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. *Large-scale data analysis has gained much importance in the scientific community due to the Big Data phenomenon. In this context, user-defined functions (UDFs) are commonly implemented in frameworks such as Apache Spark to enable large-scale data analysis. However, the use of UDF brings challenges in optimization of execution as they are opaque. This work proposes a method of optimizing data analysis workflows supported by UDF on Apache Spark. This method is based on SparkSQL's Catalyst API and Scala language macros.*

Resumo. *A análise de dados em larga escala tem ganhado muita importância na comunidade científica devido ao fenômeno do Big Data. Neste contexto, funções definidas pelo usuário (UDF) são, comumente, implementadas em frameworks como Apache Spark para viabilizar a análise de dados em larga escala. No entanto, o uso de UDF traz desafios no processo de otimização de execução pois são opacas. Este trabalho propõe um método de otimização de workflows de análise de dados apoiadas em UDF sobre o Apache Spark. Tal método é baseado na API Catalyst do SparkSQL e em macros da linguagem Scala.*

1. Introdução

A resolução de diversos problemas científicos é computacionalmente intensiva e otimizações voltadas às plataformas de hardware específicas (GPU e FPGA) e algoritmos e métodos matemáticos utilizados não são suficientes devido o aumento da quantidade de fontes de dados. Para se conduzir as análises de dados neste contexto, surgiram soluções para processamento de grandes volumes de dados escaláveis horizontalmente, tal como o modelo MapReduce, implementado em *frameworks* como Apache Spark [Zaharia et al., 2016].

Funções definidas pelo usuário (UDF) são comumente usadas nesses *frameworks* de modo a viabilizar a análise de dados em larga escala em experimentos científicos. Apesar de sua importância, o uso de UDF traz desafios no processo de otimização de execução, devido a dificuldade de se estabelecer semântica clara sobre seus comportamentos que possuem códigos relacionados ao domínio. Com a falta de conhecimento sobre a

*Os autores agradecem à FAPERJ, à CAPES e ao CNPq pelo financiamento do trabalho.

semântica da operação, o Spark é incapaz de otimizar a execução [Armbrust et al., 2015]. Este trabalho propõe um método para otimização das execuções de *workflows* de análise de dados, que possam ser escritas como UDF sobre o Apache Spark. Como primeiro estudo, avaliou-se a viabilidade do processo de otimização da execução de mapeamentos (operador *map*) operando sobre UDF usando a API *Catalyst* do Spark para abordar o problema da execução de atividades restritas (*Constrained Activity*)[1] [Ogasawara et al., 2011]. Uma avaliação preliminar aponta a viabilidade da abordagem.

Além desta introdução, o trabalho está organizado em mais três seções. Na seção 2, são apresentados os conceitos gerais necessários para o entendimento do problema e a abordagem adotada sob a forma de prova de conceito. A seção 3 traz a avaliação preliminar que corrobora com a hipótese. Por fim, a seção 4 apresenta as conclusões.

2. Prova de conceito

O Spark é um *framework* que possibilita a execução das tarefas paralelizáveis de forma distribuída em máquinas *multi-core* ou *clusters* YARN/Mesos/Kubernetes, com ênfase no processamento em *pipeline* das atividades que compõem um *dataflow* [Zaharia et al., 2016]. Ele 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]. A vantagem desta abordagem é deixar a responsabilidade sobre a complexidade do modelo de execução para o Spark. Isso diferencia a proposta de Ferreira et al. [2017] dos outros sistemas de gerenciamento de *workflow* (SGW). Desta forma, o foco se restringe a anotação semântica de UDF que tanto estabelece a relação de consumo e produção de ativações [Ogasawara et al., 2011] quanto na associação com a proveniência para prover informação necessária a otimização do *workflow*.

Nesta prova de conceito foi realizada uma análise exploratória das APIs do *Catalyst* pertencente ao módulo *SparkSQL* e dos seus pontos de extensão. O *SparkSQL* usa o componente *Catalyst* para otimizar a geração de código em tempo de execução à partir de expressões e comandos SQL, e os pontos de extensão do Spark possibilitam a personalização do ambiente com o uso de novas regras para otimização de *dataflow* de acordo com o domínio do problema.

Na álgebra de *workflow*, o operador *wMap* é definido: $T \leftarrow wMap(A, R)$, onde A é a atividade que 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. Portanto, é possível observar que a semântica do operador *wMap* é compatível com seu análogo *map* do *SparkSQL* [Ogasawara et al., 2011; Armbrust et al., 2015]. Considere o seguinte exemplo especificado pela álgebra de *workflow*: $T \leftarrow wMap(B, wMap(A, R))$. Ao implementar este *dataflow* no Spark, o *Catalyst* do *SparkSQL* usará a regra *CollapseProject* na fase de otimização. Esta regra junta as duas expressões dos dois *map* em uma, executando A e B em *pipeline*. Isso pode ser indesejado no caso em que estas atividades forem escritas em código nativo e usarem muitos recursos do hardware. Isto caracteriza, segundo Ogasawara et al. [2011], a existência de uma atividade restrita (*Constrained Ac-*

[1] Nas aplicações científicas é muito comum invocar programas com implementações paralelas que usam todos os núcleos (*cores*) disponíveis em um nó do *cluster*. Devido a essa característica, a ativação de uma atividade restrita bloqueia todos os núcleos, inibindo a utilização por outras ativações. Ogasawara et al. [2011] chama este fenômeno de atividade restrita (*Constrained Activity*).

tivity), onde uma atividade, regida por um operador *wMap* e executada em um *dataflow*, pode consumir muitos recursos computacionais impedindo que outras atividades também regidas por um operador *wMap* possam ser executadas em *pipeline*. Para solucionar isso, Ogasawara et al. [2011] propõem a criação de uma barreira forçando a materialização do resultado intermediário entre a execução das atividades.

Com o objetivo de otimizar este tipo de *workflow*, foi criada uma classe[2] escrita em linguagem Scala para funcionar como ponto de extensão e otimizar operadores *map*. Além dos pontos de extensão do *SparkSQL*, esta classe usa as funcionalidades *quasi-quotes* e macros da linguagem Scala para atuar no *dataflow* antes mesmo de passar o plano lógico da *query* ao *Catalyst*. Ela consiste de uma *case class* do Scala e é responsável por transformar planos lógicos por meio de otimizações baseadas em transformações algébricas. O objetivo desta classe é incluir uma barreira de materialização entre dois operadores *map* adjacentes. Neste caso, isso só é possível se os operadores atuarem sobre UDF anotadas com semântica de utilização excessiva de recursos em tempo de execução, informações estas obtidas da proveniência.

3. Avaliação Preliminar

Nesta seção, procurou-se avaliar a capacidade do Spark em otimizar um *dataflow* com UDF anotada com informações sobre uso excessivo de recursos usando o operador $wMap$, como ilustrado abaixo em código Scala:

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

O Spark combina, por padrão, os processamentos dessas duas UDF em *pipeline*, pois desconhece a semântica da operação executada e os recursos computacionais consumidos. Caso estas UDF sejam escritas em código nativo, cada uma delas pode consumir todos os recursos de *thread* ou memória do processador de forma que não possam ser invocadas em *pipeline*. Conforme já mencionado, isto caracteriza uma atividade restrita (*Constrained Activity*) e a solução é a criação de uma barreira forçando a materialização do resultado entre a avaliação das duas UDF.

A anotação semântica pode ser feita via sufixo indicando a quantidade de memória RAM utilizada em média ou o número de *threads*/*cores* do processador, ou ambos. Esta anotação é usada na fase inicial 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 dos segmentos. As materializações no Spark podem ser provocadas por operadores *collect* aplicados no *dataset*. Esta abordagem foi escolhida, por modificar apenas a representação do dado, preservando o esquema. O importante é que a criação da barreira não modifica a semântica do *dataflow* original.

Para uma avaliação preliminar, a criação de barreira foi aplicada sobre o *workflow* mostrado na Figura 1(a). Nele, tem-se uma relação `R0` onde são aplicados dois operadores *map* em sequência. O primeiro operador rege a udfA que após anotação semântica se transformou em udfA_M3G. O segundo operador rege a udfB que anotado se transformou em udfB_M3G. O sufixo M3G significa que o processo externo invocado pela UDF usa

[2]O código fonte Scala pode ser acessado em https://github.com/joao-parana/wff-catalyst

em média 3GB de memória RAM para cada tupla consumida. Supondo que o subsistema de proveniência tenha informado que o sistema sofrerá com problemas de desempenho ou confiabilidade nestas condições, o *dataflow* pôde ser modificado para materializar o resultado do primeiro *map*. A Figura 1(b) mostra esta situação. Uma relação `R1` foi criada com o propósito de materializar o resultado intermediário para que o Spark não realize a otimização padrão para dois operadores *map* adjacentes. Com isso, o *dataflow* deixou de ser executado em *pipeline* permitindo sua finalização, o que não seria possível na versão não original, pois o programa terminaria logo no início por falta de recursos computacionais disponíveis.

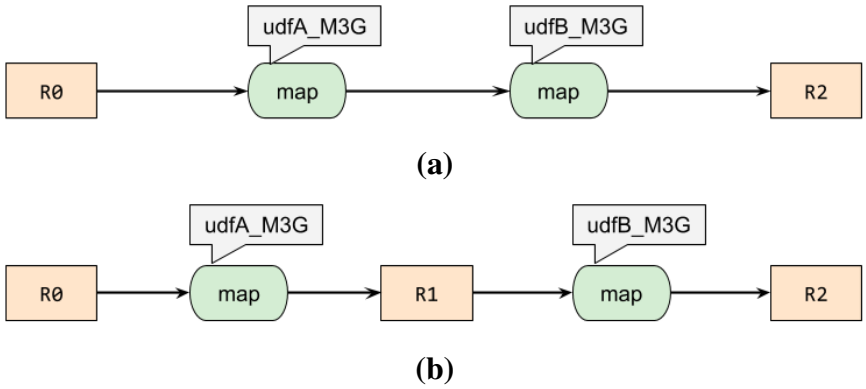

Figura 1. *workflow* antes (a) e depois (b) da inclusão da barreira.

4. Considerações finais

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 presente trabalho aponta 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. Isso inclui o caso da atividade restrita (*Constrained Activity*) com atividades legadas escritas em código nativo (C++, Fortran, Python, Cython, R, etc.) [Ogasawara et al., 2011].

Referências

Armbrust, M., Xin, R., Lian, C., Huai, Y., Liu, D., Bradley, J., Meng, X., Kaftan, T., Frankliny, M., Ghodsi, A., and Zaharia, M. (2015). Spark SQL: Relational data processing in spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 2015-May, pages 1383–1394.

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*.

Ogasawara, E., de Oliveira, D., Valduriez, P., Dias, J., Porto, F., and Mattoso, M. (2011). An algebraic approach for data-centric scientific workflows. In *Proceedings of the VLDB Endowment*, volume 4, pages 1328–1339.

Zaharia, M., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., and Venkataraman, S. (2016). Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.