UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS VII - PATOS
CENTRO DE CIÊNCIAS EXATAS E SOCIAIS APLICADAS
CURSO DE GRADUAÇÃO EM BACHARELADO EM COMPUTAÇÃO

THALYSSON HENRIQUE DA SILVA LOURENÇO

APLICANDO ESTRATÉGIAS DE BALANCEAMENTO DE CARGA PARA A TAREFA DE RESOLUÇÃO DE ENTIDADES EM SERVIDORES ERLANG

PATOS - PB
2023

THALYSSON HENRIQUE DA SILVA LOURENÇO

APLICANDO ESTRATÉGIAS DE BALANCEAMENTO DE CARGA PARA A TAREFA DE RESOLUÇÃO DE ENTIDADES EM SERVIDORES ERLANG

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Computação do Centro de Ciências Exatas e Sociais Aplicadas da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de bacharel em Computação.

Orientador: Dr. Demetrio Gomes Mestre

PATOS - PB
2023

L892a Lourenco, Thalysson Henrique da Silva.

Aplicando estratégias de balanceamento de carga para a tarefa de resolução de entidades em servidores Erlang [manuscrito] / Thalysson Henrique da Silva Lourenco. - 2023.

51 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências Exatas e Sociais Aplicadas, 2023.

"Orientação : Prof. Dr. Demetrio Gomes Mestre, Coordenação do Curso de Computação - CCEA. "

1. Algoritmos. 2. Balanceamento de carga. 3. Métodos de Indexação. 4. Erlang. I. Título

21. ed. CDD 005

Elaborada por Edson M. A. Monteiro - CRB - CRB 15/743 BSC7/UEPB

THALYSSON HENRIQUE DA SILVA LOURENÇO

APLICANDO ESTRATÉGIAS DE BALANCEAMENTO DE CARGA PARA A TAREFA DE RESOLUÇÃO DE ENTIDADES EM SERVIDORES ERLANG

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Computação do Centro de Ciências Exatas e Sociais Aplicadas da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de bacharel em Computação.

Trabalho aprovado em 27/06/2023.

BANCA EXAMINADORA

Demetrio Gomes Mestre

Prof. Dr. Demetrio Gomes Mestre

(Orientador)

Janderson Jason Barbosa Aguiar

Prof. Dr. Janderson Jason Barbosa Aguiar

(Examinador)

Ingrid M. M. de Lucena

Prof. Ma. Ingrid Morgane Medeiros de Lucena

(Examinadora)

Dedico este trabalho a minha família, amigos, professores e todos que me apoiaram nesta jornada. Este trabalho só foi possível graças a vocês.

AGRADECIMENTOS

Aos meus pais, pela dedicação e amor incondicional. Vocês são minha base e minha força para seguir em frente.

Aos meus professores e orientadores, pelo conhecimento transmitido e pela orientação ao longo deste trabalho. Sua sabedoria foi fundamental para o meu crescimento acadêmico e profissional.

A todos que compartilharam comigo momentos de aprendizado, risos e desafios, agradeço por fazerem parte desta trajetória.

“Se é uma boa ideia, vá em frente e faça
porque é mais fácil pedir desculpas do que conseguir permissão"
Grace Hopper

RESUMO

A Resolução de Entidades (RE), ou seja, a tarefa de identificar entidades que se referem a um mesmo objeto do mundo real, é uma tarefa importante e difícil para a integração e limpeza de fontes de dados. Uma das maiores dificuldades para a realização desta tarefa na era de Big Data é o tempo de execução elevado gerado pela natureza quadrática da tarefa. Assim, para reduzir o tempo de execução, a tarefa de RE pode ser realizada em paralelo com o uso de modelos programáticos construídos para rodar eficientemente em ambiente distribuído. Com o poder da computação distribuída, é possível explorar os pontos fortes de tecnologias de programação para sistemas distribuídos, como Erlang (uma linguagem de programação e sistema de tempo de execução criado para aplicações distribuídas), e promover a resolução eficiente e paralela de entidades dividindo a carga de trabalho da tarefa entre os recursos de um sistema distribuído. Para tanto, a robustez da estratégia de balanceamento da carga de trabalho, entre os nós da infraestrutura distribuída, é crucial para se alcançar alta **eficiência**. Os resultados mostram que, entre as abordagens para balanceamento de carga desenvolvidas neste trabalho, existem abordagens que, ao serem executadas, apresentaram padrões que evidenciam o aumento significativo de desempenho da tarefa de RE distribuída, promovendo, assim, uma redução no tempo de execução total e preservando a qualidade da detecção de pares de entidades similares.

Palavras-chave: Resolução de Entidades. Métodos de Indexação. Erlang. Algoritmos de Balanceamento de Carga.

ABSTRACT

Entity Resolution (ER), identifying entities that refer to the same real-world object, is an essential and difficult task for integrating and cleaning data sources. A major difficulty for this task performance in the Big Data era is the high runtime generated by the Cartesian product execution among all entities to perform the comparisons. Thus, to reduce execution time, the RE task can be performed in parallel using programmatic models built to run efficiently in a distributed environment. With the power of distributed computing, it is possible to exploit the strengths of programming technologies for distributed systems, such as Erlang (a programming language and runtime system created for distributed applications), and promote the efficient and parallel resolution of entities by dividing the workload of the task among the resources of a distributed system. Therefore, the robustness of the workload balancing strategy, among the nodes of the distributed infrastructure, is crucial to achieve high **efficiency**. The results show that, among the load-balancing approaches developed in this work, there are approaches that, when executed, presented patterns that show a significant increase in the performance of the distributed RE task, thus promoting a reduction in the total execution time and preserving the detection quality of pairs of similar entities.

Keywords: Entity Resolution. Indexing Methods. Erlang. Load Balancing Algorithms.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

RE	Resolução de Entidades
OTP	*Open Telecom Platform*
BEAM	*Bogdan/Bjorn's Erlang Abstract Machine*
EPMD	*Erlang Port Mapper Daemon*
SBM	*Standard Blocking Method*

SUMÁRIO

1 INTRODUÇÃO

Com o crescimento do volume de dados e a melhoria das habilidades dos sistemas de informação em coletar dados de várias fontes, muitas das vezes fontes distribuídas e heterogêneas, os problemas com integração e qualidade de dados tornam-se cada vez mais abundantes (CHRISTEN, 2012a). Nos últimos anos, temos visto um avanço enorme na pesquisa para processamento de grandes volumes de dados no domínio de Ciência da Computação, especialmente em áreas como Mineração de Dados, Aprendizado de Máquina, Recuperação da Informação, Banco de Dados e Comunidades de Armazém de Dados (PAPADAKIS et al., 2020).

À medida que as fontes de dados geradas pelos sistemas de informação das organizações vão aumentando, a compreensão dos dados e de seus valores vai sendo reconhecida com uma das maiores dificuldades para a utilização dessas fontes. Além disso, problemas de desempenho em sistemas de informação que processam tarefas como correlacionamento de conteúdo de páginas web, detecção de plagiarismo, detecção de fraudes e reconhecimento de pessoas em imagens, tem se tornado cada vez mais desafiadores devido a natureza de execução quadrática que estas tarefas possuem (CHRISTEN, 2012a).

Dentre as tarefas mais intrigantes que emergem deste cenário, a Resolução de Entidades (RE), também conhecida como Entity Matching, deduplicação ou Record Linkage, tem recebido considerável enfoque ultimamente devido a sua importância em vários domínios, sendo a tarefa de identificar entidades (por exemplo, pessoas, produtos, publicações ou websites) que se referem a um mesmo objeto do mundo real (KOPCKE; RAHM, 2010). É uma tarefa de extrema relevância para as áreas de Qualidade de Dados e Integração de Dados; por exemplo, para encontrar produtos repetidos em uma ou mais fontes de dados. Porém, acompanhada de sua importância vem suas dificuldades. A RE é uma tarefa intensiva em dados especialmente quando processada em grandes volumes de dados. Por exemplo, o processo de RE envolvendo dois conjuntos de dados com um milhão de entidades cada pode gerar cerca de um trilhão de pares de entidades a serem comparadas. Trata-se de um problema crítico em termos de desempenho devido ao alto custo computacional necessário para processar conjuntos de dados grandes (MESTRE, 2018).

A computação distribuída tem recebido bastante atenção ultimamente quanto à execução de tarefas envolvendo alta intensidade computacional. Poderosas infraestruturas de equipamentos e serviços distribuídos, capazes de processar milhões de tarefas simultaneamente, estão amplamente espalhadas pelo mundo. Visando-se fazer um uso eficiente de tais infraestruturas e serviços, vários modelos de computação paralela têm sido propostos (TANENBAUM; STEEN, 2023). Neste contexto, a plataforma **Erlang**, conhecida por seu modelo de concorrência, transferência de mensagens, processos leves e capacidade nativa de distribuição, oferece uma fundação robusta para a criação de sistemas distribuídos altamente confiáveis (JURIC, 2019).

No cenário digital de hoje, em que sites, aplicativos da Web e serviços online enfrentam níveis variados de demanda do usuário, o balanceamento de carga desempenha um papel importante no tratamento eficiente do aumento do tráfego. O balanceamento de carga é um conceito

crítico em redes de computadores e sistemas distribuídos que visa distribuir o tráfego de rede de entrada em vários servidores, recursos ou nós de computação paralela (GHOMI; RAHMANI; QADER, 2017). O objetivo principal do balanceamento de carga é otimizar a utilização de recursos, melhorar o desempenho, garantir alta disponibilidade e melhorar a escalabilidade de aplicativos e serviços. Ao distribuir uniformemente a carga de trabalho, os balanceadores de carga evitam que qualquer servidor ou recurso fique sobrecarregado, reduzindo o risco de degradação do desempenho ou interrupções de serviço (SHARMA; SINGH; SHARMA, 2008).

Portanto, a tarefa de RE é um problema propício a ser contemplado com soluções de computação paralela. Com a utilização da plataforma Erlang e o uso de estratégias eficientes de balanceamento de carga, a execução de tarefa de RE pode ser feita em paralelo de forma eficiente.

1.1 Objetivos

Um *cluster* Erlang possui a capacidade de distribuição nativa que, juntamente com o seu modelo de processos focado em concorrência, permite distribuir tarefas e mensagens para múltiplos nós da rede, além de facilitar o escalonamento horizontal dos recursos da rede. Por este motivo, este trabalho pretende estudar soluções para o balanceamento de carga da execução da tarefa de RE entre nós de *clusters* Erlang.

1.1.1 Objetivo Geral

O objetivo geral do trabalho consiste em implementar e avaliar algoritmos de balanceamento de carga utilizando as ferramentas nativas de concorrência e distribuição da biblioteca padrão da linguagem Elixir (JURIC, 2019) para garantir a distribuição efetiva de carga da tarefa RE (intensiva em dados) entre os nós conectados em *clusters* Erlang.

1.1.2 Objetivos Específicos

Para se alcançar o objetivo geral deste trabalho, foram necessários atingir os seguintes objetivos específicos:

- Realizar uma revisão bibliográfica sobre os conceitos de tarefas intensivas em dados, Resolução de Entidades, sistemas distribuídos, balanceamento de carga e algoritmos de balanceamento;
- Estudar a plataforma Erlang, com foco em suas funcionalidades de distribuição e concorrência, e a linguagem de programação funcional Elixir;
- Realizar um estudo de caso de implementação de algoritmos de balanceamento de carga para a execução da tarefa de RE sobre grandes conjuntos de dados, bem como analisar seus respectivos desempenhos de forma comparativa.

- Analisar os resultados obtidos para compreender e avaliar qual das estratégias de balanceamento de carga destacam-se no contexto da execução da tarefa de RE distribuída.

1.2 Justificativa

Ultimamente, tem-se visto um avanço enorme na pesquisa de RE no domínio de Ciência da Computação, especialmente em áreas como Mineração de Dados, Aprendizado de Máquina, Recuperação da Informação, Banco de Dados e *Data Warehousing* (PAPADAKIS et al., 2020). À medida que o volume de dados das fontes mantidas pelas organizações cresce e a compreensão dos dados e de seus valores vai sendo reconhecida como uma das maiores dificuldades para se utilizar essas fontes (LEE et al., 2006), a tarefa de identificar as entidades similares em fontes de dados, heterogêneas ou não, tem se tornado mais pervasiva do que nunca. Várias soluções de RE tem sido propostas para aplicar abordagens sofisticadas de aprendizado de máquina, processamento de linguagem natural e grafos (CHRISTEN, 2012a). Estas técnicas aprimoraram tanto a qualidade quanto o tempo de execução da detecção de similaridades em grandes fontes de dados contendo milhões de entidades.

Apesar dos avanços, devido às enormes proporções que as fontes de dados têm alcançado nos últimos tempos, as técnicas (especialmente as de indexação) elaboradas ao longo destes quarenta anos (sem levar em conta mecanismos de paralelização), tornaram-se ineficientes em termos de tempo de execução (PAPENBROCK; HEISE; NAUMANN, 2015). No contexto atual, não há mais interesse que um processo de RE seja executado durante vários dias (CHRISTEN; GAYLER; HAWKING, 2009). Para superar essa ineficiência, a comunidade científica tem dado ênfase à utilização de computação distribuída para processar tarefas de RE propondo abordagens distribuídas que refletem o mesmo comportamento das abordagens já consolidadas (por exemplo, RE baseada em blocagem (KOLB; THOR; RAHM, 2012b)(MESTRE; PIRES; NASCIMENTO, 2017)).

Assim, tendo em vista a necessidade de aprimoramento das abordagens já consolidadas para viabilizar que a RE possa ser utilizada em fontes de dados que apresentam tamanho na ordem de milhares/milhões de entidades sem demorar dias, é notória a existência de demandas por estudos acerca de ganho de desempenho, em termos de tempo de execução, das tarefas de RE. Para melhorar a performance e diminuir o tempo de resposta, um sistema Erlang pode ser implementado em um *cluster* para executar a tarefa de RE distribuída, a partir do uso de diversas instâncias de uma aplicação trabalhando de forma conjunta.

No entanto, esta implementação da distribuição de carga de uma tarefa intensiva em dados, como a RE, não pode ser feita de forma arbitrária, é necessário que sejam utilizados algoritmos de balanceamento para garantir que a execução das tarefas sejam distribuídas de forma eficiente, para se evitar que nós da infraestrutura distribuída fiquem ociosos enquanto outros estão sobrecarregados. Portanto, este trabalho propõe a implementação de cinco algoritmos de balanceamento de carga, utilizando as bibliotecas padrões de concorrência do Elixir e OTP (*Open Telecom Platform*) para distribuição de carga de tarefas intensivas em dados, como a RE,

entre nós de um *cluster* Erlang.

1.3 Metodologia

Tendo em vista que este trabalho possui como objetivo realizar uma investigação bibliográfica e o dos algoritmos de balanceamento de carga, capazes de executar a paralelização eficiente da tarefa de RE, as atividades realizadas durante o trabalho de conclusão do curso foram:

- **Revisão Bibliográfica**: Realização de uma revisão bibliográfica para estabelecer uma visão geral sobre o assunto do trabalho (i.e., uso de servidores de computação distribuída Erlang para execução da tarefa de RE). Essa revisão foi fundamentada em estudos científicos, artigos acadêmicos e literatura relevante, a fim de embasar teoricamente o trabalho;
- **Implementação dos Algoritmos de Balanceamento de Carga**: Desenvolvimento dos algoritmos de balanceamento de carga, usando as bibliotecas OTP e a linguagem de programação funcional Elixir, para processar a tarefa de RE distribuída sobre *clusters* Erlang;
- **Estudos Experimentais**: Realizar estudos experimentais para avaliarmos os resultados de desempenho dos algoritmos de RE incremental desenvolvidas, considerando o tempo de execução e qualidade na detecção de duplicatas da execução das técnicas;
- **Análise de resultados**: analisar os resultados obtidos para compreender e avaliar qual das estratégias de balanceamento de carga destacam-se no contexto da execução da tarefa de RE distribuída.

1.4 Estrutura do trabalho

Este trabalho apresenta seis capítulos e está organizado da seguinte maneira: no Capítulo 1, é apresentada uma visão geral deste trabalho em relação à contextualização do problema, objetivos, justificava e estrutura do trabalho; no Capítulo 2, são apresentados os conceitos e trabalhos relacionados a este trabalho de conclusão de curso; no Capítulo 3, são apresentados os algoritmos de balanceamento de carga desenvolvidos no estudo de caso; no Capítulo 4, são analisados e discutidos os resultados; no Capítulo 5, são apresentadas as considerações finais e sugestões para trabalhos futuros; e ao final, encontram-se as referências e os apêndices utilizados neste trabalho de conclusão de curso.

2 REFERENCIAL TEÓRICO

Neste capítulo estão presentes os tópicos necessários para o entendimento deste trabalho. As abordagens descritas neste documento estão inseridas no contexto de Resolução de Entidades (RE) em ambientes de computação distribuída. Portanto, faz-se necessária uma descrição sobre RE e o modelo de programação distribuída, Erlang, que foram utilizados neste trabalho, apresentando suas definições e como a RE pode se beneficiar deste modelo.

Os tópicos estão organizados em duas seções. Na Seção 2.2 será apresentado e formalizado o problema da RE. Além disso, serão discutidos os principais métodos de indexação. A Seção 2.1 apresentará o modelo de programação utilizado na computação distribuída, conhecido como Erlang. Será discutido como a tarefa de RE pode ser realizada utilizando a linguagem Erlang. Será ainda apresentada uma descrição acerca das limitações que devem ser levadas em consideração ao se modelar abordagens de RE baseadas em Erlang.

2.1 Resolução de Entidades

O problema de Resolução de Entidades, para qualquer ferramenta ou *framework* que pretenda resolvê-lo, de acordo com Mestre (2018), pode ser definido formalmente conforme segue:

Definição formal: dados dois conjuntos $A \in R$ e $B \in S$ de entidades comparáveis das fontes de dados R e S, o problema de Resolução de Entidades consiste em identificar todas as entidades que correspondem ao mesmo objeto do mundo real em $A \times B$. A definição inclui o caso especial de encontrar entidades similares dentro de um única fonte de dados $(A = B, R = S)$. Denota-se a relação de entidades comparáveis $R_E = (a_1, a_2, \ldots, a_n)$, em que cada a_i corresponde a um atributo comparável da entidade. Uma entidade que faz parte da relação E designa um valor para cada atributo. Isto significa que um conjunto de dados de entrada S contém um número finito de conjuntos de entidades $e = [(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)]$. O resultado da detecção de similaridades é representado por um conjunto de correspondências onde uma correspondência $c = (e_i, e_j, s)$ relaciona duas entidades e_i e e_j, das fontes R e S, segundo um valor de similaridade $s \in [0, 1]$ que indica o grau de similaridade entre as duas entidades .

A definição formal mostra em sua descrição um dos grandes problemas de RE, "identificar todas as entidades que correspondem ao mesmo objeto do mundo real em $A \times B$". O problema está na dificuldade de execução da tarefa de RE, uma vez que existe a necessidade de aplicação das funções de similaridade sobre o produto Cartesiano das entidades dos conjuntos A e B. A complexidade quadrática $O(n^2)$, resultante deste processo, indica que a tarefa é bastante ineficiente quando a quantidade de entidades dos conjuntos A e B alcança a ordem dos milhões (MESTRE, 2018).

Contudo, não há mais interesse que um processo de RE seja executado durante vários dias (CHRISTEN, 2012a) devido ao custo financeiro associado com a demora, mesmo para conjuntos de dados volumosos. Por exemplo, caso um conjunto de dados apresente excesso

de redundância por um longo período de tempo, a confiança e usabilidade dos dados diminui e, consequentemente, a insatisfação dos clientes aumentará. Assim, ao lidar com conjuntos de dados de grande porte, a utilização dos métodos de indexação torna-se essencial para reduzir o espaço de busca e, assim, agilizar a execução da tarefa.

2.1.1 Métodos de Indexação ou Blocagem

Os métodos de indexação são necessários quando se lida com grandes conjuntos de dados para reduzir o espaço de busca da tarefa de RE e, assim, evitar a execução do produto Cartesiano entre as entidades, restringindo a execução da RE para um subconjunto menor dos pares de entidades que tem maior probabilidade de serem similares. Vários métodos de indexação tem sido propostos ao longo dos anos. Uma avaliação dos principais métodos de indexação, desprovidos de mecanismos de paralelização, pode ser vista em (CHRISTEN, 2012a). Estes métodos geralmente utilizam uma chave, também conhecida como **chave de bloco**, para particionar as entidades a serem comparadas dentro de grupos (blocos). Assim, as entidades só são comparadas com outras entidades que compartilhem a mesma chave de bloco, ou seja, a tarefa de RE será executada restritamente dentro de cada bloco. A chave de bloco é composta geralmente por partes dos valores dos atributos da entidade (por exemplo, as primeiras três letras do título de uma publicação).

Os métodos de indexação tradicionais mais conhecidos são: o *Standard Blocking* (JARO, 1989), *o Sorted Neighborhood* (HERNáNDEZ; STOLFO, 1998), o *Fuzzy Blocking* (CHRISTEN, 2008) e o *Canopy Clustering* (COHEN; RICHMAN, 2002). Porém, neste trabalho, será considerado apenas o método de indexação tradicional *Standard Blocking*, pois é um dos métodos de indexação mais populares de RE (CHRISTEN, 2012b).

2.1.2 Método Standard Blocking

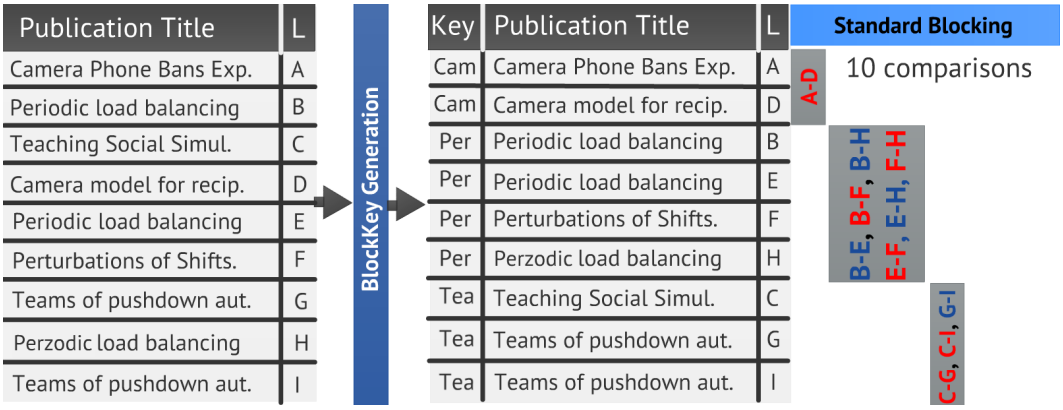
O método *Standard Blocking* (SBM) agrupa as entidades que compartilham a mesma chave de bloco dentro de um mesmo bloco conforme mencionado anteriormente. A chave de bloco é definida levando-se em consideração os valores dos atributos das entidades em cada conjunto de dados. Além disso, a chave de bloco pode ser composta por mais de um atributo; por exemplo, o atributo *coautores* pode ser combinado com o atributo *data de publicação* no contexto de RE em publicações.

Assim, dado um ou mais conjuntos de dados, com n entidades, que precisam ser submetidos a tarefa de RE, a aplicação do SBM gerará b blocos. Supondo que todos os blocos gerados apresentam a mesma quantidade n/b de entidades, o total de comparações entre pares de entidades será $O(n^2/b)$ (CHRISTEN, 2012a). Obviamente esta é uma generalização para calcular-se a complexidade do SBM no melhor caso de distribuição de entidades, mas esta suposição raramente acontece quando se lida com dados reais, pois a quantidade de entidades em cada bloco pode variar. Por exemplo, no contexto de filmes, no qual a chave de bloco é o *ano de produção*, é provável que a quantidade de filmes produzidos no ano de 1970 seja menor do que a

quantidade de produções de 2016. Assim, a quantidade de comparações de entidades no bloco do ano de 2016 vai ser maior do que o número de comparações de entidades no bloco 1970. Essa diferença nos tamanhos dos blocos pode gerar o problema conhecido como enviesamento de dados ou *data skewness* (KOLB; THOR; RAHM, 2012a), que será melhor explicado na Seção 2.2.

Por exemplo, a Figura 1 mostra que o conjunto de entrada $PublicationTitle$ consiste de n = 9 entidades (que vai de *Camera Phone Bans Exp.* até *Teams of pushdown aut.*) representados pelas letras L (de A a I). Todas as entidades são agrupadas de acordo com a chave de bloco K (*Cam*, *Per*, ou *Tea*) que por sua vez é composta pelas três primeiras letras do título da publicação. Perceba que os pares de entidades comparadas correspondem ao produto cartesiano de todas as entidades que compartilham a mesma chave de bloco. Assim, a quantidade de comparações executadas é 10 (1 para o bloco *Cam*, 6 para o bloco *Per* e 3 para o bloco *Tea*). Assumindo uma distribuição uniforme das entidades por bloco, a quantidade aproximada de comparações geradas pelo SBM é $\frac{1}{2} \cdot (\frac{n^2}{b} - n)$, em que b é o número de blocos gerados.

Figura 1 – Exemplo de execução do método Standard Blocking.



Fonte: (MESTRE, 2018).

2.1.3 Balanceamento de Carga e Enviesamento de Dados

O balanceamento de carga está relacionado com a uniformidade da distribuição de carga de trabalho entre dois ou mais nós (da infraestrutura distribuída) para otimizar a utilização de recursos, maximizar o desempenho, minimizar o tempo de resposta e evitar problemas de sobrecarga. Assim, em termos de balanceamento de carga para a RE baseada em Erlang, o que se busca é que cada nó do *cluster* Erlang execute aproximadamente o mesmo número de comparações. Para isso, a utilização de métodos de balanceamento de cargas eficientes é indispensável para se alcançar uniformidade na distribuição da carga de trabalho.

O número exato de comparações depende de vários fatores pertinentes ao problema em questão (neste caso, RE), tais como o enviesamento dos dados (diferença nos tamanhos dos blocos de entidades), quantidade de nós disponíveis para processamento, método de RE a ser utilizado, entre outros. Assim, fica evidente que a solução de balanceamento deve ser tratada no

trabalho e, nesse caso, um algoritmo arbitrário de balanceamento de carga não ajuda na solução. Por exemplo, na sistemática da abordagem **Basic** apresentada anteriormente, dependendo de como é definida a chave de bloco, pode haver a geração de blocos com grandes e pequenas quantidades de entidades provocando, assim, o enviesamento de dados. Então, tendo em vista que todas as entidades de um mesmo bloco são enviadas para um mesmo nó, a computação das comparações dos pares de entidades de blocos grandes pode ocupar por muito tempo alguns nós (da infraestrutura), deixando outros ociosos devido à disparidade (desbalanceamento) do tamanho das tarefas.

2.2 Sistemas Distribuídos

Um sistema distribuído, de acordo com Tanenbaum e Steen (2019), consiste em "um sistema de computadores em rede no qual processos e recursos estão suficientemente espalhados por vários computadores, geralmente para atender a requisitos de desempenho e confiabilidade". Eles definem os seguintes objetivos para um sistema distribuído:

- **Compartilhamento de recursos**: um sistema distribuído deve facilitar o acesso e compartilhamento de recursos remotos aos seus usuários, facilitando o trabalho colaborativo e a troca de informações entre eles.
- **Distribuição transparente**: processos e recursos podem estar espalhados em diversos computadores, mas esse fato não deverá ser revelado aos usuários. Um sistema não deve revelar aos seus usuários e/ou aplicações onde um recurso/processo está localizado ou como eles são acessados.
- **Abertura**: um sistema aberto deve oferecer componentes que possam ser facilmente usados ou integrados em outros sistemas. Um sistema deve aderir a determinadas regras e semânticas que descrevem o que os seus componentes oferecem.
- **Confiabilidade**: um sistema deve funcionar sempre dentro do esperado, onde falhas em um determinado componente não deverão ocasionar na indisponibilidade total do sistema. O ato de mascarar um erro e sua resolução é chamado de tolerância a falhas (TANENBAUM; STEEN, 2023).
- **Segurança**: "um sistema que não é seguro não é confiável" (TANENBAUM; STEEN, 2023). Deverão ser implementados mecanismos de autenticação e autorização, e mais especificamente para sistemas distribuídos, mecanismos de criptografia de dados para uma comunicação segura entre as entidades do sistema.
- **Escalabilidade**: refere-se à capacidade de um sistema se adaptar para conseguir lidar com diferentes cargas de trabalho sem afetar a sua performance. Um sistema deve suportar

a adição de usuários e recursos, que podem estar dispersos geograficamente e administrados por diferentes entidades administrativas, sem perda de performance e atrasos de comunicação notáveis (NEUMAN, 1994).

2.2.1 Balanceamento de Carga em Ambientes Distribuídos

O balanceamento de carga é um componente crítico dos sistemas distribuídos, sendo o processo de alocar e realocar a carga entre os recursos disponíveis, a fim de maximizar as taxas de transferência enquanto se minimiza o custo e tempo de resposta, melhorando o desempenho e a utilização de recursos, bem como a economia de energia (GHOMI; RAHMANI; QADER, 2017).

Balanceadores de carga implementam algoritmos específicos para determinar para qual nó uma determinada tarefa será enviada. Os algoritmos de balanceamento podem ser classificados como estáticos, que não consideram o estado dos nós para balancear a carga, e dinâmicos, onde os algoritmos dinâmicos consideram vários fatores como carga de processamento, uso de memória e total de conexões ativas em cada nó (SHAH; FARIK, 2015).

2.2.2 Erlang

Erlang é uma plataforma de desenvolvimento, que consiste em uma linguagem de programação, máquina virtual (*Bogdan/Bjorn's Erlang Abstract Machine*, ou BEAM) e sua biblioteca padrão, a OTP. Projetada para construir sistemas distribuídos altamente escaláveis e tolerantes a falhas, foi criada pela Ericsson no final dos anos 80 para uso em seus produtos de telecomunicações (JURIC, 2019).

Segundo Jurić (2019), Erlang foi criado especificamente para dar suporte ao desenvolvimento de sistemas altamente disponíveis, que estão sempre online e atendem seus clientes mesmo quando diante de circunstâncias inesperadas. Para que isso ocorra, alguns desafios técnicos devem ser enfrentados:

- **Tolerância a falhas**: um sistema deve funcionar mesmo com a ocorrência de erros. Tais erros devem ser isolados e resolvidos sem impactar o funcionamento do resto do sistema.
- **Escalabilidade**: um sistema deve estar preparado para suprir demandas repentinas sem intervenção de software.
- **Distribuição**: um sistema deve rodar em mais de uma máquina, para que a carga de trabalho seja distribuída entre elas e, caso uma máquina falhe, outra pode tomar o seu lugar sem interromper o funcionamento do sistema.
- **Responsividade**: as requisições realizadas a um sistema devem ser processadas rapidamente e não devem afetar a performance geral do sistema.

De acordo com Jurić (2019), o Erlang possui ferramentas que conseguem atacar todos esses desafios, e que sistemas que utilizam essas ferramentas se tornam altamente disponíveis através do poder da concorrência Erlang.

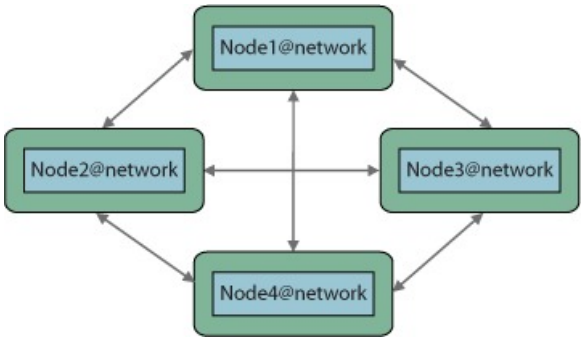
2.2.3 Erlang distribuído

Um sistema Erlang é executado em uma máquina virtual, denominada BEAM, que executa o código através de processos. Segundo Logan, Merritt e Carlson (2010), processos Erlang são uma espécie de bolha que provê isolamento e execução em paralelo, e o Erlang é construído totalmente ao redor de processos. O Erlang foi projetado para concorrência, que é a capacidade de haver várias tarefas em execução simultaneamente, e a unidade de concorrência é o processo (LOGAN; MERRITT; CARLSSON, 2010). Um processo executa código de forma independente e isolada, sem nenhum tipo de compartilhamento de memória e não pode alterar o estado de outros processos.

Devido a esse isolamento, surge a necessidade de um mecanismo de comunicação, que é a troca de mensagens. Uma mensagem é uma cópia do estado de um processo remetente enviado a um processo destinatário. Essa característica da linguagem torna o Erlang uma linguagem naturalmente distribuída, pois a troca de mensagens entre processos é feita de forma igual, mesmo que os processos estejam localizados em máquinas diferentes. Logan, Merritt e Carlson (2010) denominam essa característica "distribuição transparente", em que uma rede é nada mais que uma coleção de recursos, e o método de comunicação entre processos é o mesmo, independente de onde estejam localizados.

Na Figura 2, nota-se que as máquinas virtuais Erlang estão conectadas a uma rede e cada máquina virtual é denominada de *node* (nó). Quando dois ou mais *nodes* Erlang estão conectados, é denominado *cluster*. A comunicação entre *nodes* em um *cluster* é transitiva, ou seja, quando um *node* se conecta a outro, ele também se conecta a todos os outros *nodes* presentes no cluster, a não ser que seja explicitamente configurado o contrário (BARKLUND; VIRDING, 1999).

Figura 2 – Cluster Erlang.



Fonte: (LOGAN; MERRITT; CARLSSON, 2010).

Os *nodes* em um *cluster* se comunicam, por padrão, através do protocolo TCP/IP, por meio de um utilitário denominado *Erlang Port Mapper Daemon* (EPMD). O EPMD em uma

determinada máquina mantém um registro de quais *nodes* estão conectados nela e se comunica com o EPMD em outra máquina para identificar quais *nodes* estão presentes nessa outra máquina (BARKLUND; VIRDING, 1999).

2.2.4 Elixir

Elixir é uma linguagem de programação funcional, criada pelo brasileiro José Valim, que serve como alternativa ao Erlang, permitindo a escrita de código mais compacto que faz um trabalho melhor de revelar as intenções do programador. Elixir funciona com os mesmos conceitos de processos e mensagens do Erlang, e permite a interoperabilidade entre Elixir e Erlang (código Elixir pode chamar código Erlang diretamente) (JURIC, 2019).

O Elixir busca reduzir a quantidade de *boilerplate* (uma forma padrão de código gerado) e duplicação presente no Erlang, além de oferecer mecanismos de composição de funções (JURIC, 2019). A Ffigura 3 mostra um exemplo de código escrito em Erlang ao lado do mesmo código escrito em Elixir. Ambos os códigos fazem a mesma operação de somar 2 números utilizando concorrência.

Figura 3 – Comparação entre um código de somar no Erlang (esquerda) e Elixir (direita).

```
-module(sum_server).
-behaviour(gen_server).

-export([
    start/0, sum/3,
    init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
    code_change/3
]).
start() -> gen_server:start(?MODULE, [], []).
sum(Server, A, B) -> gen_server:call(Server, {sum, A, B}).

init(_) -> {ok, undefined}.
handle_call({sum, A, B}, _From, State) -> {reply, A + B, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

```
defmodule SumServer do
  use GenServer

  def start do
    GenServer.start(__MODULE__, nil)
  end

  def sum(server, a, b) do
    GenServer.call(server, {:sum, a, b})
  end

  def handle_call({:sum, a, b}, _from, state) do
    {:reply, a + b, state}
  end
end
```

Fonte: Elaborada pelo autor.

2.2.5 GenServer

O GenServer é um módulo da biblioteca padrão do Erlang que abstrai as funcionalidades de transmissão de mensagens dos processos Elixir. Um GenServer pode armazenar um estado e definir funções para retornar e alterar o seu estado atual de forma concorrente. (HEXDOCS, 2023)

3 ESTRATÉGIAS DE BALANCEAMENTO DE CARGA DESENVOLVIDAS

Nesta seção serão apresentados os algoritmos de balanceamento de carga desenvolvidos e os métodos utilizados para sua implementação. Os algoritmos apresentados neste Capítulo implementam o GenServer e armazenam a lista de *nodes* conectados como estado.

3.1 *Round Robin*

O primeiro algoritmo implementado foi o *Round Robin*. Ao executá-lo, cada chamada ao seu módulo retorna o primeiro elemento da lista de *nodes* como resposta e altera o estado do processo enviando esse *node* para o final da lista de *nodes*. Assim, as tarefas são distribuídas entres os *nodes* de forma sequencial. A Figura 4 mostra a implementação do algoritmo utilizando a função *GenServer.handle_call*/3.

Figura 4 – Função de retorno de *node* para o algoritmo *Round Robin*.

```
@impl true
def handle_call(:get_next_node, _from, [%{name: node} = node | nodes]) do
  {:reply, current_node, nodes ++ [node]}
end
```

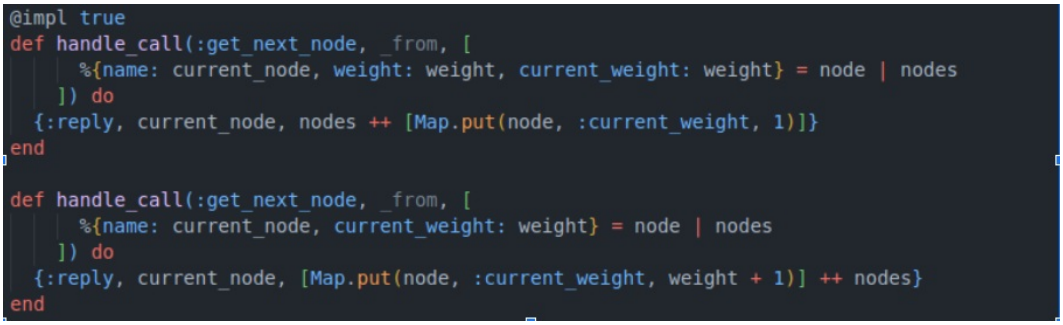
Fonte: Elaborada pelo autor.

A função seleciona e retorna o primeiro elemento da lista. A função *handle_call* deve, obrigatoriamente, retornar uma tupla de três elementos, sendo o primeiro elemento o átomo *:reply*, o segundo elemento é a resposta que o *GenServer* enviará ao processo que o chamou, e o terceiro elemento é o estado atualizado do *GenServer*.

3.2 *Round Robin* Ponderado

O *Round Robin* ponderado implementa a mesma lógica do *round robin*, mas com uma atribuição de pesos para cada *node*, sendo um *node* retornado um número de vezes determinado pelo seu peso antes de ser enviado ao final da lista. Os *nodes* são armazenados como uma lista de mapas, e cada mapa contém o nome do *node*, o seu peso atribuído e o peso atual. A Figura 5 mostra a implementação da função *handle_call* definida para o algoritmo.

Figura 5 – Função de retorno de *node* para o algoritmo *Round Robin* Ponderado.



```
@impl true
def handle_call(:get_next_node, _from, [
      %{name: current_node, weight: weight, current_weight: weight} = node | nodes
    ]) do
  {:reply, current_node, nodes ++ [Map.put(node, :current_weight, 1)]}
end

def handle_call(:get_next_node, _from, [
      %{name: current_node, current_weight: weight} = node | nodes
    ]) do
  {:reply, current_node, [Map.put(node, :current_weight, weight + 1)] ++ nodes}
end
```

Fonte: Elaborada pelo autor.

Os nodes são inicializados com peso 1, e cada chamada ao algoritmo incrementa o valor do peso atual do primeiro *node* da lista em 1. Caso o peso atual do *node* selecionado seja igual ao seu peso atribuído, o seu peso atual é reiniciado em 1 e é enviado para o final da lista. A função utiliza-se da funcionalidade de múltiplas cláusulas da linguagem Elixir, em que uma função pode possuir múltiplas cláusulas, e seus parâmetros são avaliados diretamente na declaração da função usando *pattern matching* (correspondência de padrões).

O *pattern matching* é uma das características do paradigma funcional, que consiste em comparar um termo à direita com um padrão à esquerda. No Elixir, os argumentos de uma função são considerados padrões, e ao chamar uma determinada função, os valores são correspondidos aos padrões especificados na declaração da função (JURIC, 2019).

3.3 Menos Conexões

No algoritmo de Menos Conexões os *nodes* são selecionados pelo número atual de tarefas sendo executadas atualmente por *node*. Cada chamada ao módulo incrementa um contador de conexões vinculado a cada *node*, e o *node* retornado é o que tiver a menor contagem. Ao finalizar a tarefa, o contador do *node* é decrementado. A Figura 6 mostra a implementação do algoritmo de Menos Conexões.

Figura 6 – Função de retorno de *node* para o algoritmo Menos Conexões.

```elixir
@impl true
def handle_call(:get_next_node, _from, nodes) do
  {%{name: node_name, connections: connections} = node, index} =
    nodes
    |> Enum.with_index()      You, semana passada • feat: add produtos.txt
    |> Enum.min_by(fn {%{connections: connections}, _} -> connections end)

  new_state =
    node
    |> Map.put(:connections, connections + 1)
    |> then(&List.replace_at(nodes, index, &1))

  {:reply, node_name, new_state}
end
```

Fonte: Elaborada pelo autor.

A função atribui um índice aos elementos da lista dos *nodes*, e retorna o *node* que possui o menor número de conexões ativas. Após isso, o número de conexões do *node* é incrementando em 1 e é reinserido na lista de conexões. Após isto, o *node* é retornado como resposta e a lista de *nodes* é retornada como novo estado do *GenServer*.

Após finalizar a tarefa, a função de decremento de conexões é atualizada, desta vez utilizando a função *GenServer.cast*/3. A função *handle_cast* funciona de forma semelhante à função *handle_call*, com a diferença de que a *handle_cast* não retorna uma resposta ao processo que chamou o *GenServer*. A Figura 7 mostra a implementação da função de decremento de conexões.

Figura 7 – Função de decremento de conexões para o algoritmo Menos Conexões.

```elixir
@impl true
def handle_cast({:free_connection, node_name}, nodes) do
  {%{connections: connections} = node, index} =
    nodes
    |> Enum.with_index()
    |> Enum.find(fn {%{name: name}, _} -> name == node_name end)

  new_state =
    node
    |> Map.put(:connections, connections - 1)
    |> then(&List.replace_at(nodes, index, &1))

  {:noreply, new_state}
end
```

Fonte: Elaborada pelo autor.

A função de decremento utiliza da mesma lógica da função de retorno de *node*, com a diferença que o número de conexões é decrementado, e a resposta constitui em uma tupla de

dois elementos, onde o primeiro é o átomo *:noreply* e o segundo elemento é o novo estado do *GenServer*. A função *handle_call* é assíncrona, ou seja, o processo que a chamou não aguarda resposta.

3.4 Menos Conexões Ponderado

Este algoritmo combina a lógica do algoritmo de menos conexões com uma atribuição de peso a cada *node*, assim como o *Round Robin* Ponderado. O *node* selecionado será o que estiver com o menor número de conexões ativas, dividido pelo peso alocado ao *node*. Assim, *nodes* com maior peso receberão mais tarefas. A Figura 8 mostra a implementação do algoritmo.

Figura 8 – Função de retorno de *node* para o algoritmo Menos Conexões Ponderado.

```
@impl true
def handle_call(:get_next_node, _from, nodes) do
  {%{name: node_name, connections: connections} = node, index} =
    nodes
    |> Enum.with_index()
    |> Enum.min_by(fn {%{connections: connections, weight: weight}, _} ->
      connections / weight
    end)

  new_state =
    node
    |> Map.put(:connections, connections + 1)
    |> then(&List.replace_at(nodes, index, &1))

  {:reply, node_name, new_state}
end
```

Fonte: Elaborada pelo autor.

A lógica implementada é quase idêntica à lógica do algoritmo Menos Conexões, com a diferença que o número de conexões do *node* é dividido pelo seu peso. Assim, caso dois *nodes* possuam o mesmo número de conexões, aquele com peso maior será escolhido. Para decremento de conexões, a lógica é igual ao algoritmo Menos Conexões.

3.5 Menor Tempo de Resposta

O algoritmo de menor tempo de resposta expande a lógica do algoritmo de Menos Conexões ao armazenar o tempo médio de execução das tarefas enviadas para cada *node*. Os *nodes* com menos conexões são selecionados, e aquele com o menor tempo médio de resposta será retornado. A Figura 9 mostra a implementação do algoritmo.

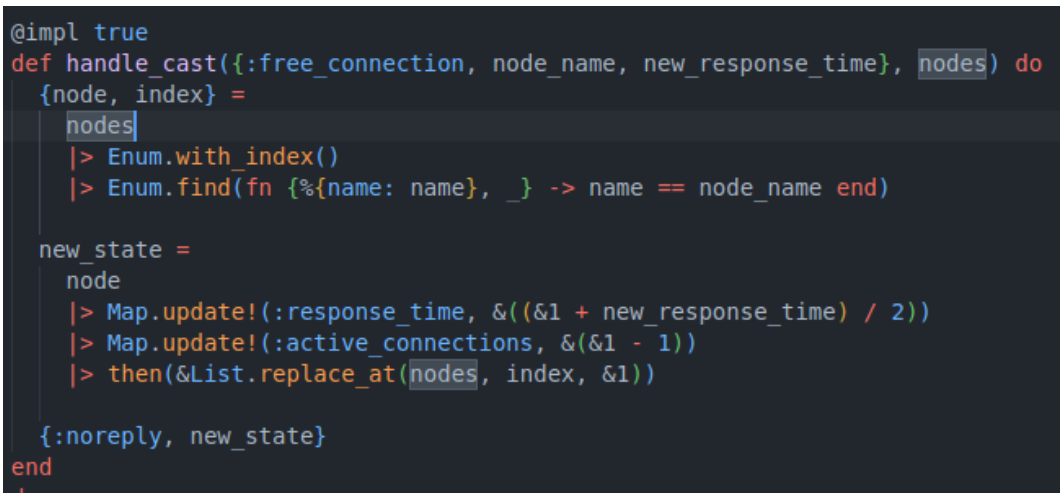
Figura 9 – Função de retorno de *node* para o algoritmo Menor Tempo de Resposta.

```
@impl true
def handle_call(:get_next_node, _from, nodes) do
  %{active_connections: active_connections_to_filter} =
    Enum.min_by(nodes, & &1.active_connections)

  {%{name: node_name} = node, index} =
    nodes
    |> Enum.with_index()
    |> Enum.filter(fn {%{active_connections: active_connections}, _} ->
      active_connections == active_connections_to_filter
    end)
    |> Enum.min_by(fn {%{response_time: response_time}, _} -> response_time end)

  new_state =
    node
    |> Map.update!(:active_connections, &(&1 + 1))
    |> then(&List.replace_at(nodes, index, &1))

  {:reply, node_name, new_state}
end
```

Fonte: Elaborada pelo autor.

A função implementada busca pelo menor número de conexões ativas, buscando entre os *nodes* com menos conexões aquele que estiver com o menor tempo médio de resposta. Após encontrar o *node* desejado, o seu número de conexões é incrementado em 1 e ele é retornado como resposta, e a lista de *nodes* atualizada com o *node* incrementado. Após finalizada a tarefa, a função de decremento de conexões, mostrada na Figura 10, é chamada.

Figura 10 – Função de decremento de conexões de *node* para o algoritmo Menor Tempo de Resposta.



```
@impl true
def handle_cast({:free_connection, node_name, new_response_time}, nodes) do
  {node, index} =
    nodes
    |> Enum.with_index()
    |> Enum.find(fn {%{name: name}, _} -> name == node_name end)

  new_state =
    node
    |> Map.update!(:response_time, &((&1 + new_response_time) / 2))
    |> Map.update!(:active_connections, &(&1 - 1))
    |> then(&List.replace_at(nodes, index, &1))

  {:noreply, new_state}
end
```

Fonte: Elaborada pelo autor.

Seguindo uma lógica semelhante à função de decremento do algoritmo Menos Conexões, além de decrementar em 1 o número de conexões ativas do *node*, o algoritmo executa uma

operação que altera o tempo médio de resposta do *node*, adicionando o tempo de resposta da última tarefa ao tempo de resposta médio do *node*, dividindo o valor por 2.

4 ESTUDO DE CASO: RESOLUÇÃO DE PRODUTOS

Neste Capítulo será apresentado um estudo de caso realizado utilizando um conjunto de dados de 100 mil descrições de produtos, visando identificar e eliminar produtos duplicados utilizando o método de indexação *Standard Blocking* (para quebra da tarefa de RE em blocos de tarefas menores de RE) e o algoritmo Jaro-Winkler (CHRISTEN, 2012a), que mede a similaridade entre duas strings. Utilizando a função $String.jaro\_distance/2$ da biblioteca padrão Elixir, as strings cujo índice de similaridade fossem maiores ou iguais a $0,7$ seriam consideradas duplicatas e, portanto, removidas do conjunto de dados.

4.1 Ambiente de execução

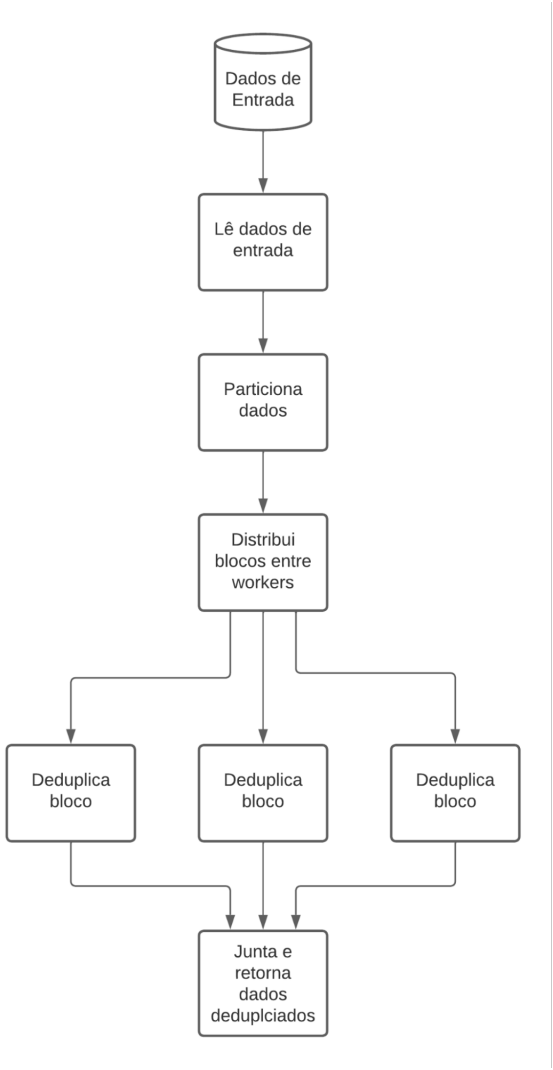
O ambiente de execução dos testes consiste em um cluster com 4 *nodes*, sendo um deles responsável por ler a entrada, dividir os dados e balancear a carga, juntamente com 3 *nodes* que recebem os dados particionados e executam um algoritmo de resolução de entidades. Após deduplicados, os dados são enviados de volta para o *node* inicial, e as partições são juntadas novamente, retornando a lista de entidades duplicadas.

Os *nodes* foram provisionados utilizando o software Docker[1] para rodar em contêineres, utilizando uma rede privada para conexão entre os *nodes*. Os algoritmos foram implementados em um projeto consistindo em 2 aplicações: *entity_resolution*, responsável por ler os dados de entrada, particionar e realizar o balanceamento; e *entity_resolution_worker*, que recebe o bloco e executa um algoritmo de resolução em um bloco de entidades utilizando o cálculo de similaridade entre strings Jaro-Wrinkler. Após a RE, o conjunto de dados sem duplicatas é retornado.

Nos casos dos algoritmos Round Robin Ponderado e Menos Conexões Ponderado, foram definidos peso 4 para o primeiro *worker*, peso 3 para o segundo *worker*, e peso 2 para o terceiro *worker*. Para os outros algoritmos, foram definidos peso 1 para todos os *workers*. Após o particionamento dos dados, os blocos são enviados de forma concorrente para serem processados nos *workers*. A Figura 11 apresenta o fluxograma do processo executado neste estudo de caso.

[1] O Docker é uma plataforma *open source* que facilita a criação e administração de ambientes isolados.

Figura 11 – Fluxograma do processo de deduplicação de dados.



Fonte: Elaborada pelo autor.

4.2 Cenários de execução

Os dados utilizados para os testes consistem em um arquivo de texto contendo uma lista de descrições de produtos, perfazendo um total de 100 mil registros de produtos. Para cada algoritmo de balanceamento de carga, foram executados 4 cenários de teste visando descobrir se algum deles ofereceria um bom desempenho para execução da tarefa de RE usando indexação. Classificados conforme o particionamento dos dados de entrada, os cenários foram configurados da seguinte maneira:

- **Cenário 1 - blocos do mesmo tamanho**: 100 blocos com 1000 elementos cada;
- **Cenário 2 - blocos de tamanho diferentes, com disparidade pequena**: bloco inicial com 5 mil elementos, cada bloco subsequente com 10% a menos que o anterior;

- **Cenário 3 - blocos de tamanho diferentes, com disparidade média**: bloco inicial com 10 mil elementos, cada bloco subsequente com 10% a menos que o anterior;
- **Cenário 4 - blocos de tamanho diferentes, com disparidade alta**: bloco inicial com 10 mil elementos, cada bloco subsequente com 50% a menos que o anterior;

Estes quatro cenários foram idealizados para analisar-se o desempenho dos algoritmos de balanceamento de carga para a RE distribuída relacionados aos diferentes cenários de enviesamento de dados que são possíveis serem encontrados nas fontes de dados de produção. Os algoritmos de balanceamento de carga para servidores Erlang implementados neste trabalho foram submetidos a análise experimental comparativa considerando-se os 4 cenários supracitados.

4.3 Total de elementos deduplicados

O Quadro 1 apresenta os totais de dados deduplicados para cada cenário. Como os blocos foram iguais para todos os algoritmos, o total deduplicado permaneceu o mesmo em todos os casos.

Quadro 1 – Total de elementos deduplicados por cenário.

Cenário	Total de Elementos	Percentual de Deduplicação
1	56537	43,46%
2	47277	52,72%
3	38872	61,13%
4	35485	64,51%

Fonte: Elaborada pelo autor.

4.4 Resultados por cenário

As seções seguintes apresentam os resultados obtidos na execução de cada cenário, com discussões sobre como os algoritmos executaram cada cenário e quais algoritmos realizaram melhor balanceamento em cada cenário.

4.4.1 Cenário 1

No cenário 1, todos os algoritmos apresentaram tempos de execução semelhantes. Os algoritmos ponderados distribuíram mais blocos para determinados *nodes*, mas, devido ao fato do *cluster* estar implementado em um único *host*, no momento que um *worker* finalizou os seus blocos, os outros *workers* ganharam mais recursos e conseguiram finalizar em um tempo semelhante às suas versões não ponderadas. O Quadro 2 mostra os resultados obtidos por cada algoritmo para o Cenário 1.

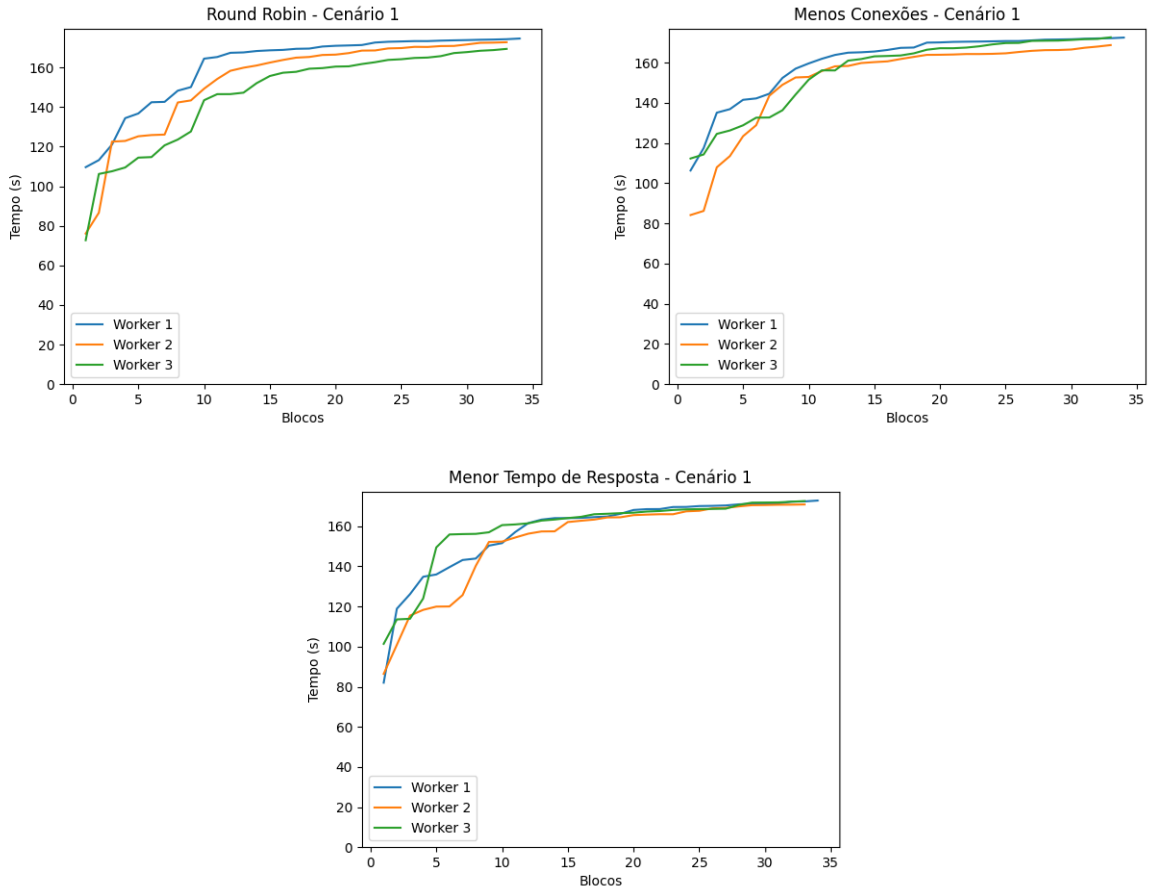
Quadro 2 – Resultados de execução dos algoritmos para o cenário 1.

Algoritmo	**Tempo Total de Execução**	**Tempo Médio de Execução de Bloco por *worker***	**Total de Blocos Enviados Para Cada *worker***
Round Robin	174 segundos	*worker* 1: 161 s *worker* 2: 153 s *worker* 3: 146 s	*worker* 1: 34 *worker* 2: 33 *worker* 3: 33
Round Robin Ponderado	172 segundos	*worker* 1: 164 s *worker* 2: 142 s *worker* 3: 99 s	*worker* 1: 45 *worker* 2: 33 *worker* 3: 22
Menos Conexões	173 segundos	*worker* 1: 160 s *worker* 2: 152 s *worker* 3: 156 s	*worker* 1: 34 *worker* 2: 33 *worker* 3: 33
Menos Conexões Ponderado	172 segundos	*worker* 1: 166 s *worker* 2: 140 s *worker* 3: 102 s	*worker* 1: 45 *worker* 2: 33 *worker* 3: 22
Menor Tempo de Resposta	172 segundos	*worker* 1: 157 s *worker* 2: 153 s *worker* 3: 159 s	*worker* 1: 34 *worker* 2: 33 *worker* 3: 33

Fonte: Elaborada pelo autor.

Analisando os gráficos de tempo de execução do cenário na Figura 12, podemos perceber que os tempos de processamento de bloco nos algoritmos *Round Robin*, Menos Conexões e Menor Tempo de Resposta ficaram bem próximos. Isso pode significar que a distribuição dos blocos finalizou antes do término da execução de um único bloco.

Figura 12 – Gráficos de tempo de execução dos algoritmos *Round Robin*, Menos Conexões e Menor Tempo de Resposta para o cenário 1.

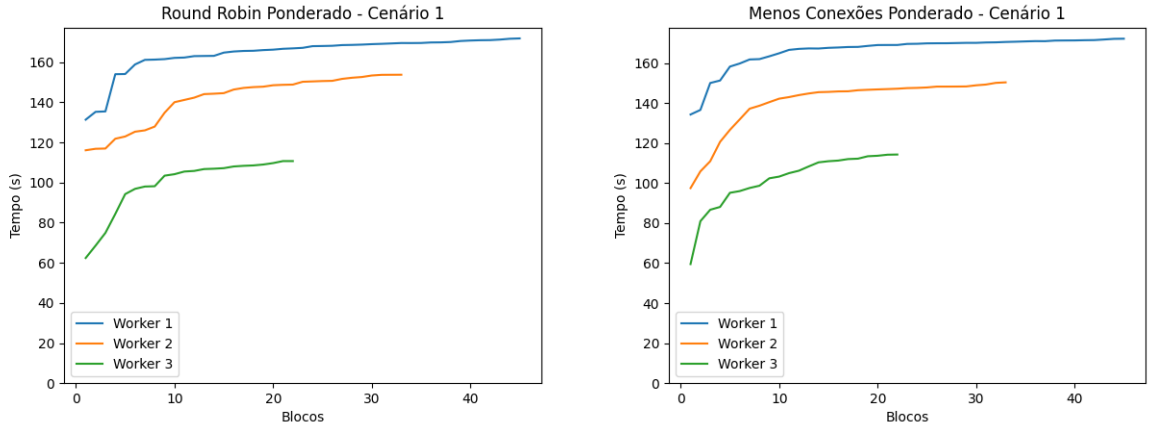


Fonte: Elaborada pelo autor.

Analisando os gráficos de tempo de execução dos algoritmos ponderados (Figura 13), percebe-se que os tempos de execução apresentam uma disparidade entre os *workers*. Isso ocorreu devido à combinação de 2 fatores:

- Os *workers* apresentam recursos uniformes, então os *workers* que receberam menos blocos finalizaram o processamento mais rapidamente.
- Os processos executados na BEAM são mantidos em filas, e caso um processo não finalize sua execução em um determinado tempo, ele é pausado e mandado para o final da fila. *Workers* com fila maior demoram mais a executar todos os processos.

Figura 13 – Gráficos de tempo de execução dos algoritmos ponderados para o cenário 1.



Fonte: Elaborada pelo autor.

Analisando os resultados obtidos, conclui-se que o algoritmo Menor Tempo de Resposta foi o melhor para este caso, pois, além de possuir o menor tempo de execução, apresentou médias de tempo de processamento mais próximas entre os *workers* que os outros algoritmos. No caso dos algoritmos ponderados, o Menos Conexões Ponderado se sobressaiu, pela mesma razão do algoritmo Menor Tempo de Resposta, ao ter médias de tempo de processamento mais próximas entre os *workers* que o *Round Robin* Ponderado.

4.4.2 Cenário 2

O Quadro 3 mostra os resultados obtidos por cada algoritmo para o cenário 2. Devido às variações nos tamanhos dos blocos, foram adicionadas colunas com os maiores e menores tempos de execução.

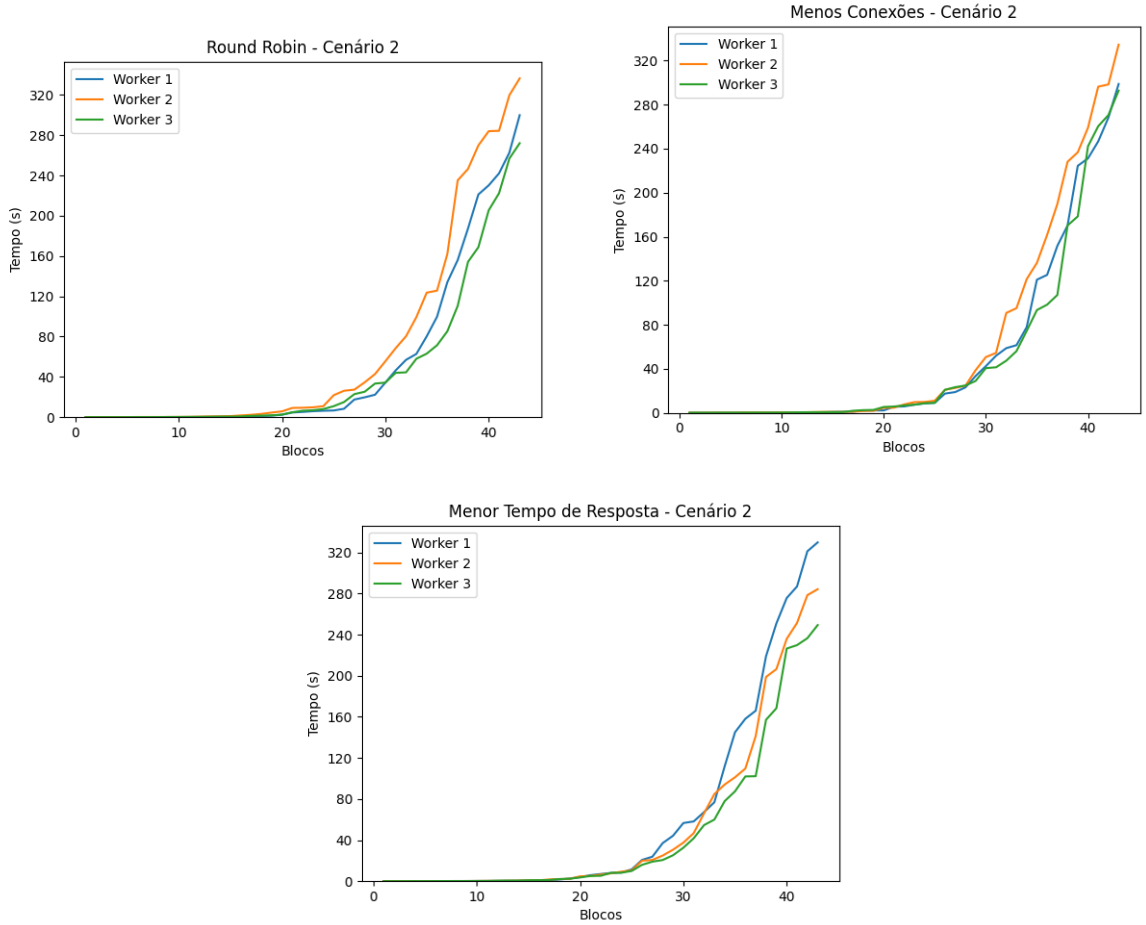
Quadro 3 – Resultados de execução dos algoritmos para o cenário 2.

Algoritmo	**Tempo Total de Execução**	**Tempo Médio de Execução de Bloco por *worker***	**Total de Blocos Enviados para cada *worker***	**Menor Tempo por *worker***	**Maior Tempo por *worker***
Round Robin	336 segundos	*worker* 1: 52 s *worker* 2: 68 s *worker* 3: 45 s	*worker* 1: 43 *worker* 2: 43 *worker* 3: 43	*worker* 1: 0ms *worker* 2: 10 ms *worker* 3: 43 ms	*worker* 1: 300 s *worker* 2: 336 s *worker* 3: 272 s
Round Robin Ponderado	314 segundos	*worker* 1: 73 s *worker* 2: 42 s *worker* 3: 22 s	*worker* 1: 59 *worker* 2: 42 *worker* 3: 28	*worker* 1: 0 ms *worker* 2: 2 ms *worker* 3: 17 ms	*worker* 1: 314 s *worker* 2: 198 s *worker* 3: 112 s
Menos Conexões	335 segundos	*worker* 1: 53 s *worker* 2: 63 s *worker* 3: 49 s	*worker* 1: 43 *worker* 2: 43 *worker* 3: 43	*worker* 1: 5 ms *worker* 2: 0 ms *worker* 3: 24 ms	*worker* 1: 299 s *worker* 2: 335 s *worker* 3: 293 s
Menos Conexões Ponderado	337 segundos	*worker* 1: 61 s *worker* 2: 61 s *worker* 3: 40 s	*worker* 1: 57 *worker* 2: 43 *worker* 3: 29	*worker* 1: 23 ms *worker* 2: 0 ms *worker* 3: 2 ms	*worker* 1: 337 s *worker* 2: 303 s *worker* 3: 225 s
Menor Tempo de Resposta	329 segundos	*worker* 1: 63 s *worker* 2: 53 s *worker* 3: 46 s	*worker* 1: 43 *worker* 2: 43 *worker* 3: 43	*worker* 1: 0 ms *worker* 2: 2 ms *worker* 3: 6 ms	*worker* 1: 330 s *worker* 2: 284 s *worker* 3: 249 s

Fonte: Elaborada pelo autor.

Assim como no cenário 1, os tempos totais de execução de cada algoritmo foram bem semelhantes. Os algoritmos não ponderados apresentaram distribuições de blocos iguais em suas execuções, assim como tempos médios de execução semelhantes para cada *worker*. Nota-se que houve uma grande diferença entre o maior e o menor tempo de processamento dos blocos, devido à diferença de tamanho: os maiores blocos, de 5 mil elementos, apresentam um tempo de execução de magnitudes maiores que os blocos de menor tamanho, mostrando a complexidade da tarefa de execução de entidades. Essa diferença de tempo de processamento é expressa nos gráficos de tempo de processamento (Figura 14).

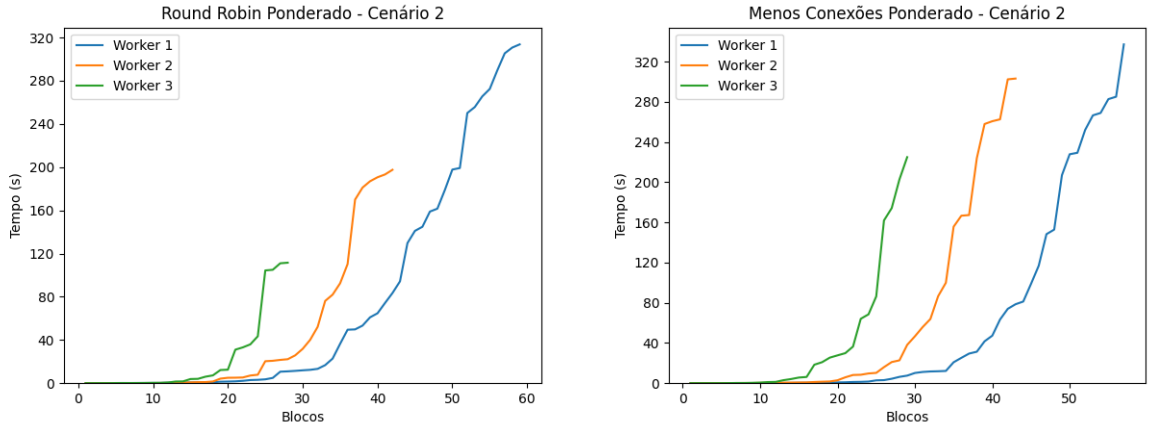
Figura 14 – Gráficos de tempo de execução dos algoritmos *Round Robin*, Menos Conexões e Menor Tempo de Resposta para o cenário 2.



Fonte: Elaborada pelo autor.

Os gráficos de tempo dos algoritmos ponderados (Figura 15) mostram comportamentos mais semelhantes às suas versões não ponderadas quando comparados ao cenário 1. Isso se deve ao fato dos tamanhos de blocos serem não uniformes, o que acaba afetando o tempo de processamento dos blocos e tamanho das filas de processamento. Ainda assim, os blocos com maior peso apresentaram maiores médias de tempo de processamento.

Figura 15 – Gráficos de tempo de execução dos algoritmos ponderados para o cenário 2.



Fonte: Elaborada pelo autor.

O *Round Robin* ponderado distribuiu de forma proporcional a quantidade de blocos entre os três *workers*, enquanto o Menos Conexões não. Isso provavelmente ocorreu pelo fato do *worker* 2 ter recebido blocos de tamanho menor, e, ao processá-los, acabou liberando mais conexões. O *worker* 2 recebeu mais blocos e ficou menos ocioso, mas isso afetou o tempo total de execução, provavelmente devido ao tamanho dos blocos recebidos.

Para este cenário, com pequenas diferenças entre os tamanhos de blocos, o algoritmo Menor Tempo de Resposta se mostrou o melhor para o balanceamento. Isso se deve ao fato de, ao acompanhar o tempo de execução médio de blocos, conseguir detectar qual *node* estava processando blocos menores e alocando blocos a estes. Para os algoritmos ponderados, o *Round Robin* Ponderado se mostrou mais vantajoso quando comparado ao de Menos Conexões Ponderado, pois apresentou menor tempo de execução.

4.4.3 Cenário 3

O Quadro 4 mostra os resultados obtidos por cada algoritmo para o Cenário 3. Este cenário demonstra a complexidade da resolução de entidades: mesmo havendo menos blocos, o tempo de execução total chegou a ser até 6 vezes maior que no cenário anterior, devido ao fato dos blocos terem tamanho maior.

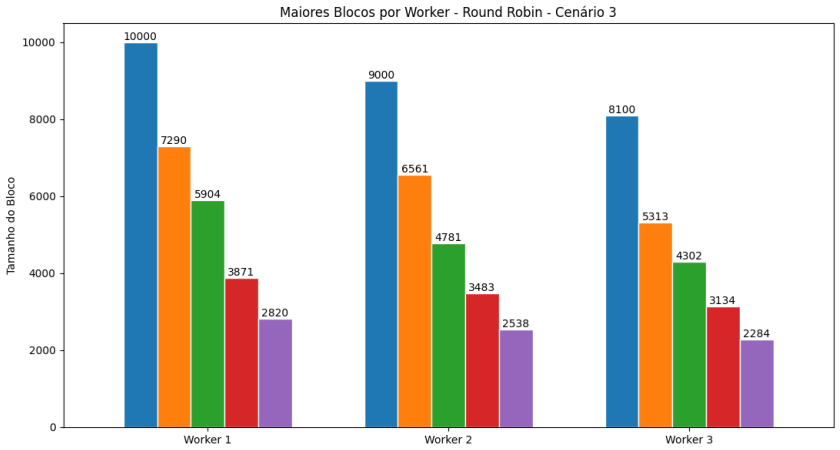
Quadro 4 – Resultados de execução dos algoritmos para o cenário 3.

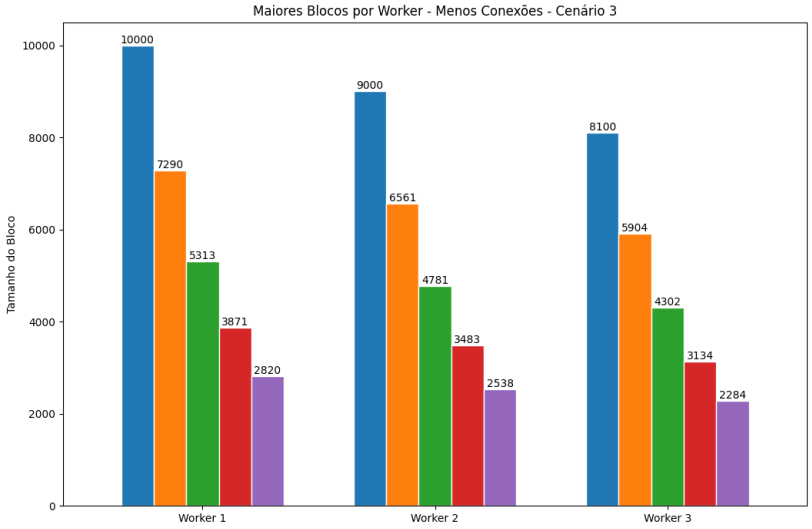
Algoritmo	**Tempo Total de Execução**	**Tempo Médio de Execução de Bloco por *worker***	**Total de Blocos Enviados para cada *worker***	**Maior Tempo por *worker***
Round Robin	27m 14s	*worker* 1: 166 s *worker* 2: 117 s *worker* 3: 81 s	*worker* 1: 24 *worker* 2: 24 *worker* 3: 24	*worker* 1: 27m 14s s *worker* 2: 16m 18s *worker* 3: 13m 35s
Round Robin Ponderado	29m 25s	*worker* 1: 209 s *worker* 2: 64 s *worker* 3: 26 s	*worker* 1: 32 *worker* 2: 24 *worker* 3: 16	*worker* 1: 29m 25s *worker* 2: 198 s *worker* 3: 154 s
Menos Conexões	27m 34s	*worker* 1: 162 s *worker* 2: 111 s *worker* 3: 86 s	*worker* 1: 24 *worker* 2: 24 *worker* 3: 24	*worker* 1: 27m 34s *worker* 2: 968 s *worker* 3: 838 s
Menos Conexões Ponderado	28m 5s	*worker* 1: 151 s *worker* 2: 110 s *worker* 3: 81 s	*worker* 1: 32 *worker* 2: 24 *worker* 3: 16	*worker* 1: 28m 5s *worker* 2: 962 s *worker* 3: 759 s
Menor Tempo de Resposta	27m 15s	*worker* 1: 163 s *worker* 2: 109 s *worker* 3: 86 s	*worker* 1: 24 *worker* 2: 24 *worker* 3: 24	*worker* 1: 27m 15s *worker* 2: 976 s *worker* 3: 842 s

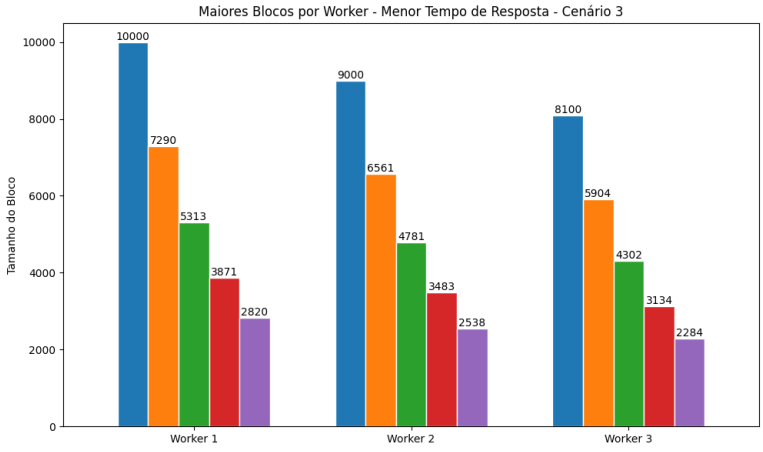
Fonte: Elaborada pelo autor.

Analisando os resultados, percebe-se que, mesmo distribuindo os blocos proporcionalmente entre os *workers*, ainda houveram diferenças consideráveis entre os tempos de execução. Para entender por que esta discrepância ocorreu, é necessário analisar os blocos enviados para cada *worker*, apresentado nas Figuras 16 e 17

Figura 16 – Gráficos de maiores blocos enviados para cada *worker* pelos algoritmos *Round Robin*, Menos Conexões e Menor Tempo de Resposta para o cenário 3.

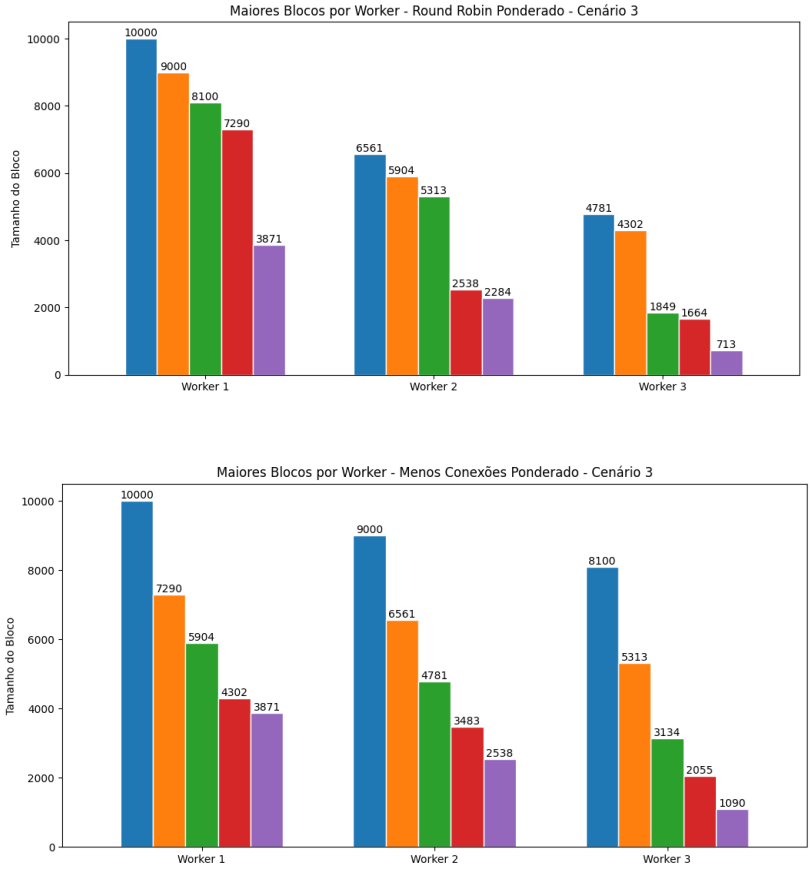






Fonte: Elaborada pelo autor.

Figura 17 – Gráficos de maiores blocos enviados para cada *worker* pelos algoritmos ponderados para o cenário 3.

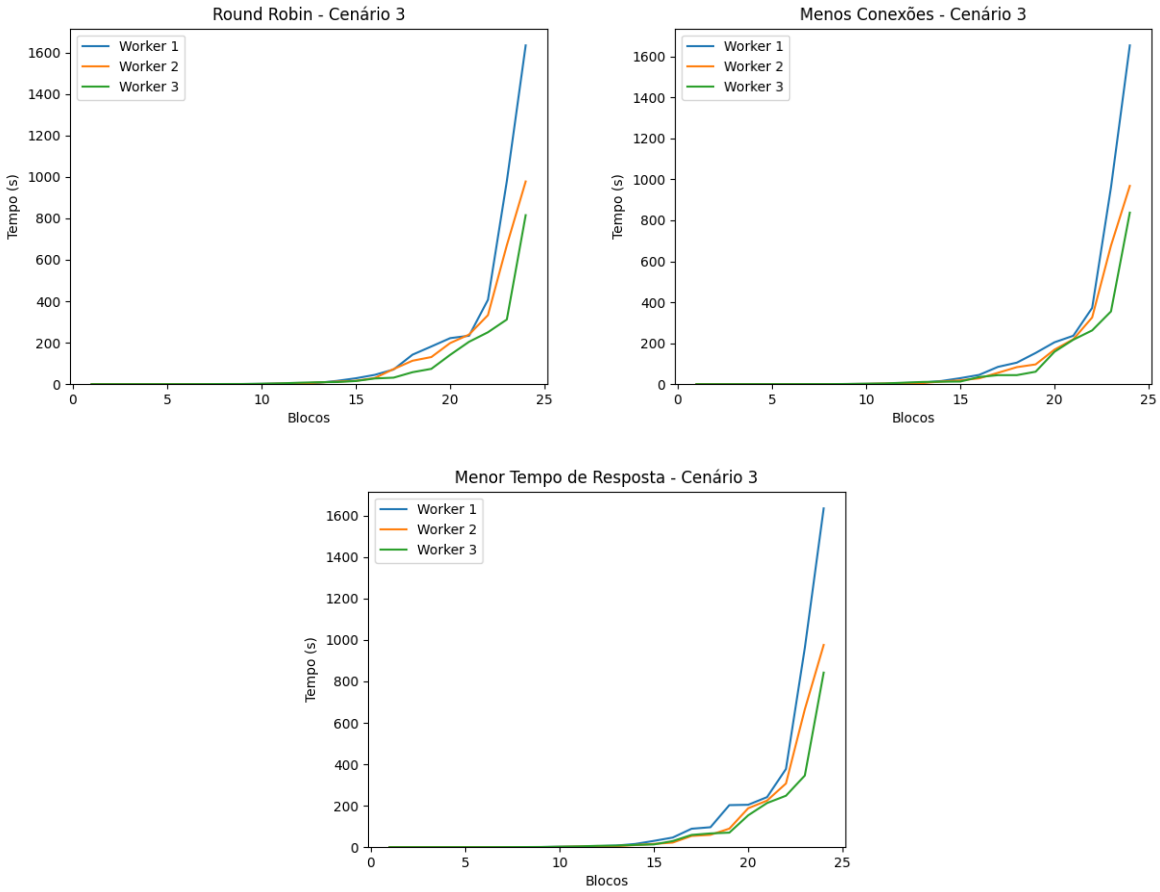


Fonte: Elaborada pelo autor.

Analisando as figuras, percebe-se que o *worker* 1 recebeu os maiores blocos em todos os casos, assim como o *worker* 2 recebeu blocos maiores que o *worker* 3. O tempo de processamento aumenta de forma exponencial à medida que o tamanho do bloco aumenta. Mesmo com uma diferença de elementos de apenas 10% entre os blocos subsequentes, o tempo de execução aumenta significativamente.

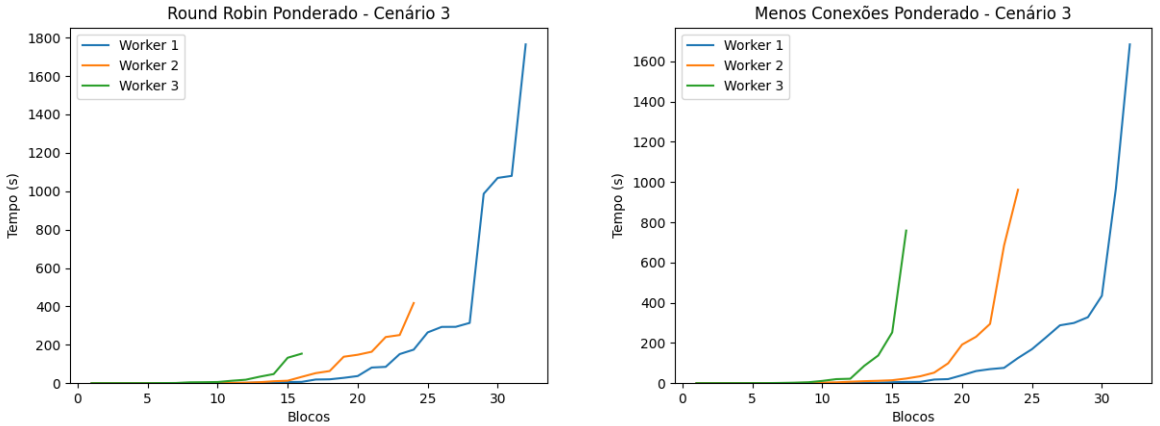
Além das diferenças no tempo de execução dos blocos, analisando os gráficos de tempo de execução por bloco (Figuras 18 e 19), percebe-se que os *workers* que receberam blocos menores ficaram muito mais ociosos, mesmo com a quantidade de blocos sendo distribuída de forma proporcional.

Figura 18 – Gráficos de tempo de execução dos algoritmos *Round Robin*, Menos Conexões e Menor Tempo de Resposta para o cenário 3.



Fonte: Elaborada pelo autor.

Figura 19 – Gráficos de tempo de execução dos algoritmos ponderados para o cenário 3.



Fonte: Elaborada pelo autor.

Neste cenário o algoritmo Menos Conexões apresentou uma certa vantagem em relação

aos outros, pois, em um cenário com uma carga de trabalho mais pesada, a carga atual dos *nodes* deveria ser considerada para a escolha do *node*. Apesar do algoritmo de Menor Tempo de Resposta considerar a carga, a média de execução não é uma boa forma de medir a carga de trabalho atual do *node* devido ao crescimento exponencial do tempo de execução. Para os algoritmos ponderados, o algoritmo Menos Conexões ponderado apresentou um resultado melhor que o *Round Robin* Ponderado, que, mesmo alocando mais blocos para o *node* com maior peso, alocou a carga de forma desproporcional.

4.4.4 Cenário 4

O Quadro 5 mostra os resultados obtidos por cada algoritmo para o Cenário 4, que apresentou resultados melhores em comparação ao cenário 3. Ainda houveram desbalanceamentos da distribuição de blocos, mas estes foram menos acentuados.

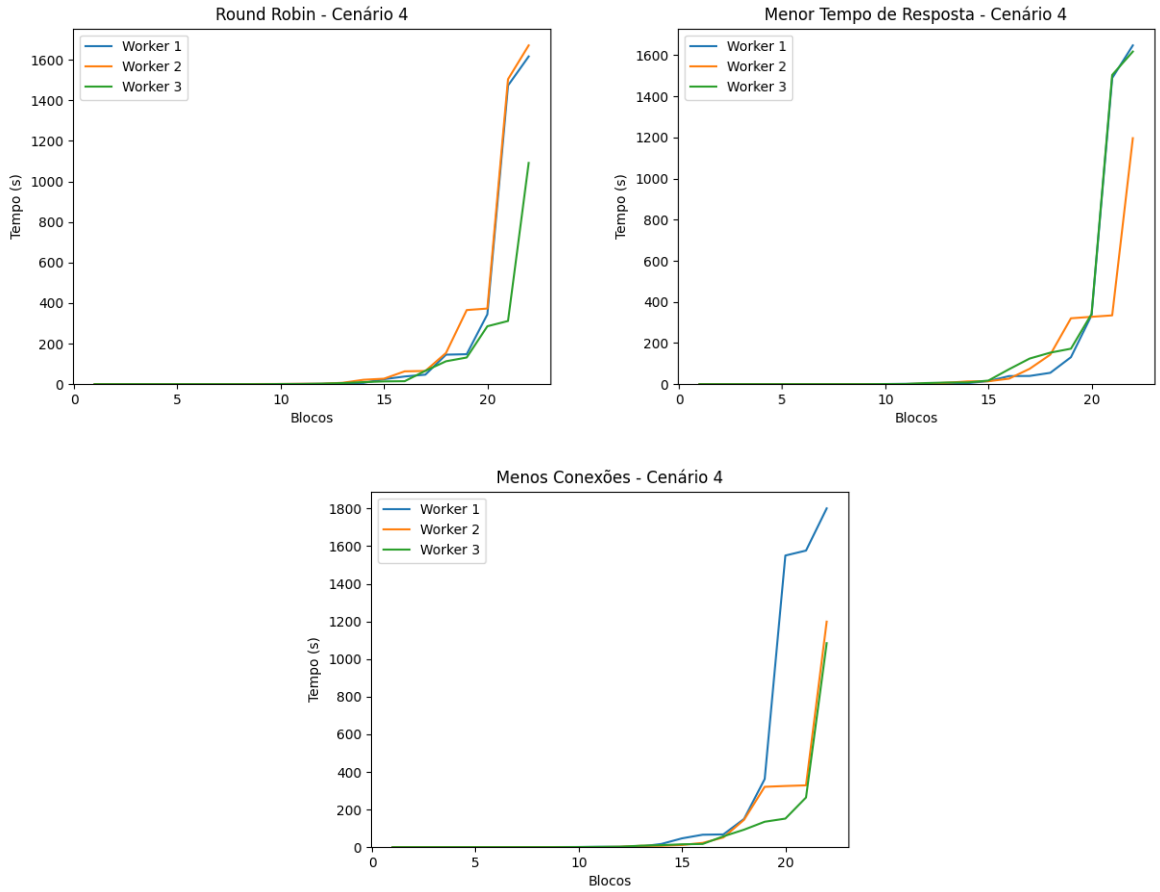
Quadro 5 – Resultados de execução dos algoritmos para o cenário 4.

Algoritmo	**Tempo Total de Execução**	**Tempo Médio de Execução de Bloco por *worker***	**Total de Blocos Enviados para cada *worker***	**Maior Tempo por *worker***
Round Robin	27m 51s	*worker* 1: 175 s *worker* 2: 194 s *worker* 3: 93 s	*worker* 1: 22 *worker* 2: 22 *worker* 3: 22	*worker* 1: 26m 56s s *worker* 2: 27m 50s *worker* 3: 18m 11s
Round Robin Ponderado	30m 42s	*worker* 1: 247 s *worker* 2: 91 s *worker* 3: 37 s	*worker* 1: 31 *worker* 2: 21 *worker* 3: 14	*worker* 1: 30m 41s *worker* 2: 17m 30s *worker* 3: 185 s
Menos Conexões	30m 1s	*worker* 1: 257 s *worker* 2: 110 s *worker* 3: 84 s	*worker* 1: 22 *worker* 2: 22 *worker* 3: 22	*worker* 1: 30m 1s *worker* 2: 19m 59s *worker* 3: 18m 5s
Menos Conexões Ponderado	28m 53s	*worker* 1: 168 s *worker* 2: 97 s *worker* 3: 217 s	*worker* 1: 29 *worker* 2: 22 *worker* 3: 15	*worker* 1: 28m 52s *worker* 2: 19m 14s *worker* 3: 26m 58s
Menor Tempo de Resposta	27m 28s	*worker* 1: 171 s *worker* 2: 112 s *worker* 3: 183 s	*worker* 1: 22 *worker* 2: 22 *worker* 3: 22	*worker* 1: 27m 28s *worker* 2: 19m 57s *worker* 3: 26m 58s

Fonte: Elaborada pelo autor.

Os algoritmos *Round Robin* e Menor Tempo de Resposta apresentaram uma característica em ambos os casos, no qual dois *workers* receberam uma carga aproximada e um *worker* recebeu menos carga. Ao contrário disso, o Menos Conexões apresentou dois *workers* com cargas semelhantes e um *worker* com mais carga que os outros. Os gráficos de tempo de execução desses algoritmos (Figura 20) confirmam esse fato.

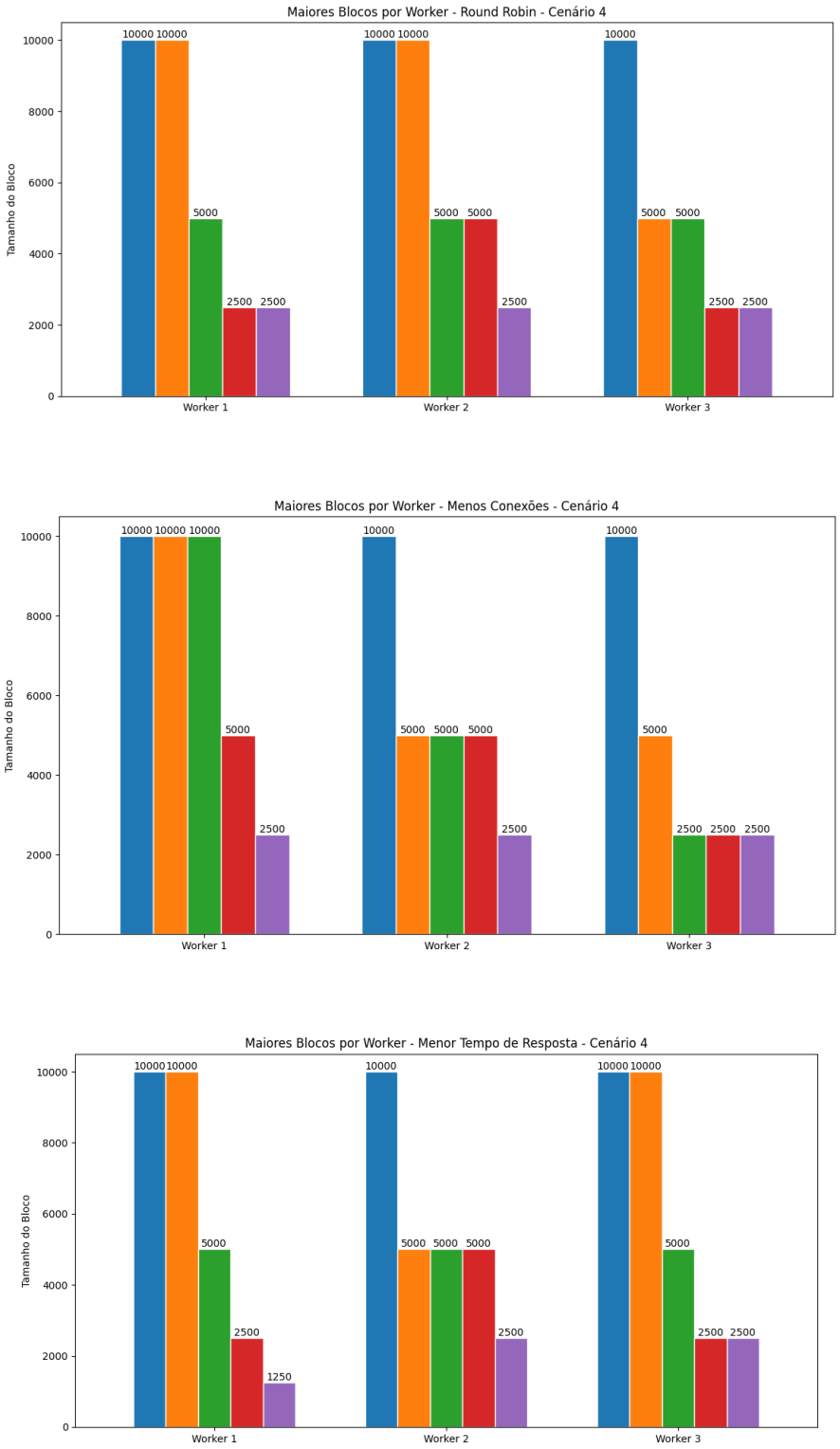
Figura 20 – Gráficos de tempo de execução dos algoritmos *Round Robin*, Menos Conexões e Menor Tempo de Resposta para o cenário 4.



Fonte: Elaborada pelo autor.

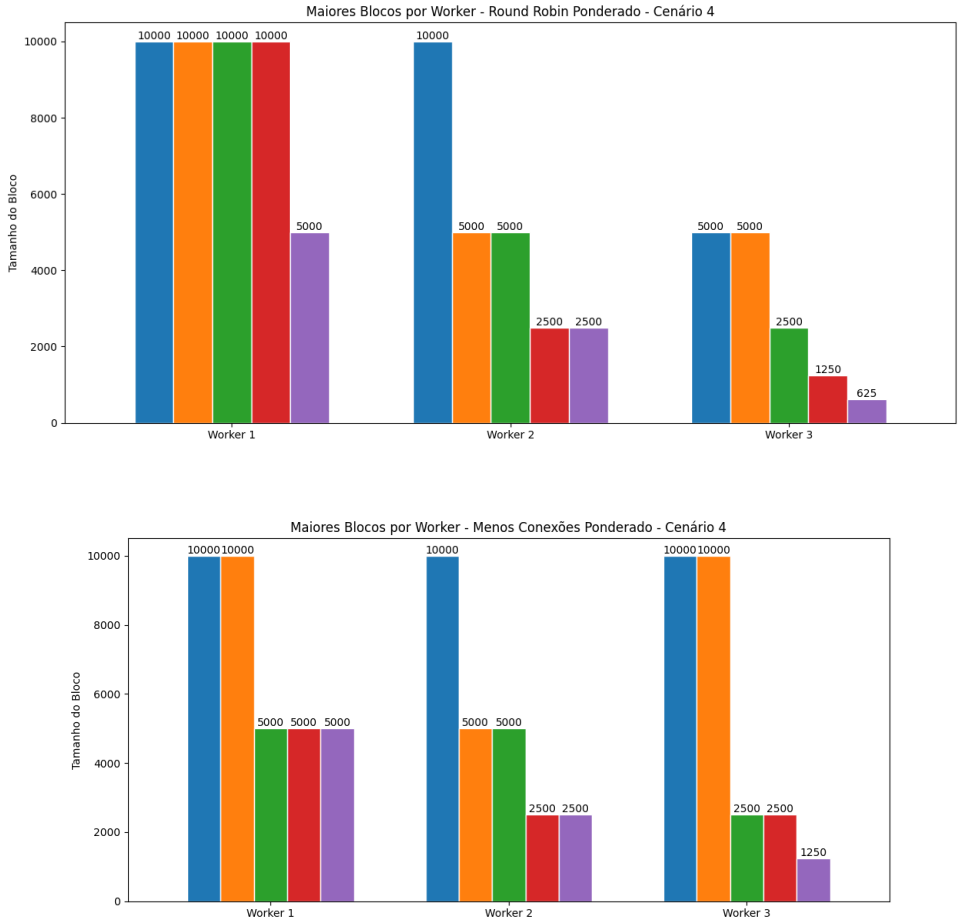
Analisando a distribuição dos maiores blocos (figuras 21 e 22), percebe-se que, no caso do Menos Conexões, o *worker* 1 recebeu uma proporção de blocos grandes bem maior que os outros dois *workers*. No caso dos algoritmos *Round Robin* e Menor Tempo de Resposta, os blocos de 10 mil foram distribuídos de forma proporcional entre os 3 *workers*. Como a diferença de tamanho entre os blocos é maior que no cenário 3, as filas de processamento são menores e o balanceamento ocorreu de forma mais igualitária.

Figura 21 – Gráficos de maiores blocos enviados para cada *worker* pelos algoritmos *Round Robin*, Menos Conexões e Menor Tempo de Resposta para o cenário 4.



Fonte: Elaborada pelo autor.

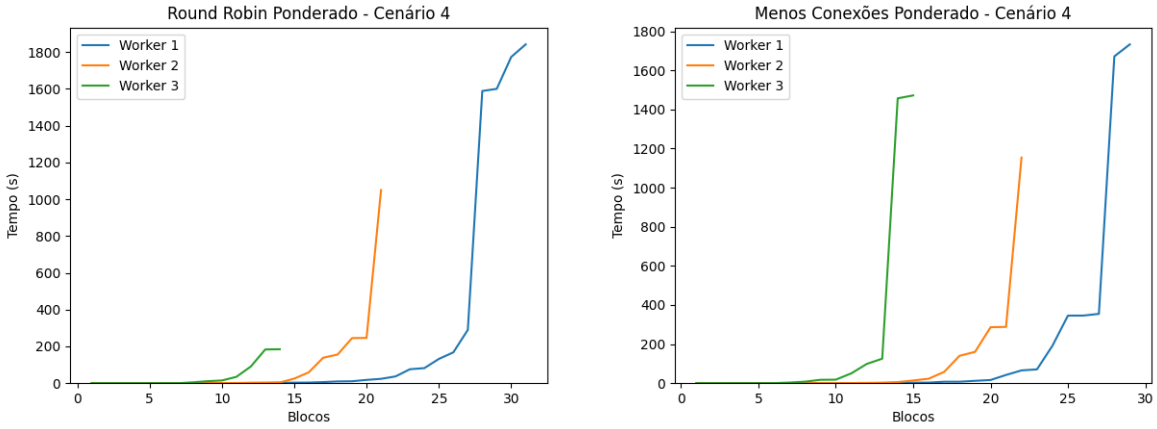
Figura 22 – Gráficos de maiores blocos enviados para cada *worker* pelos algoritmos ponderados para o cenário 4.



Fonte: Elaborada pelo autor.

Nos algoritmos ponderados, o *Round Robin* ponderado obteve um o pior resultado, alocando a maioria dos blocos de 10 mil elementos para o *worker* com maior peso. Já o Menos Conexões distribuiu de forma mais desproporcional, alocando dois blocos de 10 mil ao *worker* com menor peso. As diferenças no tempo de processamento do *Round Robin* ponderado ficaram distribuídas de forma mais proporcional que o Menos Conexões ponderado, mas ainda sim apresentou um desbalanceamento do *worker* 1 em relação aos demais. Os gráficos de tempo de execução (Figura 23) demonstram esses desbalanceamentos.

Figura 23 – Gráficos de tempo de execução dos algoritmos ponderados para o cenário 4.



Fonte: Elaborada pelo autor.

Para este cenário, o algoritmo Menor Tempo de Resposta se mostrou superior, pois os tempos médios de execução ficaram mais aproximados. O Menos Conexões alocou blocos grandes de forma desproporcional a um único *worker*, criando um desbalanceamento nos tempos de execução. Nos gráficos de distribuição de maiores *workers* dá para perceber o porquê disso ter ocorrido: os blocos maiores foram distribuídos de forma mais proporcional, e os subsequentes, que apresentam um tempo de execução consideravelmente menor, foram alocados para o *worker* com somente um bloco de 10 mil.

O algoritmo *Round Robin* Ponderado se mostrou superior ao Menos Conexões Ponderado, pois, ao analisar os gráficos de tempo de execução, percebe-se que os tempos de execução ficaram distribuídos de forma mais proporcional aos pesos alocados, além do fato dos blocos com maior peso terem sido enviados para o *worker* com maior peso.

5 CONCLUSÃO E TRABALHOS FUTUROS

Neste capítulo são apresentadas as conclusões do estudo e as principais perspectivas para trabalhos futuros. Investigou-se o comportamento de algoritmos de balanceamento de carga implementados para o processamento da tarefa de RE usando um *cluster* Erlang. Foi explorado a execução dos algoritmos em cenários em que não havia nenhum viés no tamanho dos blocos de entidades processados, bem como nos cenários em que o viés era considerado pequeno, médio ou grande.

5.1 Considerações Finais

Este trabalho apresentou o Elixir/Erlang como alternativa para a implementação de sistemas para resolução de entidades, utilizando as capacidades de distribuição do Erlang e sua programação orientada à concorrência. Para utilizar o Erlang distribuído de forma eficiente, é necessário haver formas de distribuir as tarefas entre os *nodes* de um *cluster*, e para isso foram analisados alguns algoritmos de balanceamento de carga implementados neste trabalho.

A execução dos algoritmos implementados no trabalho mostraram bons resultados, especialmente para dados mais balanceados. Há espaço para melhorias nos casos menos balanceados. Os algoritmos implementados podem ser utilizados como base para a solução de outros problemas que envolvam distribuição de tarefas intensivas em dados. Além disso, o trabalho demonstrou que o Erlang pode ser uma boa alternativa para a tarefa de resolução de entidades graças às suas capacidades de distribuição e concorrência.

5.2 Contribuições

Como contribuição, o trabalho oferece diversos algoritmos de balanceamento de carga para *clusters* Erlang. Os algoritmos podem ser usados não só para o problema da resolução de entidades, mas para outros tipos de tarefas que envolvem distribuição de tarefas intensivas em dados. Os algoritmos implementados, assim como os experimentos realizados no estudo de caso estão disponíveis em um repositório no Github[1]

5.3 Fatores Limitantes

O fato do estudo de caso ter sido implementado com todos os clusters em um único host pode ter afetado o resultado do estudo de caso, já que os *nodes* conectados compartilharam os recursos do host entre si. Devido ao alto custo da infraestrutura para esse tipo de sistema, os algoritmos ponderados não puderam ser testados de uma forma aproximada de um cenário real.

[1] https://github.com/thsl97/entity-resolution

5.4 Sugestões de Trabalhos Futuros

Como proposta futura com base neste trabalho, sugere-se estudar o uso de outros algoritmos de balanceamento dinâmico, visto que os algoritmos estáticos podem não executar de uma forma satisfatória em alguns casos. Um exemplo seria o balanceamento dinâmico com arquitetura produtor-consumidor, utilizando a biblioteca GenStage[2]. Outra sugestão seria o balanceamento utilizando bibliotecas de gerenciamento de *clusters*, como *libcluster* e *horde*, com a adição dinâmica e monitoramento de *nodes*.

[2] O GenStage é uma biblioteca Elixir usada para implementar fluxos de trabalho em estágios independentes.

REFERÊNCIAS

BARKLUND, J.; VIRDING, R. Erlang 4.7. 3 reference manual draft (0.7). *Ericsson AB*, p. 79, 1999. Citado 2 vezes nas páginas 22 e 23.

CHRISTEN, P. Febrl: a freely available record linkage system with a graphical user interface. In: *Proceedings of the second Australasian workshop on Health data and knowledge management - Volume 80*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2008. (HDKM '08), p. 17–25. ISBN 978-1-920682-61-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1385089.1385094>. Citado na página 18.

CHRISTEN, P. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642311636, 9783642311635. Citado 5 vezes nas páginas 13, 15, 17, 18 e 30.

CHRISTEN, P. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 24, n. 9, p. 1537–1555, set. 2012. ISSN 1041-4347. Disponível em: <http://dx.doi.org/10.1109/TKDE.2011.127>. Citado na página 18.

CHRISTEN, P.; GAYLER, R.; HAWKING, D. Similarity-aware indexing for real-time entity resolution. In: *Proceedings of the 18th ACM conference on Information and knowledge management*. New York, NY, USA: ACM, 2009. (CIKM '09), p. 1565–1568. ISBN 978-1-60558-512-3. Disponível em: <http://doi.acm.org/10.1145/1645953.1646173>. Citado na página 15.

COHEN, W. W.; RICHMAN, J. Learning to match and cluster large high-dimensional data sets for data integration. In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM, 2002. (KDD '02), p. 475–480. ISBN 1-58113-567-X. Disponível em: <http://doi.acm.org/10.1145/775047.775116>. Citado na página 18.

GHOMI, E. J.; RAHMANI, A. M.; QADER, N. N. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, Elsevier, v. 88, p. 50–71, 2017. Citado 2 vezes nas páginas 14 e 21.

HERNáNDEZ, M. A.; STOLFO, S. J. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 2, n. 1, p. 9–37, jan. 1998. ISSN 1384-5810. Disponível em: <http://dx.doi.org/10.1023/A:1009761603038>. Citado na página 18.

HEXDOCS. 2023. Disponível em: <https://hexdocs.pm/elixir/GenServer.html>. Citado na página 23.

JARO, M. A. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, v. 84, n. 406, p. 414–420, 1989. Citado na página 18.

JURIC, S. *Elixir in action*. [S.l.]: Simon and Schuster, 2019. Citado 5 vezes nas páginas 13, 14, 21, 23 e 25.

KOLB, L.; THOR, A.; RAHM, E. Load balancing for mapreduce-based entity resolution. In: *Proceedings of the 28th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2012. (ICDE '12), p. 618–629. ISBN 978-0-7695-4747-3. Disponível em: <http://dx.doi.org/10.1109/ICDE.2012.22>. Citado na página 19.

KOLB, L.; THOR, A.; RAHM, E. Multi-pass sorted neighborhood blocking with mapreduce. *Comput. Sci.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 27, n. 1, p. 45–63, fev. 2012. ISSN 1865-2034. Disponível em: <http://dx.doi.org/10.1007/s00450-011-0177-x>. Citado na página 15.

KOPCKE, H.; RAHM, E. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 69, n. 2, p. 197–210, fev. 2010. ISSN 0169-023X. Disponível em: <http://dx.doi.org/10.1016/j.datak.2009.10.003>. Citado na página 13.

LEE, Y. W. et al. *Journey to Data Quality*. [S.l.]: The MIT Press, 2006. ISBN 0262122871. Citado na página 15.

LOGAN, M.; MERRITT, E.; CARLSSON, R. *Erlang and OTP in Action*. [S.l.]: Manning Publications Co., 2010. Citado na página 22.

MESTRE, D. G. *Leveraging the entity matching performance through adaptive indexing and efficient parallelization*. Tese (tese de doutorado) — Centro de Engenharia Elétrica e Informática - CEEI. Universidade Federal de Campina Grande, Setembro 2018. Citado 3 vezes nas páginas 13, 17 e 19.

MESTRE, D. G.; PIRES, C. E. S.; NASCIMENTO, D. C. Towards the efficient parallelization of multi-pass adaptive blocking for entity matching. *Journal of Parallel and Distributed Computing*, v. 101, p. 27 – 40, 2017. ISSN 0743-7315. Citado na página 15.

NEUMAN, B. C. ord. Scale in distributed systems. *ISI/USC*, p. 68, 1994. Citado na página 21.

PAPADAKIS, G. et al. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 53, n. 2, p. 1–42, 2020. Citado 2 vezes nas páginas 13 e 15.

PAPENBROCK, T.; HEISE, A.; NAUMANN, F. Progressive duplicate detection. *IEEE Transactions on Knowledge and Data Engineering*, v. 27, n. 5, p. 1316–1329, May 2015. ISSN 1041-4347. Citado na página 15.

SHAH, N.; FARIK, M. Static load balancing algorithms in cloud computing: Challenges & solutions. *International Journal of Scientific & Technology Research*, v. 4, n. 10, p. 365–367, 2015. Citado na página 21.

SHARMA, S.; SINGH, S.; SHARMA, M. Performance analysis of load balancing algorithms. *International Journal of Civil and Environmental Engineering*, v. 2, n. 2, p. 367–370, 2008. Citado na página 14.

TANENBAUM, A.; STEEN, M. van. *Distributed Systems*. CreateSpace Independent Publishing Platform, 2023. ISBN 9781543057386. Disponível em: <https://books.google.com.br/books?id=c77GAQAACAAJ>. Citado 2 vezes nas páginas 13 e 20.