terceiro cenário, o usuário deseja saber a média das avaliação dos filmes de um diretor. Para isso ele utiliza o sistema proposto, selecionando o método average_vote_movies_of_director(). Em sua consulta, como mostra a Figura 5.3, tendo o diretor James Cameron como parâmetro, o usuário conseguiu como resultado a avaliação média de 7,43, sendo analisados, ao todo, 6 filmes.

Figura 5.3: Funcionamento do método average_vote_movies_of_director()

Dando prosseguimento à sua investigação, o usuário preencheu a Tabela 5.3, com os resultados obtidos através do uso do método com diferentes entradas. Averiguando os dados presentes na mesma, concluiu que o diretor "Christopher Nolan" alcançou a maior média, sendo 7,65 em 6 filmes analisados.

Tabela 5.3: Resultados obtidos pelo usuário na análise das avaliações dos diretores

Nome	Média das avaliações	Total de filmes analisados
Charlie Chaplin	7.13	10
Peter Jackson	7.11	10
Quentin Tarantino	7.32	10
Steven Spielberg	6.99	27
James Cameron	7,43	6
Christopher Nolan	7.65	6

Outro ponto notado pelo usuário foi o fato do resultado obtido para o Charlie Chaplin, ser igual ao seu resultado como ator. Ao analisar essa situação chegou a conclusão que isso ocorreu devido a base do Neo4J, constar somente dez dos seus filmes, aos quais Chaplin atuou como ator e diretor simultaneamente.

5.4 Diretores mais rentáveis

Com o objetivo de avaliar o faturamento dos filmes de alguns diretores, um usuário utilizou o método average_revenue_movies_of_director(), presente no sistema proposto por este trabalho. Como parâmetro, escolheu o diretor "James Cameron ", cujo o método retornou como resultado, explicitado na Figura 5.4, o faturamento médio, em 6 filmes, de 917.447.838 dólares por trabalho.

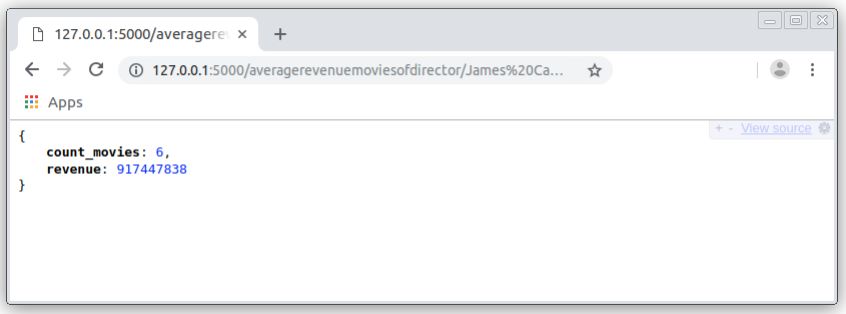

Figura 5.4: Funcionamento do método average_revenue_movies_of_director()

Com intuito de, ainda, avaliar o faturamento dos filmes de alguns diretores, o usuário executou outras consultas com vários diretores de seu interesse. Ao adicionar cada resultado na Tabela 5.4, notou que o diretor, dos pesquisados por ele, com o maior faturamento médio é o James Cameron que, em 6 filmes, atingiu o valor de 917.447.838 dólares por filme.

Tabela 5.4: Resultados obtidos pelo usuário na análise dos faturamentos dos diretores

Nome	Média das faturamento	Total de filmes analisados
Charlie Chaplin	2.651.918,1	10
Peter Jackson	359.272.130,3	10
Quentin Tarantino	110.857.415,9	10
Steven Spielberg	313.145.153,93	27
James Cameron	917.447.838	6
Christopher Nolan	273.656.639,33	6

Outro ponto avaliado pelo usuário foi que, como ocorrido e explicado na seção 5.3, o resultado para o Charlie Chaplin como diretor é equivalente ao como ator, devido as mesmas explicação *a priori* descritas.

5.5 Diretores mais lucrativos

A fim de descobrir qual diretor gerou maior lucro em seus trabalhos, um usuário seleciona, na API desenvolvida, o método average_profit_movies_of_director(), responsável por retornar essa informação. Como mostra a Figura 5.5, este usuário escolhe como parâmetro o diretor James Cameron. Em seu navegador é exibido como resposta um faturamento de 917.447.838 dólares e o custo de 105.316.666,66 dólares, gerando um lucro de 812.131.171,33 dólares por filme, em 6 filmes analisados.

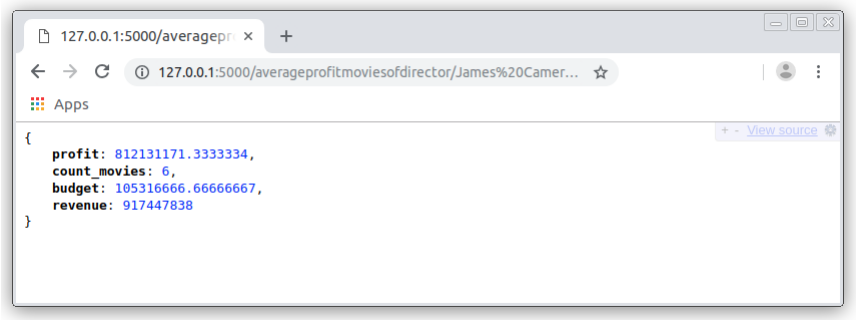

Figura 5.5: Funcionamento do método average_profit_movies_of_director()

Objetivando comparar o resultado obtido com outros diretores, o usuário executou este mesmo método com a várias entradas diferentes e as explicitou na 5.5. Com os resultados atingidos por essa iniciativa foi confirmado o "James Cameron"como o mais lucrativo dos diretores pesquisados, com um lucro médio de 812.131.171,33 dólares por filme.

Tabela 5.5: Resultados obtidos pelo usuário na análise dos lucros alcançados pelos diretores

Nome	Média de faturamento	Média de gastos	Média de lucro	Total de filmes
Charlie Chaplin	2.651.918,1	817.300,1	1.834.618	10
Peter Jackson	359.272.130,3	57.677.571	301.594.559,3	10
Quentin Tarantino	110.857.415,9	22.020.000	88.837.415,9	10
Steven Spielberg	313.145.153,93	52.812.962,96	260.332.190,96	27
James Cameron	917.447.838	105.316.666,66	812.131.171,33	6
Christopher Nolan	273.656.639,33	71.667.666,67	201.988.972,67	6

5.6 Atores mais populares

Um usuário deseja analisar a popularidades dos atores. Para isso, utilizou o sistema proposto neste trabalho para obter tal informação, sendo o método average_profit_movies_of_director() selecionado para tal tarefa. Como esse não possui parâmetro, através do seu navegador, o usuário executou método que retornou os, em ordem crescente de posição, de cima para baixo, os 10 atores cuja a soma da popularidade do seus filmes, são as maiores, como mostra a Figura 5.6. Com esses dados em mãos, o usuário concluiu que o ator "Jeff Bridges" é o de maior popularidade.

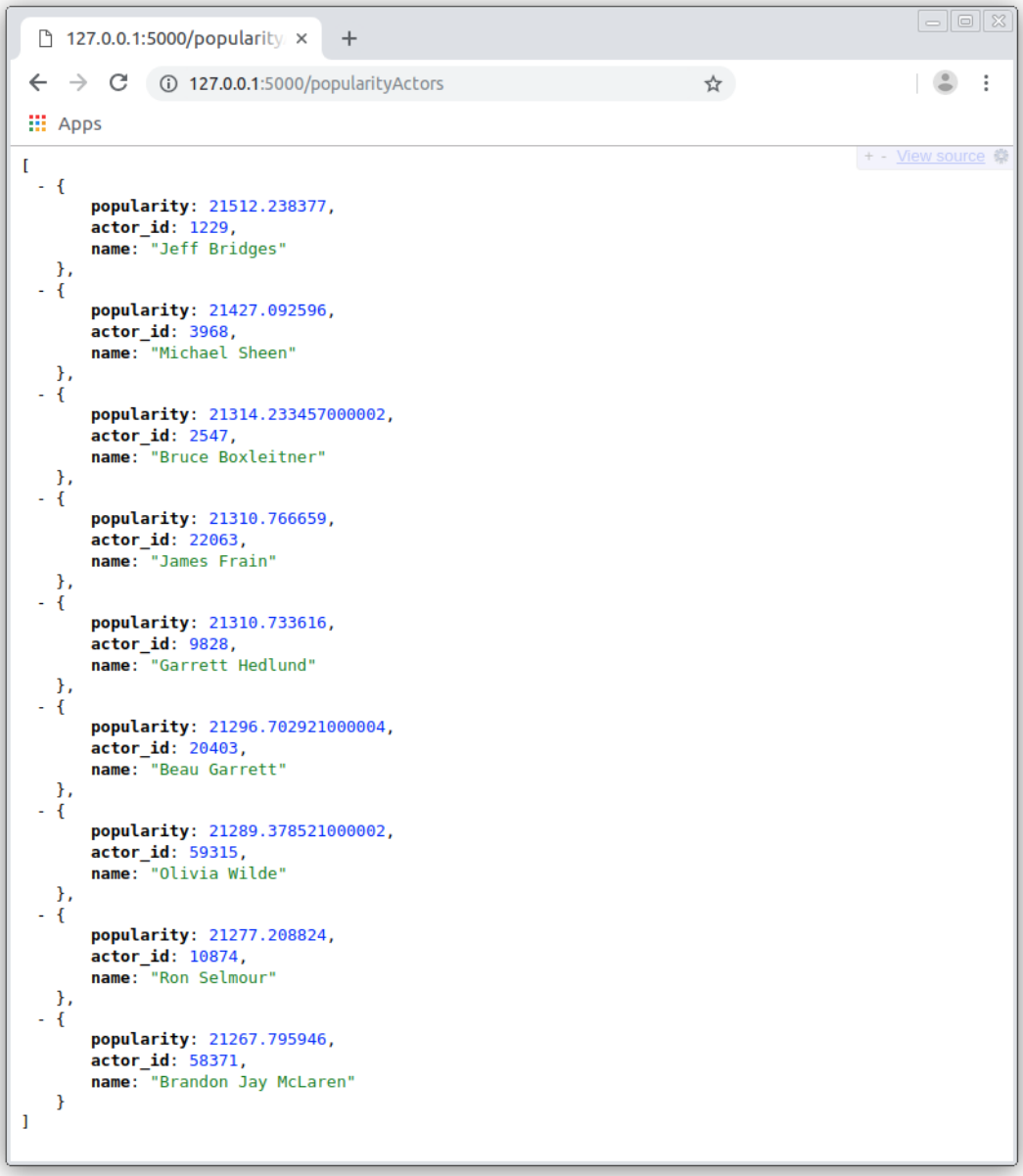

Figura 5.6: Funcionamento do método popularityActors()

5.7 Parceria mais lucrativa entre ator e diretor

Desejando descobrir qual parceria entre um ator e um diretor redeu mais lucro, um usuário escolheu, como o meio de obter essa informação, o método best_profit_ad_partnership(), pertencente a proposta deste trabalho.

Como não foi necessário escolher parâmetros, via navegador,o usuário chamou o método que, por sua vez, apresentou, como mostra a Figura 5.7, a parceira do diretor David Yates e do ator Michael Gambon como a mais lucrativa, alcançando o lucro de 4.168.477.803 dólares.

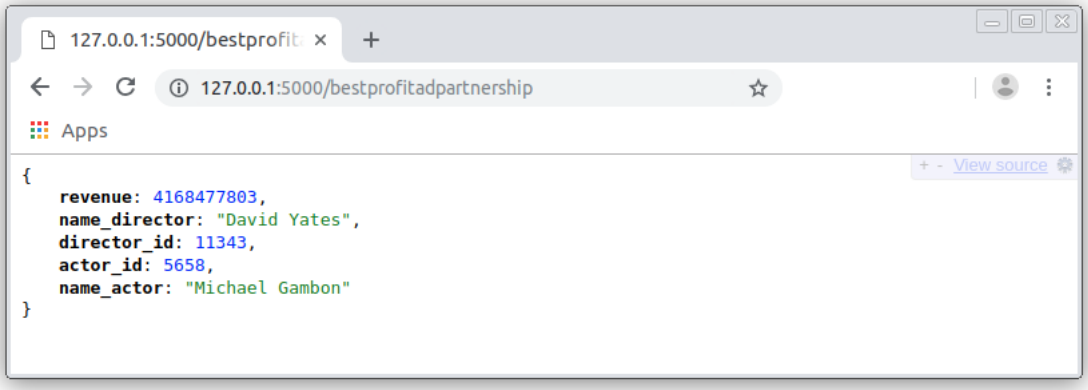

Figura 5.7: Funcionamento do método best_profit_ad_partnership()

5.8 Melhor parceria entre dois atores

Como o objetivo de analisar a melhor parceria entre dois atores, um usuário escolheu a API proposta neste trabalho para obter tal informação. Assim, o mesmo, selecionou o método best_critical_aa_partnership() presente na API e como esse não possuí parâmetro de entrada, foi executado no navegador, como mostra a Figura 5.8. Com o resultado retornado, o usuário concluiu que Terence Hill e Bud Spencer são a melhor parceria entre atores, devido aos 13 filmes em que atuaram juntos, cuja a média de avaliações é de 6.58, aproximadamente.

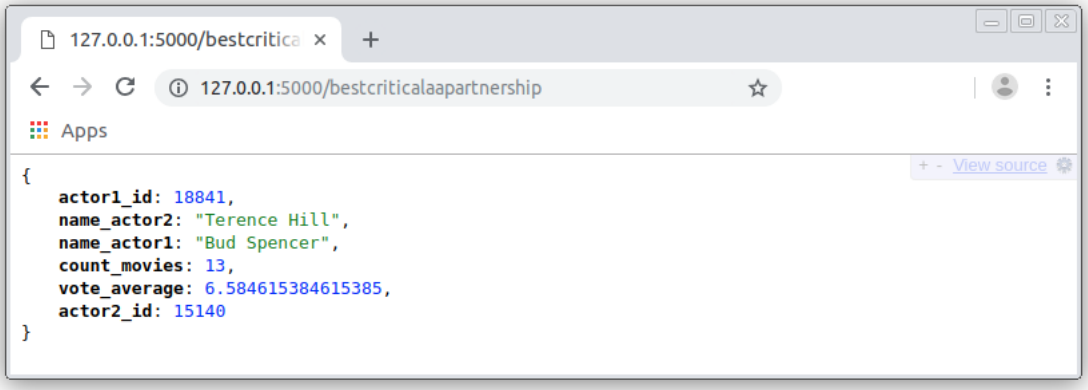

Figura 5.8: Funcionamento do método best_critical_aa_partnership()

5.9 Análise dos resultados

Analisando os resultados transcritos nas tabelas, foi notado que a integração entre as duas bases de dados ocorreu como esperado. Assim, informações contidas no Neo4J, como quais atores e diretores trabalharam em um determinado filme ou quais atores atuaram juntos, por exemplo, foram integradas às informações do PostegreSQL, como Faturamento, Gastos e Popularidade dos filmes. Com isso, como previso, pôde se obter novos conhecimentos através da estrutura proposta.

Os usuários não tiveram dificuldade em utilizar a API proposta, mesmo sendo uma proposta inicial. Entretanto, melhorias são necessárias, como a criação de uma interface que facilite o uso dos métodos desenvolvidos, principalmente, quando o usuário precisa chamar o mesmo método mais de uma vez.

6 Considerações finais

Este trabalho propôs uma abstração com o uso de uma API em Python visando diminuir as barreiras encontradas ao trabalhar com fonte de dados heterogêneas. Combinando a capacidade analítica e de se conectar às múltiplas fontes de dados, com o encapsulamento obtido com o uso de uma API, foi possível criar uma solução transparente para obtenção de conhecimento.

O arquitetura proposta se mostrou eficiente e intuitiva. Possibilitando um fácil acesso aos resultados, abstraindo da integração as fontes de dados e possibilitando, de maneira simples, a escalabilidade da heterogeneidade dos dados conectando-se facilmente à outras formas de armazenamento de dados.

O Apache Spark, devido suas abstrações e ferramentas nativas, se apresentou como uma forma robusta de integração de dados heterogêneos. Com ele foi possível transformar quaisquer forma de dados em dataframes, que são as abstrações de tabelas usadas no SparkSQL. Assim as integrações se tornaram triviais e com os dados unidos em uma só tabela, a manipulação dos mesmos também se torna trivial.

Outro fator a ser observado é a característica do Spark em manter os dados em memória para, assim, analisar-los. Isso tornou o processamento das consultas, principalmente mais complexas e com maiores volumes da dados, como a realizado no método best_critical_aa_partnership(), que analisa todos os filmes de todos os pares de atores, relativamente rápidas, obtendo o resultado em poucos segundos.

Como trabalhos futuros, métricas, como velocidade, serão utilizadas para determinar a escalabilidade da arquitetura proposta, sendo possível compara-lá às outras soluções de integração de dados disponíveis na literatura. Outro ponto a ser tangenciado em outros projetos, é a validação dos resultados, pois, devido a complexidade das consultas fora do escopo proposto, não foi possível realiza-la neste trabalho.

Outra proposta para futuros trabalhos é tornar a solução apresentada nesse trabalho mais genérica, automatizando as conexões com as bases de dados e as consultas considerando as estruturas dos bancos conectados.

Bibliografia

CHASSEUR, C.; LI, Y.; PATEL, J. M. Enabling json document stores in relational systems. In: *WebDB*. [S.l.: s.n.], 2013. v. 13, p. 14–15.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, ACM, v. 51, n. 1, p. 107–113, 2008.

DHAR, V. Data science and prediction. *Communications of the ACM*, ACM, v. 56, n. 12, p. 64–73, 2013.

GANTZ, J.; REINSEL, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, v. 2007, n. 2012, p. 1–16, 2012.

HAI, R.; QUIX, C.; ZHOU, C. Query rewriting for heterogeneous data lakes. In: SPRINGER. *European Conference on Advances in Databases and Information Systems*. [S.l.], 2018. p. 35–49.

HALEVY, A.; RAJARAMAN, A.; ORDILLE, J. Data integration: the teenage years. In: VLDB ENDOWMENT. *Proceedings of the 32nd international conference on Very large data bases*. [S.l.], 2006. p. 9–16.

HOLZSCHUHER, F.; PEINL, R. Querying a graph database–language selection and performance considerations. *Journal of Computer and System Sciences*, Elsevier, v. 82, n. 1, p. 45–68, 2016.

JSONVIEW. *JSONView*. 2016. Disponível em: ⟨https://github.com/gildas-lormeau/JSONView-for-Chrome⟩.

KOLEV, B. et al. Cloudmdsql: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases*, Springer, v. 34, n. 4, p. 463–503, 2016.

LAU, R. Y. et al. Big data commerce. *Information and Management*, ELSEVIER SCIENCE BV, v. 53, n. 8, p. 929–933, 2016.

LI, L.; CHOU, W. Design and describe rest api without violating rest: A petri net based approach. In: IEEE. *Web Services (ICWS), 2011 IEEE International Conference on*. [S.l.], 2011. p. 508–515.

MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. [S.l.]: "O'Reilly Media, Inc.", 2011.

PROVOST, F.; FAWCETT, T. Data science and its relationship to big data and data-driven decision making. *Big Data*, Mary Ann Liebert, Inc. 140 Huguenot Street, 3rd Floor New Rochelle, NY 10801 USA, v. 1, n. 1, p. 51–59, 2013.

REUTTER, J. L.; ROMERO, M.; VARDI, M. Y. Regular queries on graph databases. *Theory of Computing Systems*, Springer, v. 61, n. 1, p. 31–83, 2017.

SEARLS, D. B. Data integration: challenges for drug discovery. *Nature reviews Drug discovery*, Nature Publishing Group, v. 4, n. 1, p. 45, 2005.

SHANAHAN, J. G.; DAI, L. Large scale distributed data science using apache spark. In: ACM. *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. [S.l.], 2015. p. 2323–2324.

SIVARAJAH, U. et al. Critical analysis of big data challenges and analytical methods. *Journal of Business Research*, Elsevier, v. 70, p. 263–286, 2017.

SONG, I.; ZHU, Y. Big data and data science: what should we teach? *Expert Systems*, Wiley Online Library, v. 33, n. 4, p. 364–373, 8 2016.

TIAN, Y. et al. Joins for hybrid warehouses: Exploiting massive parallelism in hadoop and enterprise data warehouses. In: *EDBT*. [S.l.: s.n.], 2015. p. 373–384.

TUIJN, C.; GYSSENS, M. Cgood, a categorical graph-oriented object data model. *Theoretical Computer Science*, Elsevier, v. 160, n. 1-2, p. 217–239, 1996.

ZAHARIA, M. et al. Fast and interactive analytics over hadoop data with spark. *Usenix Login*, v. 37, n. 4, p. 45–51, 2012.

ZAHARIA, M. et al. Spark: Cluster computing with working sets. *HotCloud*, v. 10, n. 10-10, p. 95, 2010.

ZAHARIA, M. et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, ACM, v. 59, n. 11, p. 56–65, 2016.