

# Otimização de Parâmetros em Aplicações de *Big Data* Baseadas em Múltiplos *Frameworks*\*

Bruna de Mello Almeida<sup>1</sup>, Yuri Frota<sup>1</sup>, Daniel de Oliveira<sup>1</sup>

<sup>1</sup>Instituto de Computação – Universidade Federal do Fluminense (UFF)

brmello@id.uff.br, yuri@ic.uff.br, danielcmo@ic.uff.br

**Abstract.** Database management systems and distributed computing frameworks are crucial for applications that process large volumes of data. Configuring them manually is complex due to the number and interdependence of parameters both intra- and inter-frameworks. Current automatic solutions require many examples and do not optimize system integration. This paper evaluates a model-independent approach to optimize parameters from Apache Spark and Cassandra in an integrated way. The results show performance improvements of up to 69.99% with the integrated parameter optimization compared to the default parameter values.

**Resumo.** Os sistemas de gerência de banco de dados e os frameworks de computação distribuída são cruciais para aplicações que processam grandes volumes de dados. Configurá-los manualmente é complexo devido à quantidade e interdependência dos parâmetros tanto em um mesmo framework quanto entre frameworks. As soluções automáticas atuais necessitam de muitos exemplos e não otimizam a integração entre sistemas. Este artigo avalia uma abordagem independente de modelo para otimizar parâmetros do Apache Spark e Cassandra de forma integrada. Os resultados mostram melhorias de até 69,99% com a otimização dos parâmetros de forma integrada, em comparação com os valores default de parâmetros.

## 1. Introdução

Nos últimos anos, temos presenciado um aumento considerável da computação centrada em dados [Jin et al. 2024, de Oliveira et al. 2019], especialmente devido ao grande volume de dados disponível, fator este que contribuiu para o surgimento do cenário de *Big Data*. Esses dados são comumente multimodais, *i.e.*, estruturados, textuais, imagens, vídeos, *etc.*, e podem ser utilizados como entrada em diversos tipos de aplicações, desde o treinamento de modelos de Aprendizado de Máquina até complexas simulações científicas [Pina et al. 2024]. No entanto, processar e extrair informações úteis desses dados pode não ser uma tarefa trivial, já que em muitos casos exige o desenvolvimento de *pipelines* de forma muito diferente do modelo tradicional que considera um processamento não distribuído em sistemas *on-premise*<sup>1</sup> [Zaharia 2019].

\*O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001. Os autores gostariam de agradecer também a CNPq e FAPERJ.

<sup>1</sup>O termo *on-premise* no cenário de *big data* está relacionado ao uso de recursos computacionais locais e limitados, como a infraestrutura de *software* e *hardware* do computador pessoal.

As aplicações precisam obter, pré-processar, agregar e armazenar grandes volumes de dados de maneira eficiente e dentro de um prazo viável. Um exemplo desse tipo de aplicação são os *pipelines* de ETL (Extração, Transformação e Carga) que transferem dados externos para um *Data Warehouse* (DW) ou um *Data Lakehouse* (DL) [Haase et al. 2022]. Para atingir esse objetivo, esses *pipelines* são desenvolvidos utilizando vários *frameworks* existentes para processamento distribuído (*e.g.*, Apache Spark e Hadoop), orquestração e escalonamento (*e.g.*, Apache Airflow), e mensageria de dados (*e.g.*, Kafka), além de Sistemas de Gerência de Bancos de Dados (SGBDs) como o PostgreSQL e o Cassandra. O uso desses *frameworks* e SGBDs garante escalabilidade e confiabilidade à aplicação. Embora já existam metodologias propostas para desenvolver aplicações que integram diversos *frameworks* e SGBDs [LeFevre et al. 2016], elas não consideram um ponto crítico no seu funcionamento: a definição da melhor configuração de seus parâmetros.

Cada *framework* e SGBD pode expor centenas de parâmetros para o desenvolvedor configurar (*e.g.*, o Spark possui mais de 180 parâmetros). Devido ao grande número de parâmetros e à correlação direta entre eles, a configuração manual (mesmo considerando os usuários especialistas) se mostra complexa e propensa a erros [Yu et al. 2018]. Embora as configurações *default* de cada *framework* e SGBD ofereçam um desempenho razoável, independentemente da aplicação e da infraestrutura de *hardware* onde a aplicação executará, elas são definidas para cada *framework* isoladamente. No caso de uma aplicação que integra múltiplos *frameworks* e/ou SGBDs, pode haver uma correlação entre parâmetros *inter-framework/SGBD* e não somente *intra-framework/SGBD*. Por exemplo, se uma aplicação processa dados com o Spark e armazena o resultado do processamento no SGBD Cassandra, o número de *executors*<sup>2</sup> do Spark deve ser configurado de acordo com a quantidade de escritas concorrentes definidas no Cassandra para oferecer um melhor desempenho.

Existem diversos trabalhos na literatura que buscam auxiliar na escolha da melhor configuração de parâmetros para esses *frameworks* e SGBDs, mas não de forma integrada. Muitos deles [Popescu et al. 2013, de Oliveira et al. 2021, Huang et al. 2022, Sharma et al. 2018, Zhang et al. 2021] avaliam um conjunto de configurações representativas (*i.e.*, exemplos de configurações que produzem bom desempenho da aplicação) no espaço de parâmetros para treinar modelos de Aprendizado de Máquina. Uma das desvantagens dessas abordagens que precisam de dados para treinamento, é que a quantidade de exemplos necessária para treinar um modelo pode ser grande, principalmente devido à natureza estocástica de algumas aplicações e às diversas configurações de *hardware* possíveis, que podem influenciar o desempenho. Uma pequena quantidade de exemplos de configurações pode não ser suficiente para treinar um modelo realmente útil [Zhu et al. 2017]. Outros trabalhos [Lama and Zhou 2012, Essertel et al. 2018, Oliveira et al. 2022, Silva-Muñoz et al. 2021] focam em modelar a escolha da configuração como um problema de otimização. Diferentemente das abordagens baseadas em modelos de Aprendizado de Máquina, as abordagens baseadas em otimização não requerem um grande número de exemplos, mas pode ser muito complexo (ou mesmo inviável) modelar todas as características dos *frameworks* e da infraestrutura de execução. Apesar de representarem um avanço, as abordagens supracitadas não foram projetadas para tratar de aplicações que integram múltiplos *frameworks/SGBDs*.

De forma a não depender de um grande número de exemplos e nem precisar mo-

---

<sup>2</sup><https://spark.apache.org/docs/latest/cluster-overview.html>

delar um problema com características inerentemente complexas, é proposta neste artigo uma abordagem chamada *Marin*<sup>3</sup> (do inglês *Multiple frAmewoRks confIguratiOn eNginE*) que aplica um otimizador independente de modelo, baseado no algoritmo de corrida iterada [Maron and Moore 1997], descrito na Seção 2.1, para definir a melhor configuração de parâmetros para aplicações que utilizam múltiplos *frameworks*/SGBDs. A ideia é otimizar o conjunto de parâmetros de diferentes *frameworks* utilizados na aplicação de forma integrada, considerando as correlações entre os parâmetros intra e inter-*framework*. O algoritmo de corrida iterada testa um conjunto de configurações em paralelo, descarta de forma otimizada as configurações claramente inferiores e concentra o esforço computacional em diferenciar as melhores configurações. Embora existam diversas ferramentas que implementam esse algoritmo, utilizou-se o *Irace* [López-Ibáñez et al. 2016] neste artigo por ser uma ferramenta resiliente, apresentar flexibilidade nas configurações disponíveis e ser transparente quanto aos resultados intermediários e finais. A abordagem proposta foi avaliada por meio de um estudo de caso com um *pipeline* de ETL que foi construído sobre o Spark e o Cassandra. Os resultados mostram melhorias de até 69,99% comparado à configuração *default* dos *frameworks*/SGBD.

O presente artigo se encontra organizado em cinco seções além da Introdução. A Seção 2 discute o referencial teórico e os trabalhos relacionados ao tópico do artigo. A Seção 3 define formalmente o problema tratado no presente artigo. A Seção 4 apresenta a abordagem *Marin*. A Seção 5 apresenta a avaliação experimental, incluindo a configuração do banco de dados, a configuração do *Irace* e os experimentos realizados. Finalmente, a Seção 6 conclui o artigo e discute os trabalhos futuros.

## 2. Referencial Teórico e Trabalhos Relacionados

Nessa seção, inicialmente discutimos de forma breve o otimizador *Irace*, utilizado na abordagem *Marin*. A seguir, discutimos os trabalhos relacionados.

### 2.1. O Otimizador *Irace*

O *Irace* [López-Ibáñez et al. 2016] é um otimizador independente de modelo que implementa o algoritmo de corrida iterada [Maron and Moore 1997]. A partir de um conjunto de instâncias, ou seja, possíveis configurações de parâmetros, ele determina a melhor configuração para esse conjunto. Em cada iteração, o desempenho da aplicação é avaliado para um conjunto específico de configurações de parâmetros na infraestrutura computacional alvo, *i.e.*, o *hardware* em que a aplicação executará. Se uma configuração demonstrar um bom desempenho, ela permanece na corrida; caso contrário, é descartada. Após  $n$  iterações, as soluções não descartadas são classificadas e uma nova corrida é iniciada. Essas etapas são repetidas até que um critério de parada seja alcançado, comumente envolvendo um tempo máximo (*deadline*), número de corridas, entre outros.

Para que a escolha da melhor configuração de uma aplicação possa ser realizada com o *Irace*, o usuário precisa definir os seguintes parâmetros: *TargetRunner*, *lista de parâmetros* (com seus tipos e valores possíveis), um conjunto de *instâncias* (para avaliar as configurações) e um *Budget*. O *TargetRunner* é a aplicação em que a configuração será aplicada. No contexto desse artigo, uma aplicação que utilize múltiplos *frameworks*/SGBD. A *lista de parâmetros* contém os parâmetros associados ao

---

<sup>3</sup>*Marin* é também o nome de uma Amazona de Prata do anime “Cavaleiros do Zodíaco”.

*frameworks*/SGBD, enquanto que o conjunto de *instâncias* é representado por diferentes combinações de arquivos a serem processados por uma determinada aplicação. Finalmente, o *Budget* define a quantidade de corridas (iterações) que serão realizadas no algoritmo de corrida iterada. O *Budget* é o parâmetro mais difícil de ser definido, uma vez que ele define quantas vezes o *Trace* executará o algoritmo de corrida iterada. Seu valor depende diretamente do espaço de parâmetros e da heterogeneidade das instâncias fornecidas. É importante ressaltar que o *Trace* não indica se um determinado valor de *Budget* será capaz de gerar todas as possíveis configurações.

## 2.2. Trabalhos Relacionados

A configuração automática de parâmetros para *frameworks* que processam grandes volumes de dados é um tópico amplamente estudado devido à sua importância nas aplicações modernas. No entanto, a maioria dos estudos sobre identificação da configuração de parâmetros que proporciona o melhor desempenho concentra-se em otimizar SGBDs e *frameworks* de processamento distribuído de forma isolada, desconsiderando a possibilidade de dependência de parâmetros entre diferentes *frameworks*, *i.e.*, *inter-framework*.

Algumas abordagens focam em encontrar a configuração otimizada no contexto do *framework* Hadoop. Apesar de ser um *framework* menos eficiente do que o Spark, o Hadoop ainda é muito utilizado na indústria. O AROMA [Lama and Zhou 2012] seleciona os melhores valores de parâmetros para alocação de recursos em *clusters* Hadoop enquanto reduz os custos financeiros. O AROMA é baseado em modelos de Aprendizado de Máquina que classificam os *jobs* Hadoop por meio de algoritmos de clusterização, usando informações relacionadas à utilização de CPU, rede e disco. O PREDICT [Popescu et al. 2013] realiza previsões em tempo de execução para algoritmos iterativos intensivos em E/S implementados no Hadoop. De forma similar ao AROMA, é necessário um grande conjunto de dados de treinamento no PREDICT. É importante ressaltar que quando não há histórico de dados disponível, pode ocorrer uma configuração ineficiente do *framework*.

No contexto da escolha da configuração para o Spark, [Essertel et al. 2018] propõem o Flare, uma abordagem aceleradora para o Spark que oferece melhorias de desempenho de uma ordem de magnitude em arquiteturas com escalabilidade vertical. Inspirado por técnicas de compilação de consultas de SGBDs em memória principal, o Flare incorpora uma estratégia de geração de código projetada para corresponder aos aspectos únicos do Spark e às características das arquiteturas de escalabilidade vertical. Similarmente, [de Oliveira et al. 2021] propõem o SPACE, uma abordagem de configuração de parâmetros do Spark baseada no treinamento de modelos de Aprendizado de Máquina, em especial as árvores de decisão e as florestas randômicas. O SPACE coleta dados de execuções passadas de *pipelines* e treina os modelos baseando-se nas características do ambiente e do *pipeline* que será executado. Já [Huang et al. 2022] propõem uma abordagem baseada em Aprendizado por Reforço para buscar a melhor configuração de parâmetros para aplicações baseadas no Spark. A vantagem do Aprendizado por Reforço é que não é necessária uma grande quantidade de dados de treinamento, mas o algoritmo, baseado no tradicional *Q-Learning*, pode demorar muito para convergir para uma solução.

A otimização das configurações em SGBDs é um tópico altamente pesquisado nas últimas décadas [Mozaffari et al. 2024]. Existem diversos tipos de otimizações que podem ser implementadas em um SGBD, mas nesse artigo o interesse é na otimização do

ponto de vista de integração entre o SGBD e outro *framework*, considerando elementos como a volumetria a ser armazenada e a quantidade de escritas concorrentes, sem levar em conta as características específicas do *schema* e consultas que serão executadas. Nesse sentido, [Sharma et al. 2018] propõem uma abordagem chamada NoDBA que faz uso de Aprendizado por Reforço para definição das melhores configurações de parâmetros no PostgreSQL. A partir do *benchmark* TPC-H, a abordagem NoDBA explora o espaço de configurações possíveis e analisa o desempenho de cada uma das 22 consultas do TPC-H para as configurações. Similarmente, [Zhang et al. 2021] propõem o uso de Aprendizado por Reforço para otimização de parâmetros globais do SGBD. A abordagem proposta por [Zhang et al. 2021] utiliza recompensas/penalizações para avaliar cada uma das possíveis configurações.

Diferentemente das abordagens anteriormente citadas de otimização de SGBDs, [Oliveira et al. 2022] e [Silva-Muñoz et al. 2021] não utilizam técnicas de Aprendizado de Máquina. A ideia de [Oliveira et al. 2022] é combinar múltiplas técnicas existentes de otimização de parâmetros para produzir uma otimização global mais eficiente do SGBD. A abordagem proposta por [Oliveira et al. 2022] não necessita de um grande volume de exemplos para definir a melhor configuração. A abordagem proposta por [Silva-Muñoz et al. 2021] também não é dependente de uma gama de exemplos para realizar a escolha da configuração. Similarmente a *Marin*, a abordagem proposta por [Silva-Muñoz et al. 2021] utiliza a otimização baseada no algoritmo de corrida iterativa, mas só considera os parâmetros do SGBD Cassandra. Em suma, embora representem avanços significativos, as abordagens supracitadas focam exclusivamente na escolha da configuração para um SGBD ou *framework* específico, desconsiderando que as aplicações podem utilizar outros *frameworks* e que pode haver uma dependência de parâmetros inter-*framework*/SGBD.

### 3. Formulação do Problema

Nesta seção, o problema de escolha da melhor configuração dos parâmetros de múltiplos *frameworks* de forma integrada é formalmente definido. Neste artigo, nos baseamos no formalismo proposto por [Teylo et al. 2017], onde uma aplicação é modelada como um grafo acíclico direcionado  $A = (M, Dep)$ , onde  $M$  é o conjunto de métodos executados pela aplicação (vértices), como os métodos de agregação e junção utilizados no processamento de dados, e  $Dep$  é um conjunto de arcos que representa as dependências de dados entre os métodos em  $M$ . Dado um método  $m_i \in M$ , seja  $E(m_i)$  seu conjunto de dados de entrada e  $S(m_i)$  o conjunto de dados de saída produzidos pelo método  $m_i$ . Uma dependência entre os métodos  $m_i$  e  $m_j$  é denotada por  $(m_i, m_j) \in Dep \leftrightarrow \exists e_k \in E(m_j) | e_k \in S(m_i)$ .

Em relação ao *hardware* em que a aplicação é executada, seja  $R = \{r_1, \dots, r_k\}$  o conjunto de máquinas disponíveis para a execução dedicada da aplicação. Portanto, dada uma aplicação  $A$ , um conjunto de dados de entrada  $E$  e um conjunto de recursos computacionais  $R$ , seja  $P(A, E, R) = \{pv_1, pv_2, \dots, pv_m\}$  o conjunto de valores de parâmetros dos *frameworks*/SGBDs utilizados pela aplicação  $A$ . Cada valor de parâmetro  $pv_i$  representa um dos parâmetros dos *frameworks*, como por exemplo: o número de núcleos ou a quantidade de memória usada por cada Spark *executor*, ou a quantidade de escritas concorrentes no Cassandra, *etc*.

Assim, dada uma aplicação  $A$  e seu conjunto de dados de entrada  $E$  a ser executado em um *hardware*  $R$ , o objetivo da abordagem proposta neste artigo é encontrar o conjunto

de valores de configuração de parâmetros  $P^*(A, E, R)$  de modo que o desempenho de  $A$  seja maximizado, o que é equivalente a minimizar o tempo de execução de  $A$ . Mais formalmente,  $\exists P^*(A, E, R)$  tal que:

$$TE(A, E, R, P^*(A, E, R)) = \min_{\forall P(A, E, R)} TE(A, E, R, P(A, E, R)) \quad (1)$$

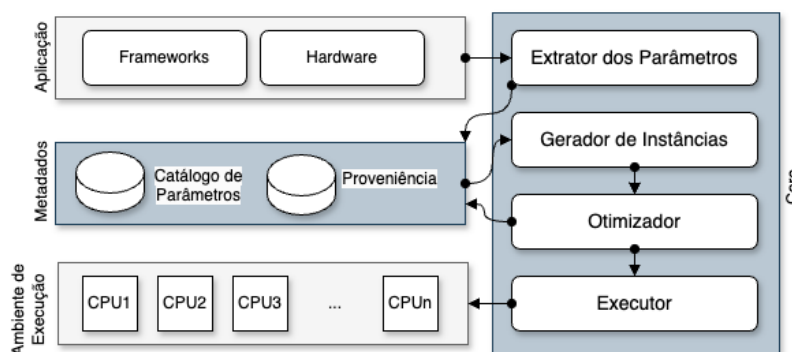
onde  $TE(A, E, R, P(A, E, R))$  é o tempo de execução da aplicação  $A$  processando os dados em  $E$  nos recursos de *hardware* disponíveis  $R$  com o conjunto de valores de parâmetros  $P(A, E, R)$ . No contexto deste artigo, o  $P^*(A, E, R)$  é obtido por meio da execução do otimizador *Irace*.

#### 4. A Abordagem Marin

A arquitetura da *Marin* para a otimização de parâmetros de múltiplos *frameworks* de forma integrada é composta por seis componentes principais, conforme apresentado na Figura 1: (i) Extrator de Parâmetros, (ii) Gerador de Instâncias, (iii) Otimizador, (iv) Executor, (v) Catálogo de Parâmetros, e (vi) Repositório de Proveniência. Além disso, os componentes são organizados em duas camadas: *Core*, que contém os componentes de (i) a (iv), e *Metadados*, que contém os componentes (v) e (vi).

A execução da *Marin* se inicia com o componente *Extrator de Parâmetros* dos *frameworks* utilizados na aplicação. Como cada aplicação pode utilizar diferentes *frameworks*, diferentes conjuntos de parâmetros podem ser identificados, *i.e.*, múltiplos  $P(A, E, R)$ . Nesta etapa, todos os parâmetros de todos os *frameworks* são identificados e salvos em um *Catálogo de Parâmetros*. Esse catálogo inclui os nomes dos parâmetros, *e.g.*, *spark.executor.instances*, e os possíveis valores que cada parâmetro pode assumir, *e.g.*, 1 a 8. Os possíveis valores podem ser informados pelo usuário ou retornados de execuções passadas da mesma aplicação por meio de consultas no *Repositório de Proveniência*. Além dos parâmetros dos *frameworks*, o *Extrator de Parâmetros* também identifica as características do *hardware* em que a aplicação será executada, *i.e.*,  $R = \{r_1, \dots, r_k\}$ . Essa informação é importante porque, dependendo do *hardware* disponível, a escolha de valores para um parâmetro pode possuir alguma restrição. Por exemplo, se a aplicação for executada em uma máquina com 8 núcleos, ao definir o parâmetro *spark.executor.cores*, o limite superior deve ser 8.

Uma vez que os parâmetros dos *frameworks* e as características do *hardware* foram identificados e catalogados, o componente *Gerador de Instâncias* pode ser invocado. Ele é responsável por gerar um conjunto de instâncias que serão avaliadas pelo *Otimizador*. Essas instâncias contêm um subconjunto dos parâmetros e um conjunto de valores associados, obtidos a partir dos possíveis valores de cada parâmetro no catálogo. Após a geração das instâncias, o *Otimizador* é invocado. Na versão atual, foi utilizado o *Irace* como otimizador. Conforme discutido na Seção 2.1, antes de executar o *Irace*, o *Otimizador* deve gerar um arquivo de cenário a partir das instâncias previamente geradas. Em seguida, o *Irace* é invocado e retorna as  $n$  melhores configurações de acordo com o algoritmo de corrida iterada. Essas configurações são armazenadas no *Repositório de Proveniência* para consultas futuras. Finalmente, o *Executor* recebe a melhor configuração das  $n$  retornadas pelo *Irace*, *i.e.*,  $P^*(A, E, R)$ , e executa no *hardware* disponível, coletando o tempo real de execução da aplicação e atualizando o *Repositório de Proveniência* com



**Figura 1. Arquitetura da Abordagem Marin. Os componentes são organizados em duas camadas: (i) Core, que contém as principais funcionalidades e (ii) Metadados, que armazena o catálogo de parâmetros e os dados de proveniência.**

essa informação. A Marin é uma abordagem de código aberto e será disponibilizada no GitHub <https://github.com/UFFeScience/marin>.

## 5. Avaliação Experimental

Para a avaliação da abordagem Marin, foram realizados diversos experimentos utilizando como estudo de caso um *pipeline* de ETL com o objetivo de identificar e sumarizar o número de menções a determinadas pessoas em textos obtidos na internet. Esse *pipeline* é executado múltiplas vezes ao longo do dia, visto que a produção de conteúdo *online* é contínua, consumindo um grande volume de dados, o que torna a sua configuração uma tarefa prioritária. O *pipeline* utiliza o *framework* Apache Spark para execução distribuída e o Cassandra para armazenar os resultados agregados. Inicialmente na seção, o *pipeline* de ETL é detalhado. Em seguida, as configurações do ambiente e dos experimentos são apresentadas. Finalmente, os resultados experimentais são discutidos.

### 5.1. Estudo de Caso

A aplicação escolhida como estudo de caso é um *pipeline* de ETL que tem como objetivo contar a quantidade de menções a determinadas pessoas em textos obtidos na internet. Esse *pipeline* possui um método de contagem de palavras em textos similar ao tradicional *Word Count*<sup>4</sup>, porém com filtros para contagens de palavras específicas. A Figura 2 apresenta os principais métodos do *pipeline*. A aplicação realiza uma varredura no diretório informado pelo usuário para encontrar os arquivos no formato *txt* que serão processados e armazena os nomes dos arquivos em uma lista. Após o carregamento da lista, um *dataframe* vazio chamado  $df_1$  é criado.

Uma estrutura de repetição itera sobre a lista dos arquivos a serem processados, e a cada iteração, o conteúdo de um arquivo é armazenado em um RDD<sup>5</sup> (do inglês *Resilient Distributed Dataset*) chamado  $RDD_1$ . Inicialmente, uma transformação é aplicada para remover caracteres especiais e o resultado é armazenado em um novo RDD chamado  $RDD_2$ . Após, os dados no  $RDD_2$  são submetidos ao processo de contagem de palavras que foi construído utilizando-se duas transformações e uma *action* do Spark: (i) *flatMap*,

<sup>4</sup><https://spark.apache.org/examples.html>

<sup>5</sup>Um RDD é uma coleção imutável de elementos de dados distribuídos em múltiplas máquinas.

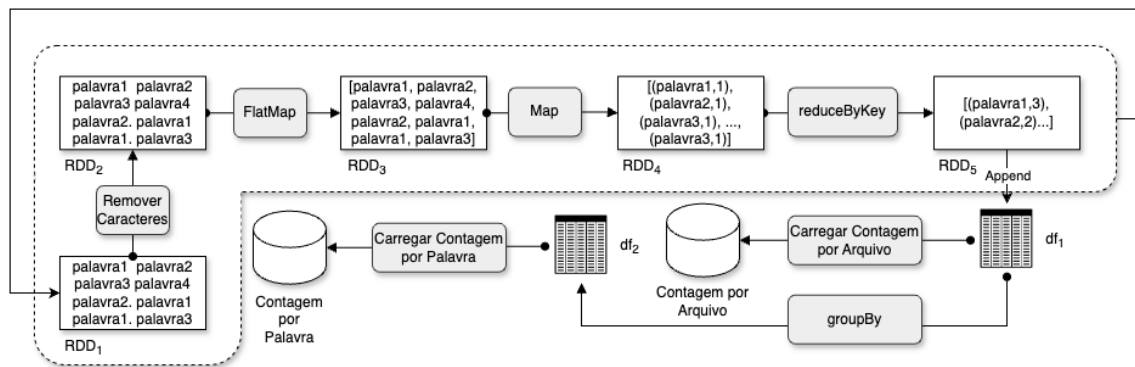


Figura 2. O pipeline de ETL escolhido como estudo de caso.

(ii) *map* e (iii) *reduceByKey*. O primeiro é responsável pela *tokenização* das palavras do texto e gera o  $RDD_3$ , enquanto os outros dois são encarregados de adicionar a contagem unitária (gerando o  $RDD_4$ ) e realizar a soma das contagens por palavra (gerando o  $RDD_5$ ), respectivamente.

Os resultados produzidos em cada iteração são adicionados ao *dataframe*  $df_1$  por meio de uma operação de *append*. Ao final do processamento da lista de arquivos, o conteúdo do *dataframe*  $df_1$  é armazenado em uma tabela no SGBD Cassandra com quatro atributos: (i) *index* - atributo do tipo *int* que representa a chave primária, (ii) *source* - atributo do tipo *text* que representa o nome do arquivo processado, (iii) *word* - atributo do tipo *text* que representa a palavra extraída do arquivo, e (iv) *count* - atributo do tipo *int* que representa a contagem total daquela palavra no arquivo.

Como o *dataframe*  $df_1$  possui a contagem de palavras apenas por arquivo, e o *pipeline* tem como objetivo realizar a contagem global para toda a coleção de documentos, um *groupBy* é executado, dessa vez sem considerar o atributo de nome do arquivo. O resultado da contagem global é armazenado no *dataframe*  $df_2$  e posteriormente carregada em uma tabela no Cassandra que segue a seguinte estrutura: (i) *word* - atributo do tipo *text* que representa a palavra extraída do arquivo, e (ii) *count* - atributo do tipo *int* que representa a contagem total daquela palavra no arquivo.

## 5.2. Configuração do Ambiente e dos Experimentos

Para a execução dos experimentos apresentados nesta seção, a *Marin* e seus componentes foram implantados em um servidor equipado com dois processadores Intel Core i9-12900. Cada processador possui 16 núcleos, totalizando 32 núcleos de processamento no servidor. Além disso, o servidor conta com 128 GB de RAM e 10 TB de disco. Para o Cassandra, foi criado um banco de dados local contendo um *keyspace* e as duas tabelas com a estrutura mencionada anteriormente. A configuração do *keyspace* utilizou a estratégia de replicação *SimpleStrategy* com um fator de replicação unitário.

Para avaliar a abordagem *Marin*, foram realizados cinco experimentos distintos. No primeiro experimento ( $Ex_1$ ), o *pipeline* do estudo de caso foi executado com as configurações *default* dos *frameworks*, *i.e.*, Spark e Cassandra. No segundo experimento ( $Ex_2$ ), foram considerados apenas os parâmetros do Spark para a otimização, mantendo os parâmetros do Cassandra com valores *default*. No terceiro experimento ( $Ex_3$ ), apenas os parâmetros do Cassandra foram considerados e otimizados, mantendo os parâmetros do



Spark com valores *default*. No quarto experimento ( $Ex_4$ ) consideramos a combinação dos valores de parâmetros obtidos com as otimizações do Spark e Cassandra de forma isolada. Finalmente, no quinto experimento ( $Ex_5$ ), os parâmetros dos dois *frameworks* foram otimizados de forma integrada. As otimizações referentes aos experimentos descritos foram realizadas modificando-se o arquivo de parâmetros do *Irace* e mantendo todo o restante do cenário com as mesmas configurações.

Apesar de o Spark e o Cassandra possuírem mais de 300 parâmetros em conjunto, nos experimentos realizados consideramos os 13 parâmetros mais significativos. Esses parâmetros foram obtidos a partir de execuções de testes preliminares realizados com o próprio *Irace*, que consideraram tanto as configurações de *hardware*, *e.g.*, memória e quantidade de núcleos, quanto os métodos utilizados no *pipeline* de dados, como o *map* e *flatMap*. A Tabela 1 apresenta cada um dos 13 parâmetros considerados, associados a um identificador (ID), seu tipo (i - inteiro, r - real, c - categórico), a faixa de valores considerada pelo *Gerador de Instâncias* e os valores *default* de cada parâmetro. Os parâmetros de P1 a P8 são denominados *Parâmetros Spark*, enquanto os parâmetros de P9 a P13 são denominados *Parâmetros Cassandra*.

**Tabela 1. Parâmetros escolhidos para otimização usando a abordagem *Marin*.**

ID	Parâmetro	Tipo	Faixa de Valores	Default
P1	spark.executor.memory	i	(1,128)	1
P2	spark.executor.cores	i	(1,8)	1
P3	spark.executor.instances	i	(2,128)	2
P4	spark.sql.shuffle.partitions	c	(200, 300, 400, 500)	200
P5	spark.default.parallelism	i	(1,8)	8
P6	spark.storage.memoryFraction	r	(0, 0.9)	0.6
P7	spark.shuffle.compress	c	(true, false)	true
P8	spark.sql.sources.partitionOverwriteMode	c	(dynamic, static)	static
P9	cassandra.output.consistency.level	c	(ANY, LOCAL_ONE)	LOCAL_ONE
P10	cassandra.input.split.sizeInMB	c	(16, 32, 64, 128, 256, 512, 1024, 2056)	64
P11	cassandra.output.batch.size.rows	c	(auto, 100, 500, 1000, 2000, 2500, 3000)	auto
P12	cassandra.output.batch.grouping.buffer.size	c	(1000, 2000, 3000, 4000, 5000)	1000
P13	cassandra.output.concurrent.writes	c	(5, 10, 15, 20)	5

Finalmente, para configurar o *Irace*, definimos como instâncias do problema a execução do *pipeline* consumindo 300 arquivos de texto. O *TargetRunner* foi configurado como o *pipeline* em questão, e o *Budget* máximo foi estabelecido em 1.000 iterações. Foi definido também que seriam analisadas as *top 4* configurações fornecidas pelo *Irace* para cada experimento. Finalmente, foi estabelecido que a configuração inicial a ser explorada pelo *Irace* seria aquela com os valores *default* de todos os parâmetros, conforme apresentado na Tabela 1.

### 5.3. Discussão dos Resultados

As Tabelas 2 e 3 apresentam as quatro melhores configurações obtidas pelo *Irace* considerando apenas os parâmetros do Spark e do Cassandra de forma isolada, respectivamente. É possível observar que, mesmo ao considerar cada *framework* e SGBD de forma isolada na escolha das configurações, sete dos treze parâmetros foram configurados com valores diferentes dos padrões e com valores não triviais para a escolha humana. Por exemplo, a melhor configuração do Spark define que a quantidade de memória por *executor* (parâmetro P1) deve ser de 12GB, o que não é uma escolha comum (geralmente são escolhidos valores em potências de dois). O mesmo vale para o melhor valor do número de instâncias de *executors* (Parâmetro P3), que define 113 instâncias de *executors*. Já na melhor configuração do

Cassandra, o *Irace* reduziu o tamanho máximo da partição (Parâmetro P10) de 64MB para 32MB, e aumentou a quantidade de escritas concorrentes (Parâmetro P13) de 5 para 20. É importante ressaltar que as configurações obtidas pelo *Irace* e apresentadas nas Tabelas 2 e 3 não consideram as correlações de parâmetros inter-*framework*.

**Tabela 2. Melhores configurações obtidas pelo *Irace* considerando somente parâmetros do Spark**

Parâmetro	Conf 1	Conf 2	Conf 3	Conf 4
P1	12	8	6	7
P2	6	7	7	8
P3	113	115	123	126
P4	200	200	200	200
P5	1	1	1	1
P6	0.4283	0.4578	0.4241	0.4698
P7	true	true	true	true
P8	static	static	static	static

**Tabela 3. Melhores configurações obtidas pelo *Irace* considerando somente parâmetros do Cassandra**

Parâmetro	Conf 1	Conf 2	Conf 3	Conf 4
P9	ANY	LOCAL.ONE	ANY	LOCAL.ONE
P10	32	256	128	256
P11	2000	auto	2000	100
P12	1000	5000	3000	5000
P13	20	20	20	20

Por outro lado, a Tabela 4 apresenta as quatro melhores configurações obtidas pelo *Irace*, considerando os parâmetros do Spark e Cassandra de forma integrada, ou seja, levando em conta a correlação de parâmetros inter-*framework*. Podemos perceber uma mudança significativa nos valores dos parâmetros em comparação com os resultados apresentados nas Tabelas 2 e 3. Por exemplo, a quantidade de *executors* do Spark foi reduzida de 113 para 62 e o parâmetro *PartitionOverwriteMode* (Parâmetro P8) foi alterado para “*Dynamic*”. Em especial, essa última mudança no valor do parâmetro faz com que o Spark não exclua as partições antecipadamente e apenas sobrescreva aquelas partições nas quais os dados foram escritos em tempo de execução. Por padrão, o Spark usa o valor “*Static*” para esse parâmetro. É importante ressaltar que essa configuração pode afetar a escrita das tabelas no Cassandra, pois elas são sempre sobrescritas no modo dinâmico, e a escolha do modo estático pode resultar em perda de desempenho. Além disso, a quantidade de *executors* do Spark está diretamente relacionada com a quantidade de escritas concorrentes que o Cassandra permite. A definição correta desse parâmetro pode evitar a sobrecarga do Cassandra ao lidar com muitas solicitações em paralelo.

Para comparar as configurações obtidas em cada um dos casos, o *pipeline* apresentado na Subseção 5.1 foi executado para cada um dos experimentos supracitados, *i.e.*,  $Ex_1$ ,  $Ex_2$ ,  $Ex_3$ ,  $Ex_4$  e  $Ex_5$ . Em cada experimento, foi escolhida a melhor configuração gerada pelo *Irace*, *i.e.*,  $Conf_1$ , conforme apresentado nas Tabelas 2, 3 e 4. A Figura 3 apresenta o tempo de execução (em segundos) do *pipeline* de ETL com cada uma das configurações definidas.

Podemos observar que as configurações padrão dos *frameworks* oferecem um desempenho aceitável, mas deixam bastante espaço para otimização. Como essas

**Tabela 4. Melhores configurações obtidas pelo *lrace* considerando de forma integrada parâmetros do Spark e do Cassandra**

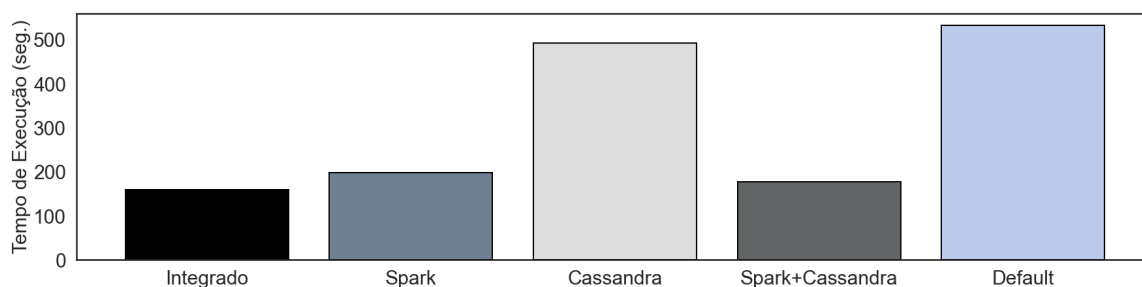
Parâmetro	Conf 1	Conf 2	Conf 3	Conf 4
P1	16	22	13	22
P2	1	7	2	1
P3	62	44	21	96
P4	300	300	500	500
P5	1	1	1	1
P6	0,6597	0,6459	0,4538	0,6712
P7	false	true	true	false
P8	dynamic	static	static	static
P9	ANY	LOCAL_ONE	LOCAL_ONE	ANY
P10	16	128	2056	2056
P11	100	auto	1000	100
P12	4000	1000	3000	4000
P13	20	20	20	15

configurações padrão são projetadas pelos desenvolvedores dos *frameworks* sem uma infraestrutura computacional específica em mente, é natural que não apresentem o melhor desempenho. A escolha da configuração apenas para os parâmetros do Cassandra também não apresentou um bom desempenho (7,6% de redução em comparação com a configuração padrão), muito devido à quantidade de *executors* padrão do Spark ser pequena e acabar sendo um ponto de contenção no *pipeline*. A configuração dos parâmetros apenas do Spark apresentou um desempenho melhor do que a configuração apenas do Cassandra (62,74% de redução em comparação com a configuração padrão), principalmente devido ao aumento da quantidade de *executors* para processar os dados.

Finalmente, as configurações que consideraram a otimização dos parâmetros dos dois *frameworks* em conjunto obtiveram os melhores resultados. O *Exp<sub>4</sub>* (Spark+Cassandra na Figura 3), que considera as otimizações isoladas de Spark e Cassandra combinadas em uma única execução, apresentou uma redução de 65,85% em comparação com a configuração *default*, enquanto que a otimização considerando todos os parâmetros de forma integrada (Integrado na Figura 3), apresentou uma redução de 69,99% em comparação com a configuração *default*. Embora a diferença em valores absolutos do tempo de execução seja pequena (poucos minutos no pior caso), os *pipelines* de ETL são executados várias vezes ao longo do tempo. Nos casos em que a frequência de execução é alta, o ganho acumulado ao longo do tempo não pode ser negligenciado. Supondo que esse *pipeline* seja executado três vezes ao dia, obtendo dados de redes sociais no período da manhã, tarde e noite, ao longo de um mês, aproximadamente nove horas de processamento teriam sido poupadas se considerarmos a otimização integrada dos parâmetros, em comparação à execução com valores *default*.

## 6. Conclusões

Nos últimos anos, houve um aumento significativo na computação centrada em dados, o que levou ao desenvolvimento de diversas aplicações que incorporam múltiplos *frameworks* e SGBDs. Apesar das metodologias existentes para desenvolver aplicações que integram esses *frameworks* e SGBDs, a definição da melhor configuração de parâmetros é um ponto crítico que muitas vezes não é abordado. Este artigo propõe a abordagem *Marin*, que utiliza um otimizador independente de modelo baseado no algoritmo de corrida iterada para definir a melhor configuração de parâmetros, otimizando de forma integrada e con-



**Figura 3. Tempo de execução em segundos para a melhor configuração definida pelo *lrace* em cada um dos experimentos.**

siderando as correlações entre parâmetros intra e inter-*framework*. A *Marin* foi avaliada por meio de um estudo de caso com um *pipeline* de ETL construído sobre Spark e Cassandra, obtendo melhorias de até 69,99% em comparação com a configuração *default* dos *frameworks*.

Os resultados dos experimentos mostraram-se satisfatórios quanto à otimização dos *frameworks* de forma integrada. Foi possível observar que a otimização de forma isolada produziu resultados melhores que aqueles obtidos considerando os valores *default* do Spark e do Cassandra, porém piores que a otimização de forma conjunta. Quanto ao otimizador, a utilização do *lrace* se mostrou adequada para a busca das melhores configurações sem que fosse preciso o treinamento de um modelo de Aprendizado de Máquina. Uma vez que todos os experimentos foram realizados em uma única máquina com múltiplos núcleos computacionais e com um *pipeline* de ETL simplificado, trabalhos futuros incluem a avaliação da *Marin*: (i) com aplicações de maior escala, *e.g.*, da bioinformática [Ocaña et al. 2011, Ocaña et al. 2015], e (ii) em ambientes distribuídos onde os nós que possam ser utilizados pelos *frameworks* estejam em pontos geográficos diferentes.

## Referências

- de Oliveira, D. C. M., Liu, J., and Pacitti, E. (2019). *Data-Intensive Workflow Management: For Clouds and Data-Intensive and Scalable Computing Environments*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.
- de Oliveira, D. E. M. et al. (2021). Towards optimizing the execution of spark scientific workflows using machine learning-based parameter tuning. *Concurr. Comput. Pract. Exp.*, 33(5).
- Essertel, G. et al. (2018). Flare: Optimizing Apache Spark with native compilation for scale-up architectures and medium-size data. In *13th USENIX OSDI*, pages 799–815.
- Haase, C., Röseler, T., and Seidel, M. (2022). METL: a modern ETL pipeline with a dynamic mapping matrix. *CoRR*, abs/2203.10289.
- Huang, X., Zhang, H., and Zhai, X. (2022). A novel reinforcement learning approach for spark configuration parameter optimization. *Sensors (Basel)*, 22(15):5930.
- Jin, W., Wang, H., Zha, D., Tan, Q., Ma, Y., Li, S., and Lee, S.-I. (2024). Dcai: Data-centric artificial intelligence. *WWW '24*, page 1482–1485, New York, NY, USA.

- Lama, P. and Zhou, X. (2012). Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC '12*, pages 63–72, New York, NY, USA.
- LeFevre, J., Liu, R., et al. (2016). Building the enterprise fabric for big data with vertica and spark integration. In *SIGMOD*, SIGMOD '16, page 63–75, New York, NY, USA.
- López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., and Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.
- Maron, O. and Moore, A. W. (1997). The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11(1):193–225.
- Mozaffari, M., Dignös, A., Gamper, J., and Störl, U. (2024). Self-tuning database systems: A systematic literature review of automatic database schema design and tuning. *ACM Comput. Surv.* Just Accepted.
- Ocaña, K. A. C. S., de Oliveira, D., Ogasawara, E. S., Dávila, A. M. R., Lima, A. A. B., and Mattoso, M. (2011). Sciphy: A cloud-based workflow for phylogenetic analysis of drug targets in protozoan genomes. In *6th BSB 2011, Brasilia, Brazil*, pages 66–70.
- Ocaña, K. A. C. S. et al. (2015). Data analytics in bioinformatics: Data science in practice for genomics analysis workflows. In *IEEE e-Science 2015*, pages 322–331.
- Oliveira, R., Baião, F., Machado, J., Almeida, A. C., and Lifschitz, S. (2022). Autonomic combination and selection of tuning actions. In *SBBD 2022*, pages 39–51. SBC.
- Pina, D. B., Chapman, A., Kunstmann, L. N. O., de Oliveira, D., and Mattoso, M. (2024). Dlprov: A data-centric support for deep learning workflow analyses. In *Proc. of the 8th DEEM-SIGMOD 2024, Santiago, Chile*, pages 77–85. ACM.
- Popescu, A., Balmin, A., Ercegovac, V., and Ailamaki, A. (2013). Predict: Towards predicting the runtime of large scale iterative analytics. *PVLDB*, 6(14):1678–1689.
- Sharma, A., Schuhknecht, F. M., and Dittrich, J. (2018). The case for automatic database administration using deep reinforcement learning. *ArXiv e-prints*.
- Silva-Muñoz, M., Franzin, A., and Bersini, H. (2021). Automatic configuration of the cassandra database using irace. *PeerJ Comput. Sci.*, 7:e634.
- Teylo, L., de Paula Junior, U., Frota, Y., de Oliveira, D., and Drummond, L. M. A. (2017). A hybrid evolutionary algorithm for task scheduling and data assignment of data-intensive scientific workflows on clouds. *Future Gener. Comput. Syst.*, 76:1–17.
- Yu, Z., Bei, Z., and Qian, X. (2018). Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *ASPLOS'18*, pages 564–577.
- Zaharia, M. (2019). Lessons from large-scale software as a service at databricks. SoCC '19, page 101, New York, NY, USA.
- Zhang, J. et al. (2021). Cdbtune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *VLDB J.*, 30(6):959–987.
- Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., and Yang, Y. (2017). Best-config: tapping the performance potential of systems via automatic configuration tuning. SoCC '17, page 338–350, New York, NY, USA.