Rumo à Otimização de Operadores sobre UDF no Spark

João Antonio Ferreira[1], **Fábio Porto**[2], **Rafaelli Coutinho**[1], **Eduardo Ogasawara**[1]

[1]CEFET/RJ - Centro Federal de Educação Tecnológica Celso Suckow da Fonseca

[2]LNCC - Laboratório Nacional de Computação Científica

joao.parana@acm.org, fporto@lncc.br

rafaelli.coutinho@cefet-rj.br, eogasawara@ieee.org

Abstract. *The analysis of unstructured and semi-structured data on a large scale has been of great importance in the scientific community due to the large volume of knowledge stored on the WEB. UDF (User Defined Functions) can be used in this context, but the difficulty of establishing clear semantics about their behavior prevents Spark from performing the necessary optimizations. This work aims to develop data analysis workflows optimizations that can be written as UDF in large-scale distributed processing environments, integrating with applications written in Python, R, Java, Kotlin and Scala. To do this, it is intended to demonstrate that it is possible to use SparkSQL's functionalities to develop a set of filter and map optimizations that operate over UDF using the Spark* Catalyst API. *Workflow algebra is used for UDF semantic annotation. As proof of concept, two optimization classes were created and evaluated. The experiments indicate that it is possible to consider information from a retrospective provenance for semantic annotation on UDF ruled by filter and map operators to get the desired optimizations.*

Resumo.
A análise de dados não estruturados e semi-estruturados em larga escala tem ganhado muita importância na comunidade científica devido ao grande volume de conhecimento armazenado na WEB. UDF (User Defined Functions) podem ser usadas neste contexto, porém a dificuldade de se estabelecer semântica clara sobre seus comportamentos, impede o Spark de realizar as otimizações necessárias. Este trabalho tem como objetivo desenvolver a otimização das execuções de workflows de análise de dados que possam ser escritas como UDF em ambientes de processamento distribuído em larga escala, integrando com aplicações escritas em Python, R, Java, Kotlin e Scala. Para isso, pretende-se demonstrar que é possível usar as funcionalidades do SparkSQL e desenvolver um conjunto de otimizações em filtros (filter) e mapeamentos (map) que operam sobre UDF usando a API Catalyst *do Spark. A álgebra de workflow é usada para anotação semântica de UDF. Como prova de conceito, foram criadas e avaliadas duas classes de otimizações. Os experimentos indicam que é possível considerar informações de proveniência retrospectiva para anotação semântica de otimização de UDF regidas pelos operadores filter e map.*

1. Introdução

Diversos problemas científicos de análise de dados são computacionalmente intensivos, o que exige dos pesquisadores o desenvolvimento de otimizações nos algoritmos e o uso de plataformas de hardware específicas, como GPU e FPGA, que são mais adequadas a este tipo de processamento [Morcel et al., 2016]. Em termos de otimização de algoritmos, as plataformas tradicionais de computação científica focam a otimização em métodos matemáticos, como na álgebra linear [Nothaft et al., 2015] e no cálculo matricial. No entanto, surgiram recentemente necessidades de uso intenso de técnicas de aprendizagem estatística e funções definidas pelo usuário (UDF), em especial, sobre dados semiestruturados. Este movimento foi impulsionado pelo aumento da quantidade de informações disponíveis aos cientistas de dados. Em paralelo a este movimento, necessidades de processar volumes de dados extremamente grandes levaram ao desenvolvimento de soluções de análise de dados em larga escala, escaláveis horizontalmente, tal como o modelo MapReduce [Dean and Ghemawat, 2008], implementado pelo Apache Hadoop e pelo Apache Spark [Zaharia et al., 2010].

Outra linha de ação vem desenvolvendo nos últimos anos bibliotecas escritas em Python (NumPy, SciPy, Scikit-learn) e pacotes R para prover aos cientistas toda uma classe de soluções numéricas para análise exploratória, cálculo vetorial, álgebra linear, métodos estatísticos, visualização de dados e aprendizado de máquina. O uso de UDF no Spark permite aliar a análise de dados em larga escala com as soluções escritas em Python e R. No entanto, apesar de sua importância, o uso de UDF não tem ganhado suficiente atenção da comunidade científica devido a dificuldade de se estabelecer semântica clara sobre seus comportamentos, que via de regra possuem códigos arbitrários relacionados ao domínio e, portanto, com características muito diferentes entre elas. Com a falta de conhecimento sobre a semântica da operação, o Spark é incapaz de otimizar a execução, criando barreiras para sua utilização em diversos cenários [Armbrust et al., 2015]. Rheinländer et al. [2017] dedicam um estudo extenso a questão da otimização de *workflow* com UDF, apontando os principais desafios relacionados e as abordagens possíveis para solução, dentre as quais a álgebra de *workflow* de Ogasawara et al. [2011].

Neste contexto, este trabalho propõe o desenvolvimento de otimização das execuções de *workflows* de análise de dados, que possam ser escritas como UDF em ambientes de processamento distribuído em larga escala, integrando com aplicações de terminal (*i.e.* não visuais) que usem bibliotecas e códigos escritos em Python, R, Java, Kotlin e Scala. Para isso, pretende-se demonstrar que é possível usar as funcionalidades do ecossistema Spark/Hadoop e desenvolver, em linguagem Scala, um conjunto de otimizações em filtros (operador *filter*) e mapeamentos (operador *map*) que operam sobre UDF usando a API *Catalyst* do Spark.

Além desta introdução, o trabalho está organizado em mais quatro seções. Na seção 2, são apresentados conceitos gerais necessários para o entendimento do problema e da metodologia aplicada. Em seguida, na seção 3, apresenta-se a metodologia adotada nesse trabalho e a implementação da solução proposta na forma de prova de conceito. A seção 4 mostra a avaliação preliminar com os resultados que corroboram a hipótese. Finalmente, a seção 5 apresenta as conclusões e os próximos passos.

2. Referenciais Teóricos

Os estudos sobre Spark se iniciaram com Zaharia et al. [2010]. Os autores trataram do problema de executar processos distribuídos em *cluster* com tolerância a falhas e a manutenção dos dados intermediários em memória, sempre que possível, para obter desempenho superior ao do Apache Hadoop. Assim, Spark é um *framework* que possibilita a execução das tarefas paralelizáveis de forma distribuída em máquinas *multi-core* ou *clusters* YARN/Mesos, com ênfase no processamento em *pipeline* das atividades que compõem um *dataflow* [1]. O estudo evoluiu e o Spark ganhou outros módulos, como o SparkSQL [Armbrust et al., 2015] e o SparkR [Venkataraman et al., 2016]. Nesta seção serão apresentados conceitos gerais de Spark e de álgebra de *workflow* [Ogasawara et al., 2011], que é outro conceito importante usado neste trabalho.

2.1. Apache Spark, *SparkSQL* e *Catalyst*

O ambiente Spark permite otimizações de código por meio do *Catalyst* [Armbrust et al., 2015], que é um arcabouço de software para apoiar a otimização de expressões de qualquer natureza incluindo aquelas referentes às relações (*relations*) e seus esquemas (*schemas*) participantes de um *dataflow*. O Spark possui uma arquitetura modular que permite a inclusão de módulos adicionais. O *SparkSQL* é considerado um dos módulos mais importantes do Spark e faz uso do *Catalyst* para otimizar a geração de código em tempo de execução à partir de expressões e comandos SQL. Este módulo permite acesso a *datasets* usando uma interface baseada em álgebra relacional para processamento de dados estruturados [Armbrust et al., 2015]. As interfaces do *SparkSQL* fornecem informações adicionais sobre a estrutura dos dados em catálogo e a computação que está sendo executada. Internamente, o *SparkSQL* usa essas informações extras para realizar as otimizações.

Para facilitar o acesso às funcionalidades do *Catalyst*, foram criados na versão 2.2 do Spark, vários pontos de extensão[2], permitindo que desenvolvedores façam suas próprias implementações. Isso possibilita a personalização do ambiente Spark com o uso de regras próprias para otimização nos seguintes escopos: i) analisador gramatical (*parser*); ii) analisador (*analyzer*); iii) otimizador (*optimizer*); iv) estratégia de planejamento físico (*physical planning strategy*)[3]. Estes pontos de extensão são suficientes para desenvolver uma gama enorme de otimizações personalizadas no *SparkSQL*.

Além da otimização de código, o *SparkSQL* permite outras funcionalidades graças à funcionalidade de inferência de esquema, como por exemplo, JOIN entre tabelas de sistema de gerência de banco de dados (SGBDs) relacionais e arquivos JSON ou XML. Além disso, ele apoia a criação de UDF e seu respectivo registro no catálogo, permitindo ao componente *Catalyst* considerá-las no processo integrado de otimização do *dataflow*.

Segundo Armbrust et al. [2015], para usar o *Catalyst* é necessário lançar mão de características específicas da linguagem Scala, tais como, funções parciais, múltiplas listas de argumentos, correspondência de padrões, *quasiquotes* e tipagem implícita [Wampler and Payne, 2014]. Com essas características é possível criar uma DSL (*Domain*

[1] *dataflow* significa: *data-centric workflow* ou fluxo de trabalho centrado nos dados, que difere de fluxo de trabalho centrado no controle (*control-centric workflow*) fora do escopo desse trabalho.

[2] https://issues.apache.org/jira/browse/SPARK-18127

[3] O processo de desenvolvimento d otimizações no *Catalyst* (*SparkSQL*) foi publicado em um Wiki no GitHubcujo endereço é:
- https://github.com/joao-parana/wff-catalyst/wiki/Otimização-de-consultas

Specific Language) interna para traduzir o código Scala para SQL [Cheney et al., 2013]. Desta forma, no *SparkSQL*, dado um *dataflow* especificado em código Scala imperativo pode-se transformá-lo em código declarativo, tal como SQL, e otimizá-lo usando o *Catalyst*. Esta é abordagem que foi adotada neste trabalho.

2.2. Álgebra de *Workflow*

Rheinländer et al. [2017] destacam a relevância da otimização de UDF e apontam a abordagem algébrica para execução de *workflow* proposta por [Ogasawara et al., 2011] como uma das soluções para a anotação de semântica em UDF, que permite otimizações. Esta proposta algébrica tem como premissa a existência de operadores que regem a execução das atividades e que impõem uma semântica sobre a produção e o consumo dos dados. Esses operadores podem ser considerados como adicionais a álgebra relacional com a característica de associarem as relações de entrada a uma UDF escrita numa linguagem de programação disponível. Estes operadores funcionam como anotadores de semântica nas UDF para que o otimizador use as heurísticas adequadas em suas regras.

Neste contexto, a notação $A < R_1, ...R_n, T, F >$ representa uma atividade A que depende das relações de entrada R_1, ... R_n, de uma função definida pelo usuário F e que gera uma relação de saída com esquema T. Cada uma das relações de entrada está associada aos esquemas: S_1, ... S_n, respectivamente. Pode-se definir $udf(A)$ como uma função definida pelo usuário associada a um operador algébrico que rege a atividade A. A álgebra de *workflows* de [Ogasawara et al., 2011] descreve 6 operadores, mas serão detalhados aqui apenas dois (*MAP* e *FILTER*) que estão dentro do escopo deste trabalho. O operador *MAP* é definido da seguinte forma: $T \leftarrow MAP(A, R)$, onde a atividade A produz uma única tupla na relação de saída T para cada tupla consumida na relação de entrada R. Os esquemas de R e T podem ser diferentes num caso mais geral. Já o operador *FILTER* é descrito assim: $T \leftarrow FILTER(A, R)$, onde a atividade A produz uma ou nenhuma tupla na relação de saída T para cada tupla consumida na relação de entrada R. Neste processo, o esquema de T (saída) é o mesmo de R (entrada). Portanto, é possível observar que a semântica dos operadores *MAP* e *FILTER* de [Ogasawara et al., 2011] são compatíveis com seus análogos *map* e *filter* do *SparkSQL*.

3. Prova de conceito

O Apache Spark foi adotado para apoiar os usuários na execução de *workflow* de análise de dados com UDF em ambientes de processamento distribuído em larga escala [Ferreira et al., 2017]. Uma das vantagens desta abordagem é deixar a responsabilidade de toda a complexidade do modelo de execução, incluindo resiliência e desempenho, para o Spark. Isso diferencia a proposta de [Ferreira et al., 2017] dos outros sistemas de gerenciamento de *workflow* (SGW), que assumem esta responsabilidade de gestão do ambiente de execução e distribuição de tarefas entre os nós do *cluster*. Desta forma, o foco se restringe a anotação semântica de UDF para prover informação necessária a otimização do *workflow* em tempo de execução. Para isso, uma análise exploratória das APIs do *Catalyst* e dos pontos de extensão (citados na seção 2.1) foi realizada, e o escopo deste trabalho foi limitado em dois tipos de otimizações: os operadores *map* e *filter*.

Como motivação para nosso experimento *in-silico*, imaginemos um problema descrito da seguinte forma: considere um fragmento de workflow, com duas UDF regidas por um operador $FILTER$, como mostrado abaixo, já em linguagem Scala:

A otimização já implementada no Spark (chamada *CombineFilters*) permite transformar esta expressão da Figura 1(a) nesta outra da Figura 1(b), pois junta as expressões de dois filtros adjacentes.

```
myDataset.filter(slowerUDF(some, parameters)).filter(fasterUDF(param1, ...))   (a)

myDataset.filter(slowerUDF(some, parameters) && fasterUDF(param1, ...))        (b)
```

Figura 1. (a) Segmento de *workflow* antes da otimização *CombineFilters* do Spark; (b) Segmento de *workflow* depois da otimização *CombineFilters* do Spark,

Esta otimização é possível, pois $FILTER(B, FILTER(A, R)) = FILTER(B\&\&A, R))$ pela álgebra de *workflow*, com $\&\&$ sendo o operador lógico AND. É importante dizer que na fase de geração de código, o gerador de *byte-code* do *SparkSQL* otimiza ainda mais, gerando um (*branch*) de curto circuito entre as duas expressões. Apesar de todas as otimizações implementadas pelo Spark ainda resta um problema: este código avalia primeiro uma função dispendiosa ($slowerUDF$) e, em caso desta retornar $true$, segue para avaliar o restante da expressão. Quando o primeiro teste na $slowerUDF$ falha, a avaliação sofre o curto-circuito, mas ainda assim existe uma penalidade de desempenho significativa, pois o custo computacional de $slowerUDF$ pode ser muito grande e penaliza o processamento de todas as tuplas.

Com o objetivo de otimizar este tipo de *workflow*, foi criada uma classe (chamada *ChangePositionOfTwoAdjacentsFiltersContainingUDF* [4]) escrita em linguagem Scala para funcionar como ponto de extensão. Ela consiste de uma *case class* do Scala[5] e é responsável por transformar planos lógicos, aplicando otimizações. O objetivo desta classe é trocar a posição de dois operadores *Filter* adjacentes. Isso só é possível se os operadores atuam sobre UDF anotadas com semântica de custo de execução diferenciado e se existe ganho de desempenho relevante no *workflow* com a troca. Para isso, foram implementados dois métodos, *hasTwoUDFOneInEachFilter* e *apply*. O primeiro método avalia dois *Filters* e devolve um valor booleano, indicando se os dois atendem os requisitos em relação a regerem ou não UDF anotadas. O segundo é o ponto de entrada do *Catalyst* na classe de otimização personalizada. Este método *apply* recebe um plano lógico e deve retornar um plano lógico de acordo com os requisitos de interface do *Catalyst*. De modo análogo, foi desenvolvida outra classe em linguagem Scala para otimizar operadores *map* (chamada *AddBarrierForConstrainedActivity*). No entanto, esta classe não usa plenamente o *Catalyst* e sim, as funcionalidades *quasiquotes* e macros da linguagem Scala para atuar no *dataflow* antes mesmo de passar ao *Catalyst*.

A Figura 2 mostra o casamento de padrão para o caso da otimização de *filter* contendo UDF. Observe que é necessário fazer o casamento de padrão ao mesmo tempo em que os valores dos objetos são extraídos e atribuídos como um todo a uma variável. Este casamento de padrão é conhecido como ligação de variáveis em cláusulas *case* (*Binding Variables in case Clauses*). Na Figura 2, as variáveis *f*, *ab* e *ff* indicam, respectivamente, o primeiro *Filter*, o *AnalysisBarrier* e o segundo *Filter*. Elas são acessadas no código que

[4]O código fonte Scala da implementação pode ser acessado no repositório público no endereço https://github.com/joao-parana/wff-catalyst

[5]a classe *ChangePositionOfTwoAdjacentsFiltersContainingUDF* estende a classe *Rule[LogicalPlan]*

processa o casamento do respectivo padrão para que se consiga alterar a árvore de regras do plano lógico de acordo com a necessidade.

```
case f @ Filter(condition: Expression, ab @ AnalysisBarrier(ff @ Filter(_, grandGChild)))
    if hasTwoUDFOneInEachFilter(f, ff) => {
```

Figura 2. Casamento de padrão para dois operadores *Filter* com um elemento *AnalysisBarrier* entre eles. O método *hasTwoUDFOneInEachFilter* verifica se os filtros estão aptos a otimização.

4. Avaliação Preliminar

Nesta avaliação procurou-se verificar a capacidade do Spark em otimizar UDF em duas situações distintas: *i)* UDF anotada com informações relativas a custo computacional médio de execução em runtime, por cada tupla consumida; *ii)* UDF anotada com informações sobre a restrição da atividade associada em termos de uso excessivo de recursos computacionais que [Ogasawara et al., 2011] chamou de *Constrained Activity* em seu trabalho sobre Álgebra de Workflow. Ainda que o problema seja genérico, pode-se definí-lo por via de dois exemplos distintos descritos nas seções seguintes.

4.1. Otimização com Filter

Em ciência dos materiais estuda-se os fenômenos fluência (*creep*) e fadiga (*fatigue*) térmica que os materiais sofrem resultante de tensões cíclicas causadas por variações de temperatura [Cardoso et al., 2015]. Existem modelos que descrevem como um dado material sofre degradação diminuindo sua vida útil. Considere que o cientista tenha acesso a um *dataset* contendo o histórico de temperaturas as quais um dado equipamento esteve submetido nos últimos 3 anos e precisa criar um workflow que determine os momentos em que ocorreram danos por fadiga maiores que um certo limiar, mas apenas aqueles que ocorreram simultaneamente a um transiente segundo certos critérios. Ele define o seguinte workflow em Scala:

$$dsR0.filter(doFatigueDamageCalc(p) > 0.7).filter(getTransientStart(q) > 0)$$

onde $doFatigueDamageCalc$ e $getTransientStart$ são funções escritas em Python para calcular respectivamente o dano por fadiga térmica e o *timestamp* relativo do transiente de temperatura (caso ocorra, será maior que zero). Ao rodar o workflow percebe-se que o processo não terminará no tempo desejado é que é necessário melhorar o desempenho. Este é um exemplo que ilustra uma situação real que pode ocorrer com um pesquisador.

Anotando as UDF com semânticas de custo de execução [6], pode-se trocar a ordem de execução dos filtros (a operação lógica AND é comutativa) de forma que a função $getTransientStart$ seja executada primeiro, o que potencialmente diminuirá consideravelmente o custo total de execução do programa [7]. É neste ponto que entra a álgebra de workflow.

[6] considera-se aqui as informações de proveniência retrospectiva para anotação semântica de otimização de UDF regidas pelos operadores filter

[7] Outros exemplos incluem fenômenos elétricos de corrente alternada com retificação, que podem ser considerados neste tipo de otimização pois $seno(\theta) < 0$ indica sinal retificado e constante que poderia ser ignorado em alguns experimentos *in-silico* de custo computacional muito alto

Para obter o resultado da fadiga térmica descrito anteriormente foi construído um experimento *in-silico* baseado na seguinte especificação: dado 2 UDF regidas pelo operador FILTER da Álgebra de Workflow de Ogasawara et al. [2011], que usam um *dataset* com Schema $S = \{x, p, q\}$, deseja-se trocar a ordem das operações no plano lógico de execução *(logical plan)*, dado que uma operação nomeada A depende apenas de $\{x, p\}$ e a operação nomeada B depende apenas de $\{x, q\}$. Tanto a operação A quando a operação B são regidas pelo operador Filter. Operações, neste contexto, são UDF escritas em Scala ou Java [8] e o *dataset* é um objeto *Dataset/DataFrame* ou *LocalRelation* do Spark. Assume-se que a troca de ordem é necessária devido aos custos diferenciados de execução sendo que o custo de uma operação é consideravelmente maior que o da outra.

Dado o segmento de Workflow data-centric:

```
FILTER(udfB(q) > 0, FILTER(udfA(p) > 0.7, R0))
```

com S0 = {x, p, q} sendo o esquema de R0 e também:

```
udfA(p: Int) => { doFatigueDamageCalc(p) }
udfB(q: Int) => { getTransientStart(q) }
```

No Spark, após anotação semântica, o código acima se transforma em:

```
spark.udf.register("udfA_99", udfA)
spark.udf.register("udfB_10", udfB)
dsR0.filter("udfA_99(x,p)>0.7").filter("udfB_10(x,q)>0")
```

Onde dsR0 é o objeto Dataset representando a relação de entrada R0.

As UDF são definidas em linguagem Scala, registradas no catálogo e usadas com sinônimos (alias) udfA_99 e udfB_10, que trazem no seu sufixo um inteiro > 0, separado do nome original por underscore para facilitar a execução do analisador (que identifica os custos de computação). Este sufixo é a anotação semântica citada.

Ao executar o teste observou-se, pelo *log*, que a udfB com custo 10 aparece antes da udfA com custo 99 no plano lógico otimizado, como desejado, demonstrando o funcionamento da otimização.

4.2. Otimização com Map

No segundo exemplo, relacionado à *Constrained Activity*, usou-se o operador MAP:

$$myDataset.map(heavyUDF1(some, parameters)).map(heavyUDF2(param, ...))$$

Neste caso, o Spark juntará os processamentos das duas UDF em *pipeline* pois desconhece a semântica da operação executada por ela e os recursos computacionais consumidos. Caso estas UDF sejam escritas em código nativo, cada uma delas poderá consumir todos os recursos de *thread* ou memória do processador de forma que não possam ser invocadas em *pipeline*. Ogasawara et al. [2011] chama este fenômeno de *Constrained Activity* e propõe que a solução seja a criação de uma barreira forçando a materialização do resultado intermediário entre a avaliação das duas UDF.

[8] Uma UDF escrita em Scala ou Java pode invocar código externo escrito em Python ou R desde que os nós do *cluster* tenham acesso a este código de de forma canônica

É possível criar as barreiras citadas anotando as UDF com informações de uso de memória e de processadores. Neste caso, o resultado da execução é obtido, o que não seria possível na versão não otimizada original, pois o programa terminaria logo no início por falta de recursos computacionais disponíveis. Da mesma forma que no exemplo do operador FILTER, a anotação semântica é feita via sufixo. Entretanto ela é usada na fase inicial da análise gramatical (*parser*) para dividir o workflow original em dois segmentos e ordená-los antes de passar ao *Catalyst* para que seja feita as outras otimizações possíveis em cada um os segmentos. As materializações no Spark podem ser provocadas por operadores *collect* e foi esta a abordagem escolhida.

5. Considerações finais

As otimizações propostas neste trabalho são largamente utilizadas em banco de dados e podem também ser usadas em *workflows* de análise de dados. Em [Ferreira et al., 2017] foi abordado a possibilidade de usar o Spark como base para a implementação de um *framework* de análise de dados usando a álgebra de *workflow* para facilitar o trabalho dos cientistas. Este trabalho demonstra que esta abordagem é viável e que é possível criar implementações de otimizações arbitrárias incluindo UDF com o uso dos pontos de extensão disponíveis no SparkSQL. Desta forma, pretende-se, em trabalhos futuros, adaptar as heurísticas propostas por [Ogasawara et al., 2011] para este ambiente provido pelo ecossistema Spark/Hadoop. Além disso, possíveis abordagens de implementação no Apache Spark serão analisadas para as estratégias de despacho (*Dispatch*) [Ogasawara et al., 2011], que trata da execução de UDF regida por operadores em um fluxo de dados.

Referências

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM.

Cardoso, B., Matt, C., Furtado, H., and de Almeida, L. (2015). Creep damage evaluation in high-pressure rotor based on hardness measurement. *Journal of Materials Engineering and Performance*, 24(7):2784–2791.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Ferreira, J., Gaspar, D., Monteiro, B., Silva, A. B., Porto, F., and Ogasawara, E. (2017). Uma Proposta de Implementação de Álgebra de Workflows em Apache Spark no Apoio a Processos de Análise de Dados. In *Brazilian e-Science Workshop*.

Morcel, R., Ezzeddine, M., and Akkary, H. (2016). Fpga-based accelerator for deep convolutional neural networks for the spark environment. In *Smart Cloud (SmartCloud), IEEE International Conference on*, pages 126–133. IEEE.

Nothaft, F. A., Massie, M., Danford, T., Zhang, Z., Laserson, U., Yeksigian, C., Kottalam, J., Ahuja, A., Hammerbacher, J., Linderman, M., et al. (2015). Rethinking data-intensive science using scalable analytics systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 631–646. ACM.

Ogasawara, E., Dias, J., Oliveira, D., Porto, F., Valduriez, P., and Mattoso, M. (2011). An algebraic approach for data-centric scientific workflows. *Proc. of VLDB Endowment*, 4(12):1328–1339.

Rheinländer, A., Leser, U., and Graefe, G. (2017). Optimization of complex dataflows with user-defined functions. *ACM Computing Surveys (CSUR)*, 50(3):38.

Venkataraman, S., Yang, Z., Davies Liu, E. L., Falaki, H., Meng, X., Xin, R., Ghodsi, A., Franklin, M., Stoica, I., and Zaharia, M. (2016). Sparkr: Scaling r programs with spark.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.