**UNIVERSIDADE FEDERAL DO CEARÁ**

**DEPARTAMENTO DE COMPUTAÇÃO**

**CURSO DE CIÊNCIA DA COMPUTAÇÃO**

**MATHEUS HENRIQUE MACHADO PERICINI**

# UTILIZAÇÃO DE METAHEURISTICAS PARA BALANCEAMENTO DE CARGA EM AMBIENTES MAPREDUCE

**FORTALEZA**

**2017**

MATHEUS HENRIQUE MACHADO PERICINI

# UTILIZAÇÃO DE METAHEURISTICAS PARA BALANCEAMENTO DE CARGA EM AMBIENTES MAPREDUCE

Dissertação apresentada ao Curso de Curso de Ciência da Computação do Departamento de Computação do da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação.

Área de Concentração: Banco de Dados

Orientador: Prof. Dr. Javam de Castro Machado

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

---

P519u Pericini, Matheus Henrique Machado.
Utilização de Meta-heurísticas para Balanceamento de Carga em Ambientes Mapreduce / Matheus Henrique Machado Pericini. – 2017.
72 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2017.
Orientação: Prof. Dr. Javam de Castro Machado.

1. Mapreduce. 2. Hadoop. 3. Skew. 4. Particionamento. 5. Meta-heurística. I. Título.

CDD 005

MATHEUS HENRIQUE MACHADO PERICINI

# UTILIZAÇÃO DE METAHEURISTICAS PARA BALANCEAMENTO DE CARGA EM AMBIENTES MAPREDUCE

Dissertação apresentada ao Curso de Curso de Ciência da Computação do Departamento de Computação do da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação.

Área de Concentração: Banco de Dados

Aprovada em:

BANCA EXAMINADORA

---

Prof. Dr. Javam de Castro Machado (Orientador)
Universidade Federal do Ceará - UFC

---

Francisco Heron de Carvalho Junior
Universidade Federal do Ceará - UFC

---

Sérgio Lifschitz
Pontificia Universidade Catolica PUC - RJ

À todas as pessoas que acreditaram em mim. Em especial à minha mãe, a minha avó e a minha noiva, as três mulheres da minha vida.

## AGRADECIMENTOS

Ao Prof. Dr. Javam de Castro Machado por me orientar em meu mestrado.

Ao doutorando Lucas Gonçalves, pelo auxílio dado durante todo o meu mestrado.

Aos Professores Dr. Flavio Rubens de Carvalho Sousa, Dr. Leonardo Oliveira Moreira e Me. Victor Aguiar Evangelista de Farias por terem me auxiliado durante meu mestrado.

Aos membros do Laboratório de Sistemas e Banco de Dados (LSBD) pelo seu companheirismo, apoio, e por propiciarem um ambiente em que eu pudesse me desenvolver.

À minha mãe, a minha avó e a minha noiva, que me deram total apoio e incentivo durante todo o meu período acadêmico!

Ao meu querido avô que não pode estar comigo durante toda a trajetória do meu mestrado.

Ao Doutorando em Engenharia Elétrica, Ednardo Moreira Rodrigues, e seu assistente, Alan Batista de Oliveira, aluno de graduação em Engenharia Elétrica, pela adequação do *template* utilizado neste trabalho para que o mesmo ficasse de acordo com as normas da biblioteca da Universidade Federal do Ceará (UFC).

Agradeço a todos os professores por me proporcionar o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, por tanto que se dedicaram a mim, não somente por terem me ensinado, mas por terem me feito aprender.

E ao CNPQ pelo financiamento da pesquisa de mestrado via bolsa de estudos.

"O sonho é que leva a gente para frente. Se a gente for seguir a razão, fica aquietado, acomodado."

(Ariano Suassuna)

## RESUMO

Com o aumento do número de dados obtidos por grandes empresas, foi necessário elaborar novas estratégias para o processamento desses dados de modo a manter sua relevância e aproveitar suas informações. Uma das estratégias que tem sido amplamente utilizada tem como base um modelo de programação, chamado *MapReduce*, que utiliza divisão e conquista para processar os dados em um *cluster* de máquinas. O *Hadoop* é uma das implementações mais consolidadas do modelo de *MapReduce*. Mas mesmo tal estratégia é passível de melhorias. Nela o tempo de execução é dependente de todas as máquinas fazendo com que qualquer máquina sobrecarregada gere um atraso na entrega do resultado. Essa sobrecarga é causada por um problema chamado comumente de *Data Skew* que consiste em uma divisão desigual dos dados causado pelo tamanho dos dados, o modo como eles são divididos, ou o processamento desigual dos dados. Visando resolver esse problema, propusemos o MALiBU, uma melhoria da estratégia de execução do *MapReduce* que particiona os dados entre as máquinas usando uma meta-heurística dentre elas *Simulated Annealing*, *Local Beam Search* ou *Stochastic Beam Search*. Resultados experimentais mostraram melhorias no desempenho do *MapReduce* quando se faz uso de meta-heurística para distribuir os dados entre as máquinas, bem como mostraram, dentre as três meta-heurísticas avaliadas, qual delas melhor balanceia a carga.



**Keywords:** Mapreduce. Hadoop. Skew. Particionamento. Meta-heurística.

## ABSTRACT

With the increase in the number of data obtained by large companies, it was necessary to elaborate new strategies for the processing of this data in order to maintain the relevance of the information that they contain. One of the strategies that has been widely used is based on a programming model, called *MapReduce*, which uses division and conquest to process the data in a *cluster* of machines. Hadoop is one of the most consolidated implementations of the MapReduce model. But even such a strategy is subject to improvement. In it, the runtime depends on all the machines causing any overloaded machine to generate a delay in the delivery of the result. This overhead is caused by a problem commonly called *Data Skew* which consists of an unequal division of data, either by the size of the data or by the way it is divided. In order to solve this problem, we have proposed the MALiBU, an improvement of the execution strategy of *Hadoop*, which partitions the data between the machines using a meta-heuristic among them *Simulated Annealing*, *Local Beam Search* or *Stochastic Beam Search*. Experimental results showed improvements in the performance of Hadoop when using metaheuristics to distribute the data among the processing elements of the model, as well as among the three meta-heuristics evaluated, which has the best results.



**Keywords:** MapReduce. Hadoop. Skew. Partitioning. Metaheuristics.

## LISTA DE ILUSTRAÇÕES

## LISTA DE TABELAS

## LISTA DE ALGORITMOS

## LISTA DE SÍMBOLOS

| | |
|---|---|
| $C$ | Cluster de máquinas |
| $k$ | Chave |
| $v$ | Valor |
| $t$ | Tupla |
| $p$ | Partição |
| $P$ | Particionamento |
| $X$ | *Threshold* |
| $Y$ | Número de passos |
| $\beta$ | Número de particionamento por passos |
| $T(p)$ | Função que calcula o tamanho de uma partição |
| $F(P)$ | Função que calcula o desbalanceamento de um particionamento |
| $P_opt$ | Particionamento ótimo |
| $M(P)$ | Função que calcula a menor partição de um particionamento |
| $S(P)$ | Função que modifica um particionamento |
| $H(P)$ | Função de balanceamento por *Hill Climb* |
| $B(P)$ | Função de balanceamento por meta-heurística |

**SUMÁRIO**

# 1 INTRODUÇÃO

## 1.1 MOTIVAÇÃO

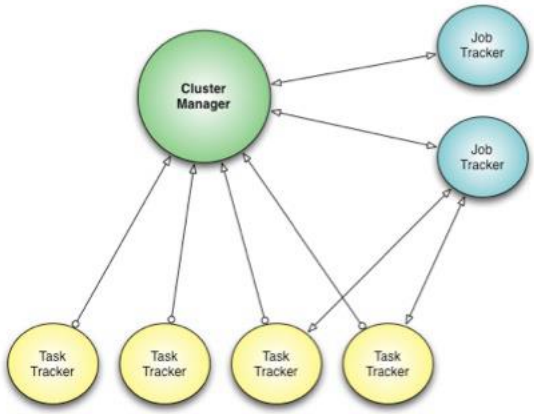
Conforme as tecnologias relacionadas à nuvem, sensores, e computação científica avançam, cada vez mais dados são produzidos por aplicações que as utilizam, demandando técnicas mais eficientes para armazenar e analisar tais dados, tais como *Association Rule Learning*, *Classification Tree Analysis*, *Genetic Algorithms*, *Machine Learning*, *Regression Analysis*, *Sentiment Analysis* e analises de redes sociais.

Uma das estratégias para gerar e analisar grandes conjuntos de dados é a utilização do *MapReduce* (DEAN; GHEMAWAT, 2008), um modelo de computação distribuída que divide o conjunto de dados para ser computado por um grupo de máquinas chamado *cluster*, evitando a necessidade do uso de máquinas com alto poder de processamento e tornando o processo mais robusto e escalável por melhor conseguir administrar seus recursos e pela fácil inclusão de máquinas adicionais. A estrutura padrão do *MapReduce* consiste na execução de duas operações: *map* e *reduce*. Na fase de *map*, o nó mestre divide os dados de entrada em grupos menores e utiliza um *Node Manager* para transmitir esses grupos para os nós trabalhadoras. Os nós que recebem os grupos de dados os separam em dados intermediários, armazenando-os e enviando-os para o nó mestre que os coordena através de um *job tracker*. Na fase de *reduce*, o nó mestre distribui os dados intermediários separados no passo anterior de acordo com uma regra de particionamento. Os nós responsáveis pela fase de *reduce* coletam os dados intermediários de acordo com o particionamento criado, os processam e combinam, para formar a saída para o problema original.

Essa estratégia de divisão e conquista permite que o *MapReduce* processe grandes quantidades de dados em um *cluster* distribuído. Além disso, o usuário de um modelo *MapReduce* precisa apenas implementar as funções de *map* e *reduce*, deixando todas as outras operações e cuidados com as máquinas para o sistema do *cluster* que gerencia os recursos, executa a divisão de tarefas e distribui os dados de maneira independente. Muitas companhias de grande porte, tais como *Google*, *Amazon*, *Facebook* e *Yahoo!*, utilizam o *MapReduce* para processar seus grandes volumes de dados (LIU QI ZHANG, 2016). Na Figura 1, damos o exemplo do modelo de *MapReduce* usado pelo *Facebook* chamado *Corona* (FACEBOOK, 2012), que adiciona um *cluster manager* para monitorar a carga e o recurso do *cluster* e um *job tracker* para cada trabalho, permitindo rodar múltiplos trabalhos em paralelo, cujos recursos são monitorados e redistribuídos

sempre que possível, garantindo uma melhor utilização e independência do *cluster*.

Figura 1 – Corona: representação da estratégia de *MapReduce* usada pelo *Facebook*




Apesar das diversas vantagens apresentadas pelo *MapReduce*, sua simplicidade pode levar a problemas ao processar aplicações ou dados específicos. Referências envolvendo *MapReduce* consideram que os dados a serem processados são divididos de maneira igualitária entre as partições para a fase de *reduce*.

Entretanto, em muitas aplicações do mundo real os dados não são distribuídos de maneira uniforme e a função de particionamento escolhida pode não garantir que a alocação dos dados para as máquinas seja balanceada. Esse fenômeno é chamado *Partitioning Skew*, um problema que ocorre quando a distribuição dos dados entre as máquinas está desbalanceada ou parte dos dados precisa de um maior poder computacional para ser processado do que os outros. Isso resulta em algumas máquinas se tornarem sobrecarregadas durante a fase de *reduce* enquanto outras se tornam ociosas ao já terem completado suas tarefas. A ociosidade de certas máquinas significa um desperdício de recursos que poderiam estar sendo aproveitados executando outras operações. E a sobrecarga de dados costuma causar um aumento no tempo de execução das máquinas em processamento o que, consequentemente, atrasa a execução como um todo uma vez que o nó mestre depende do fim da execução de todas as máquinas.

O problema do *partitioning skew* e possíveis soluções foi abordado em diversos trabalhos recentes (RAMAKRISHNAN *et al.*, 2012; GUFLER *et al.*, 2011; KWON Y., 2012). Algumas referências tratam o problema de maneira *offline*, isto é, esperam que a fase de *map* seja concluída para obter todas as informações dos dados intermediários e efetuar o melhor balanceamento possível com todas as chaves presentes, ou colhem uma amostra do conjunto de dados antes da fase de *map* e, a partir dela, executam um pré-processamento dos dados tentando

prever sua distribuição. Porém, tais estratégias levam a uma sobrecarga da rede e um excesso de operações de entrada e saída. Essa sobrecarga é causada ao para a fase intermediária de *shuffle*, que ocorre entre as fases de *map* e *reduce*, fazendo-a esperar mais dados intermediários do que o necessário (GUFLER *et al.*, 2011).

Outra estratégia é realocar parte dos dados ainda não processados em máquinas que já tenham processado sua carga designada e estejam ociosas (KWON Y., 2012) através de um monitor de tarefas que avisa o nó mestre quando um nó trabalhador está disponível. Apesar dessa última estratégia melhorar o desempenho de grandes conjuntos de dados, ela gera uma grande quantidade de sobrecarga, principalmente considerando pequenos grupos de dados. Isso se deve às contínuas interrupções nas execuções de máquinas em trabalho para realizar a divisão de sua carga com as máquinas ociosas.

Uma maneira de melhor entender esse problema é moldá-lo como um problema já existente na literatura. No caso, uma variação do problema de escalonamento onde queremos atribuir uma distribuição de tarefas para um grupo de processadores, de modo que queremos que todos os processadores terminem ao mesmo tempo. Ou seja, precisamos que todas as partições possuam a mesma quantidade de chaves para que elas terminem em tempos parecidos. O conjunto solução para esse tipo de problema é demasiadamente amplo para se considerar um único caminho para a solução ideal. Como não temos controle sobre quais dados entram e em que ordem, procuramos trabalhar com opções que rapidamente consigam checar as possíveis soluções de maneira eficiente.

Com essa finalidade, criamos o MALiBU, uma solução para o problema de *Partitioning Skew* que substitui a fase de particionamento pela utilização de uma meta-heurísticas: *Simulated Annealing*, *Local Beam Search* ou *Stochastic Beam Search*; para executar as tomadas de decisão para achar uma distribuição mais equilibrada dos dados intermediário.

A escolha das meta-heurísticas citadas anteriormente se deve, principalmente, à sua simplicidade. Quando o resultado novo é melhor, a *Simulated Annealing* o escolhe. Entretanto, quando o resultado novo é pior, ao invés dele ser descartado e dado uma chance para que o mesmo ajude a guiar a solução para uma configuração melhor saindo de um ótimo local e indo para o caminho do ótimo global. Já a as meta-heurísticas *Local Beam Search* e *Stochastic Beam Search* permitem configurar quantos caminhos serão explorados por operação, com a diferença de que a *Local Beam Search* escolhe apenas os melhores caminhos enquanto a *Stochastic Beam Search* permite que qualquer caminho seja escolhido com uma chance baseada na qualidade do

mesmo, o que a aproxima da *Simulated Annealing* opera.

## 1.2 OBJETIVOS

### *1.2.1 Objetivo Geral*

Reduzir o impacto da má distribuição de dados (*partitioning skew*) causado pelo mau particionamento dos dados intermediários.

### *1.2.2 Objetivos Específicos*

Conforme apresentado no Objetivo Geral queremos:

- Melhorar o particionamento dos dados intermediários alterando a fase de particionamento do *MapReduce*.
- Utilizar meta-heurísticas para efetuar o particionamento de conjuntos de dados entre máquinas alocadas pra a fase de *reduce*

## 1.3 CONTRIBUIÇÕES

Como resultado desta dissertação, criamos:

- Uma estratégia mais eficiente de balancear a carga particionada para a fase de *reduce*.
- Diversas comparações entre meta-heurísticas e outras formas de particionamento como soluções alternativas para o padrão do *Hadoop*

### *1.3.1 Produção Científica*

Parte das contribuições contidas nesta dissertação possibilitaram a publicação do seguinte artigo:

- Matheus H. M. Pericini, Lucas G. M. Leite, Javam C. Machado. Malibu: Metaheuristics approach for online load balancing in mapreduce with skewed data input no XXXV Simpósio Brasileiro de Redes de Computadores (SBRC 2017).

## 1.4 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada da seguinte forma: No Capítulo 2, detalhamos o *Mapreduce*, falamos sobre *Skew* de particionamento e suas possíveis causas, e explicamos o funcionamento das meta-heurísticas. No Capítulo 3, dissertamos sobre trabalhos relacionados que buscam resolver o mesmo problema com estratégias diferentes. No Capítulo 4, apresentamos o nosso trabalho, explicamos como as meta-heurísticas funcionam para reduzir o problema do *Skew* de particionamento e mostramos seus algoritmos. No Capítulo 5, mostramos os resultados alcançados em diversos cenários utilizando um conjunto de dados real e um experimento clássico de *MapReduce* de contagem de palavras. Finalmente, no Capítulo 6, apresentamos a conclusão sobre o nosso trabalho e falamos sobre os trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Com o aumento da produção de dados graças à melhoria ao acesso a tecnologia, à inclusão de mais usuários no mundo digital, e ao aumento do número de objetos ligados à rede mundial de computadores, tanto o armazenamento desses dados quanto o estudo dos mesmos passou a ser feito de maneira distribuída. A distribuição desses dados e das tarefas relacionadas a eles passaram a gerar outros problemas e paradigmas para as computação. Dentre esses problemas, focamos no problema do balanceamento de carga atrapalhando a distribuição uniforme da carga de trabalho entre dois ou mais computadores, discos, ou outro tipo de recurso, com o intuito de otimizar a utilização desses recursos.

Sobre o problema do balanceamento de carga, o restante desse capítulo é dividido da seguinte maneira. A Seção 2.1 apresenta o modelo do *MapReduce* para computação distribuída que utiliza o conceito do balanceamento de carga para a divisão de tarefas muito grandes em tarefas menores que são distribuídas em um conjunto de máquinas. A Seção 2.2 trata do problema do *skew* que ocorre, dentre outros motivos, quando a carga não é particionada de maneira balanceada. Por fim, a Seção 2.3 introduz o uso de meta-heurísticas como uma maneira de melhor efetuar o balanceamento de carga.

## 2.1 MAPREDUCE

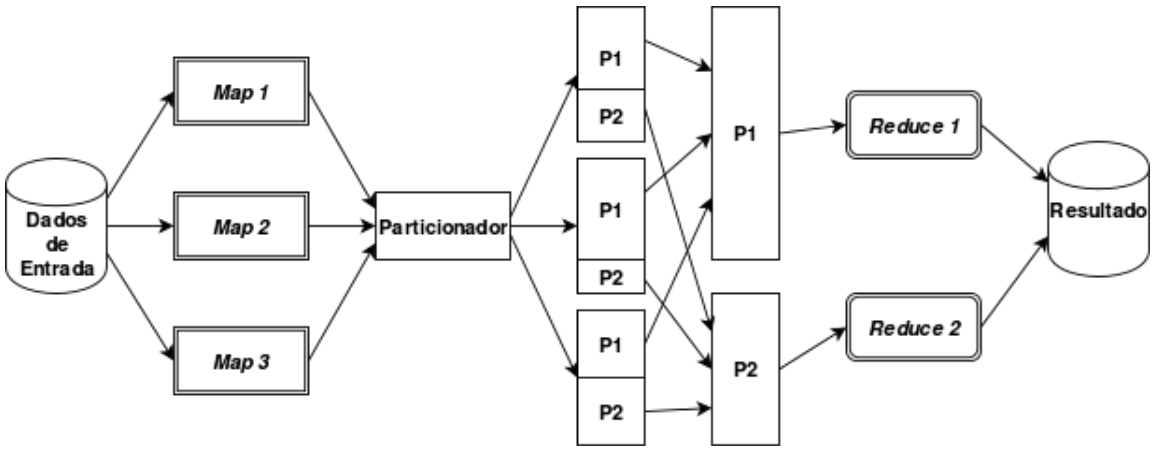
*MapReduce* (DEAN; GHEMAWAT, 2008) é um modelo computacional de programação distribuída que utiliza uma ou mais máquinas para formar um *cluster*. Esse modelo recebe uma conjunto de dados como entrada e os classifica, agrupa, e distribui por meio de duas funções principais que dão o nome ao modelo. A primeira função, denominada *map*, é responsável por receber o conjunto de dados inicial, dividido em blocos, e transformando-os em dados intermediários no formato de tuplas $< CHAVE, VALOR >$ em que $CHAVE$ é a representação de uma informação e $VALOR$ é a quantidade daquela informação. Após a transformação da entrada em dados intermediários, esses dados passam por uma função de particionamento que os dividem entre as máquinas agrupando tuplas usando o cálculo do valor *hash* da chave de cada tupla. As partições são distribuídas para as máquinas que executarão a segunda função, dada o nome de *reduce*, que, por sua vez, processa os dados intermediários contidos nas partições e retorna o resultado conforme mostrado na Figura 2.

A Figura 2 ilustra um exemplo do modelo de *MapReduce* com cinco nós, três de *map*

e dois de *reduce*. Utilizando uma função simples de contagem de palavras, que recebe um texto como entrada e entrega uma lista com o número de aparições de cada palavra como solução, para explicar o modelo teríamos os seguintes passos:

1. O texto inicial é dividido em três blocos de dados.
2. Cada bloco é enviado para um nó responsável por fazer o *map* onde as palavras serão separadas em tuplas.
3. As tuplas passam pela mesma função de particionamento que as divide em duas partições.
4. Cada partição é enviada para um nó de *reduce* que conta as palavras.
5. O resultado dos nós de *reduce* são agrupados e enviados como solução.

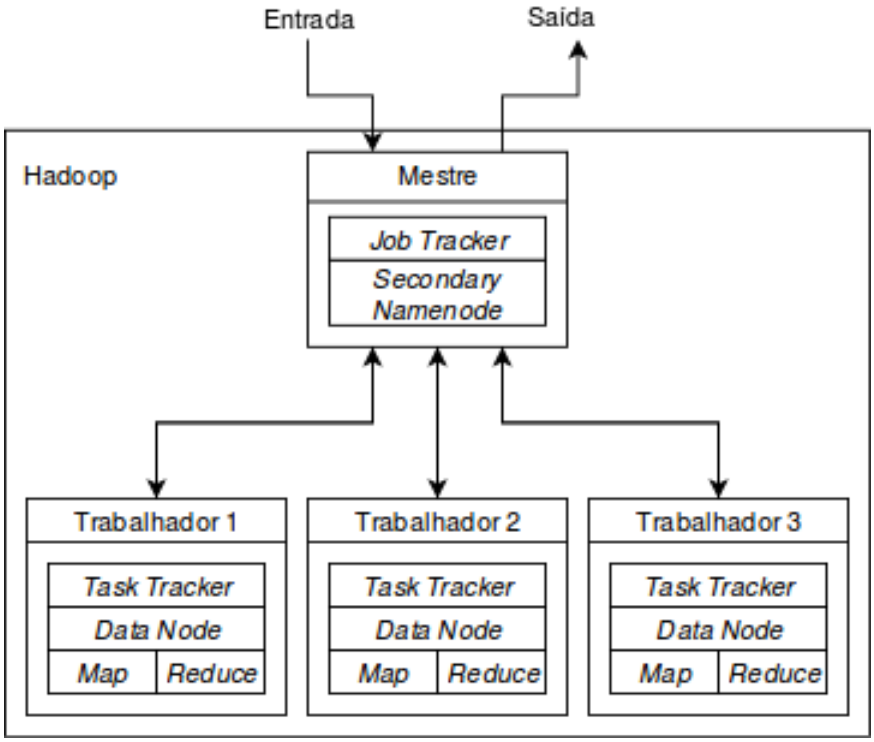
Figura 2 – Representação do *MapReduce*




O modelo do *MapReduce* foi implementado em diferentes linguagens de programação (ISARD *et al.*, 2007; HE *et al.*, 2008; RANGER *et al.*, 2007). Dentre as implementações existentes, uma das primeiras foi o *Hadoop* (SHVACHKO *et al.*, 2010), desenvolvida pela *Apache*. A *Hadoop* utiliza uma estrutura de *cluster* onde um nó chamado mestre controla as ações de todo o *cluster* e coordena os outros nós que são comumente chamados de trabalhadores ou escravos. Para isso, no nó mestre, é implementado o *Job Tracker*, que recebe todas as informações dos trabalhos que o *cluster* está rodando. É através do *Job Tracker* que o mestre recebe o pedido de aplicação, colhe as informações de cada escravo, identifica nós com problemas e exibe o progresso das tarefas de *map* e *reduce*. Para se comunicar com o mestre, os escravos possuem um *Task Trackers* que recebe as orientações do *Job Tracker* sobre qual tarefa dentre *map* e *reduce* aquele nó deverá executar, bem como comunicam ao *Job Tracker* sobre o progresso de cada nó. Comandado pelos *Task Tracker*, cada nó possui um sistema de armazenamento chamado *Data Node* que é usado para armazenar os dados intermediários das tarefas até que requisitado pelo

*Job Tracker* e pelos *Task Trackers* para as próximas etapas. A arquitetura da *Hadoop* é mostrada na Figura 3.

Figura 3 – Arquitetura da implementação da *Hadoop*




## 2.2 SKEW DE PARTICIONAMENTO

Usualmente, quando se fala em distribuição de dados, assume-se que o ambiente e essa distribuição são homogêneos. Esse pensamento é causado pela ideia de que, com a divisão igualitária da carga e dos recursos, todas as máquinas terminariam ao mesmo tempo, não ocorrendo casos de máquinas ociosas ou sobrecarregadas. Entretanto, o particionamento dos dados intermediários citados na Seção 2.1 dificilmente é efetuado de maneira igualitária.

Durante cada chamada da função de *map*, logo após a criação dos dados intermediários, o sistema chama a função de particionamento que agrupa as chaves de acordo com a função $Hash(HashCode\ mod\ number\ of\ reducers)$, usada como função padrão pela implementação do *Hadoop*. Essa função padrão calcula o valor *hash* da chave de cada tupla e o divide pelo número de tarefas de *reduce* que o usuário configurou, utilizando o resto da divisão como o indicador de para qual partição aquela tupla deve ir, garantindo que todas as tuplas com os mesmos valores de chave sejam enviadas para a mesma partição. O tamanho de uma partição acaba sendo definido pelo número de tuplas em cada uma, pois é o processamento dessas tuplas

que configura o processamento de cada partição. Evidentemente, quando a quantidade de tuplas $< CHAVE, VALOR >$ não está distribuída de maneira uniforme, a alocação das partições se torna desbalanceada. Por exemplo, no caso das aplicações de contagem de palavras, algumas palavras vão aparecer muito mais do que outras. Como palavras iguais se tornam chaves iguais, partições contendo as palavras mais populares usualmente vão possuir uma carga maior. Esse tipo de situação caracteriza o que é chamado de *partitioning skew*, um problema em que alguns nós de redução recebem grandes cargas, aumentando o tempo de resposta das tarefas a eles alocadas e, consequentemente, o tempo de resposta total da aplicação, uma vez que o resultado final depende que todas as tarefas finalizem. Muitos autores estudam o *skew* de particionamento em aplicações *MapReduce* e suas causas (KWON *et al.*, 2011; OKCAN; RIEDEWALD, 2011; IBRAHIM *et al.*, 2010; ATTA *et al.*, 2011).

Nas Subseções a seguir trataremos de três causas para o *partitioning skew*. A seção 2.2.1 tratará de chaves com frequência enviesada, isto é, quando muitas chaves diferentes são enviadas para a mesma partição. A seção 2.2.2 tratará de chaves com tamanho enviesado, isto é, quando algumas chaves possuem um volume muito maior que outras tornando-as um problema para particionar. Por fim, A seção 2.2.3 tratará de tempo de execução enviesado, isto é, quando trechos do problema necessitam de mais poder computacional e de mais tempo.

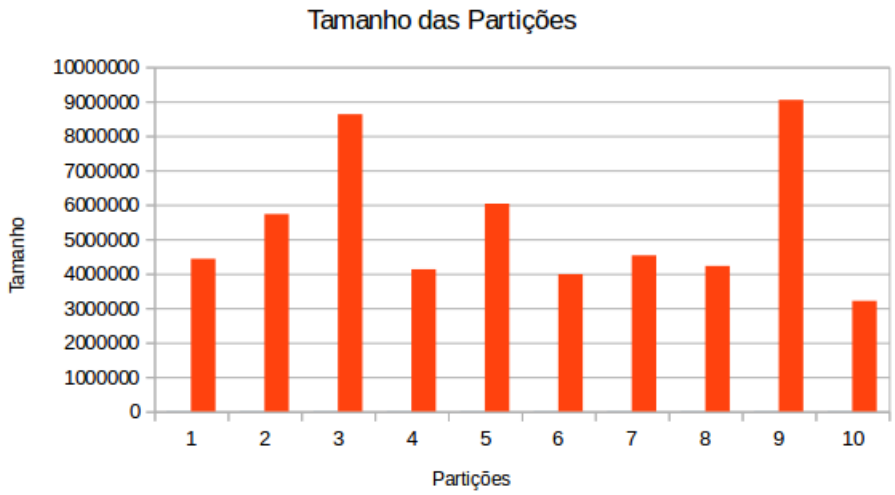
### *2.2.1 Chaves com Frequência Enviesadas (Skewed Key Frequencies)*

No primeiro cenário a ser considerado, temos que um grande volume de chaves pode ser enviado para uma mesma partição, fazendo com que essa partição precise de mais esforço computacional para que todos os dados sejam analisados. Isso ocorre porque a função padrão descrita no início dessa seção não possui um controle do modo como as chaves são distribuídas. Sem utilizar qualquer conhecimento do conteúdo que já foi ou ainda falta ser distribuído, a máquina é incapaz de dividir as chaves para as partições de modo a garantir que todas recebam uma quantidade semelhante de chaves.

A Figura 4 foi criada através do resultado de um experimento utilizando uma função de contagem de palavras conhecida como *wordcount* particionada pela função padrão, descrita no início dessa seção, usando como entrada um conjunto de *reviews* em livros da *Amazon* (MCAULEY *et al.*, 2015b; MCAULEY *et al.*, 2015a). Uma vez que essa aplicação não possui um fator aleatório, foi realizada uma única execução em um *cluster* de máquinas virtuais usando *Hadoop* e dez nós de *reduce* resultado em um gráfico apresentando as dez partições criadas

durante a fase de particionamento e seus respectivos tamanhos em número de chaves contidas nas mesmas. A partição enviada para o nó 9 possui um tamanho total de 9043509 chaves combinadas, contando as diferentes e as repetidas, enquanto a partição enviada para o nó 10 possui um tamanho combinado de 3206669 chaves. Assumindo que o tempo para processar cada chave siga uma progressão linear e todas as partições iniciem ao mesmo tempo, o nó responsável pela partição 10 ficou ociosa por 64% do tempo de processamento da fase de *reduce* uma vez que o tempo total acabou sendo baseado no nó 9, por possuir mais chaves que as demais.

Figura 4 – Exemplos de frequência de chaves enviesadas



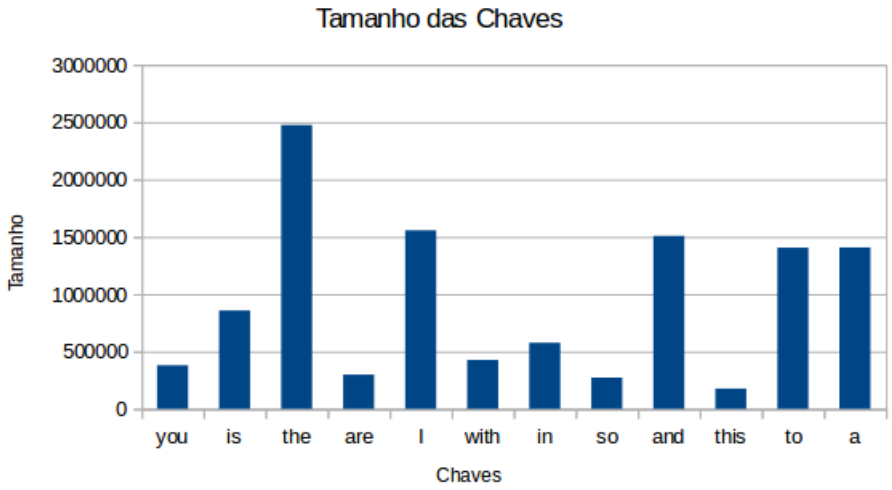

### *2.2.2 Chaves com Tamanho Enviesado (Skewed Key Sizes)*

A exemplo da aplicação de contagem de palavras, algumas palavras aparecem muito mais que as outras, e o número de aparições de palavras tais como "*the*"e "*a*"cresce dentro do texto de maneira muito mais rápida do que o restante das palavras o que pode tornar as partições possuidoras dessas chaves desbalanceadas com relação à partições que não tenham palavras tão repetidas. O tamanho das chaves, representado pelo número de aparições da mesma em um dado problema, funciona como mais um empecilho para o particionamento igualitário dos dados. Semelhante ao caso anterior em que a função padrão não podia garantir equilibrar o número de chaves entre as partições, a função padrão de particionamento de chaves também não possui um controle quanto ao tamanho de cada chave, podendo agrupar em uma mesma partição as chaves maiores e deixar as chaves menores divididas de forma desigual entre as outras partições.

A Figura 5 mostra as maiores chaves de cada partição do experimento anterior ordenadas pelo número da partição ("*you*", "*is*", "*the*", "*are*", "*I*", "*with*", "*in*", "*so*", "*and*",

"*this*"). Além delas, acrescentamos também outras chaves da partição 9 ("*to*", "*a*") cujo número de aparições lhes garantem um lugar dentre as maiores chaves encontradas no experimento. É possível notar que a principal causa da grande concentração de chaves na partição 9 é a presença de chaves muito grandes ("*and*"com 1507938 repetições, "*to*"com 1405633 repetições, "*a*"com 1406978 repetições) enquanto a partição 10 possui a menos frequente dentre as chaves mais frequentes de cada partição ("*this*"com 175867 repetições). Caso a segunda maior chave da partição 9 tivesse sido enviada para a partição 10 a diferença entre as duas diminuiria em 2813956 chaves e a partição 10 se aproximaria do tamanho médio das partições.

Figura 5 – Exemplo de chaves com o tamanho enviesado



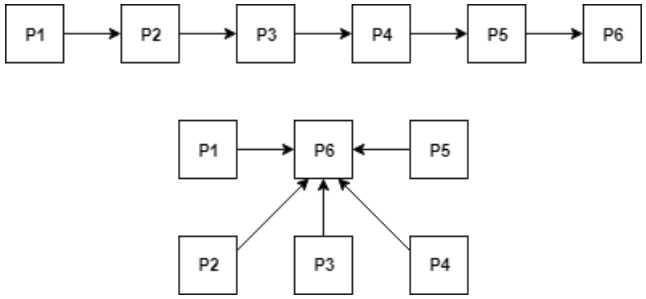

### *2.2.3 Tempo de Execução Enviesado (Skewed Execution Time)*

Em certos casos, a complexidade da função de redução escrita pelo usuário pode ser um fator determinante para o tempo de execução das máquinas, mesmo que estas estejam balanceadas. Pode ser necessário mais tempo para processar uma única chave grande do que múltiplas chaves pequenas, mesmo que o grupo de chaves pequenas possuam o mesmo tamanho que a chave grande.

A Figura 6 apresenta dois exemplos relacionados à classificação de páginas que remetem ao tempo de execução enviesado. Em ambos os exemplos a página 6 é alcançada por cinco páginas. Entretanto, no exemplo 1, o percurso efetuado para a página 1 alcançar a página 6 é muito maior do que no exemplo 2, pois a ligação entre essas duas páginas acaba sendo efetuado de maneira indireta, precisando processar todas as outras páginas. De maneira análoga, o custo de processamento da página 5 no exemplo 1 acaba sendo maior do que no exemplo 2, uma vez

que nenhuma página no exemplo 2 precisa processar a página 5 para alcançar outra.

Figura 6 – Exemplo de tempo de execução enviesado




## 2.3 META-HEURÍSTICAS

O problema do balanceamento de carga é um problema de otimização que, dependendo da carga a ser distribuída, pode necessitar de uma quantidade de recursos e tempo demasiadamente grandes para que a solução ótima seja encontrada. Entretanto, não necessariamente necessitamos de tal solução. Uma vez que para procurar solucionar o problema do *partitioning skew* pode ser necessário modificar o modo como as chaves são distribuídas entre as partições, torna-se fundamental a utilização de técnicas cuja tomada de decisão ajude a encontrar uma solução boa de forma mais rápida.

Algoritmos determinísticos seguem passos específicos recebendo uma entrada fixa em um ambiente controlado e retornando a solução considerando as informações que receberam como entrada. Mas em determinadas ocasiões o número demasiadamente grande de soluções possíveis pode atrapalhar o julgamento de algoritmos determinísticos fazendo com que a demora para retornar a solução ótima seja mais prejudicial do que obter um resultado bom o suficiente em um tempo hábil. Nesse tipo de situação são utilizadas heurísticas, métodos que trabalham com tomadas de decisões, tornando-as mais capazes de obter uma solução boa dada uma entrada aleatória de um problema. Entretanto, uma boa entrada não significa a melhor. Tomando como exemplo a heurística *take-the-best*, apenas a melhor dentre as opções apresentadas é escolhida. Caso um ótimo local seja essa melhor opção, a heurística pararia naquela posição por falta de opções melhores. As meta-heurísticas são procedimentos de alto nível que utilizam um algoritmo para encontrar uma solução satisfatória para um problema genérico, são não determinísticas e aproximativas na maioria dos casos, e analisam o conjunto de soluções de uma maneira mais eficiente tomando decisões que, inicialmente, poderiam parecer escolhas ruins, mas que auxiliam em encontrar soluções melhores a longo prazo.

Um fator importante para o desempenho das meta-heurísticas é a quantidade de passos a serem efetuados, isto é, o número de interações que a meta-heurística terá com a solução que ela está montando a partir dos dados de entrada. Essa variável, podendo receber outros nomes em casos específicos tais como *temperatura* para a meta-heurística do *simulated annealing* explicada a seguir, é um dos fatores responsáveis por quão próxima a resposta da meta-heurística será da solução ótima. Entretanto, o número de passos também é um dos principais fatores relacionados ao tempo de execução das meta-heurísticas, sendo necessários testes e estudos de caso para encontrar uma boa proporção entre o tempo decorrido pela meta-heurística e a qualidade da resposta.

Utilizaremos essa seção para introduzir as três meta-heurísticas utilizadas nesse trabalho. A seção 2.3.1 abordará a *Simulated Annealing*. A seção 2.3.2 abordará as meta-heurísticas *Local Beam Search* e *Stochastic Beam Search*. Essas três meta-heurísticas foram escolhidas pelo modo como resolvem problemas de otimização, a tomada de decisão que executam, o modo como são implementadas e suas semelhanças.

### *2.3.1 Simulated Annealing*

Em metalurgia, *Annealing* é o processo de endurecer metais ou vidro aquecendo e resfriando eles gradualmente, permitindo que o material alcance um estado cristalino de baixa energia (RUSSELL *et al.*, 2003). A *Simulated Annealing* (SA) é uma técnica semelhante que modifica a solução de maneira mais abrupta quando no início (quente) e diminui essas modificações quando próxima do fim da execução (frio). De certa forma, imaginemos que o conjunto de soluções seja uma superfície irregular, a configuração inicial seja o ponto onde soltamos uma esfera nessa superfície, e a solução final seja o ponto onde essa esfera parou, como mostrado na Figura 7 (DAS, 2014). Quando a meta-heurística altera a configuração inicial em busca da solução ideal, sempre escolhendo o melhor resultado quando possível, é como se a esfera rolasse pela superfície em busca de uma posição de estabilidade. Eventualmente, a meta-heurística pode encontrar uma solução ótima local, o equivalente a esfera ficar alojada em um buraco na superfície, estabilizando naquela posição. Para que a esfera continue rolando pela superfície, é necessária uma ação adicional forte o suficiente para retirar aquela esfera do seu repouso. De maneira semelhante, é necessária uma ação para que a meta-heurística saia daquela solução local forçando uma escolha indesejada para que uma solução melhor do que a local encontrada anteriormente surja. Essa ação ocorre quando o resultado encontrado pela

meta-heurística é pior do que o resultado da posição em que ela já está, e a força dessa ação é dependente da diferença entre os dois resultados e do número de passos restantes, quanto mais passos restarem para serem executados, ou menor a diferença entre as soluções, maiores são as chances de escolher um caminho com um resultado pior do que o ponto atual. Com o tempo, o caminho da solução se solidifica para evitar que escolhas ruins sejam feitas próximas ao final da execução.

Figura 7 – Exemplificação do *Simulated Annealing* com a analogia da esfera


### *2.3.2 Beam Search*

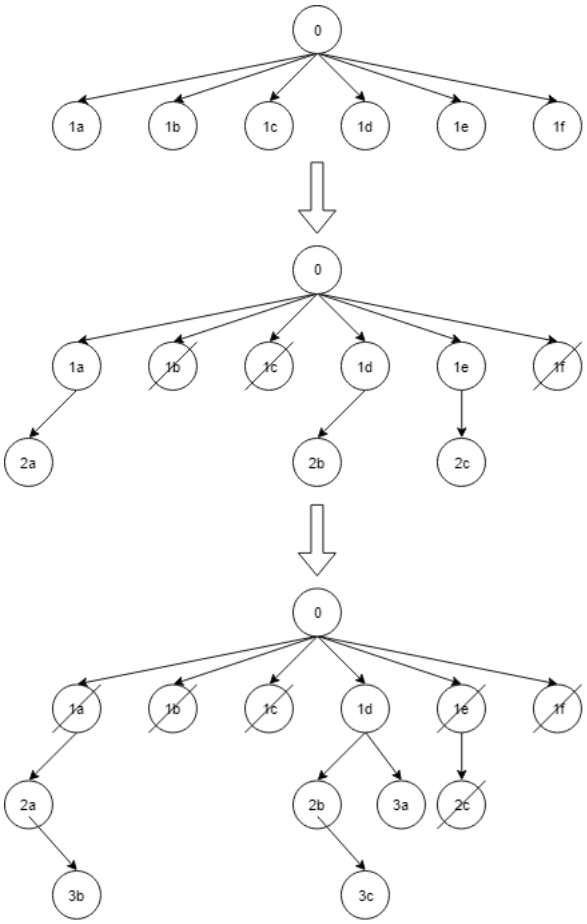
Enquanto no *Simulated Annealing* apenas um caminho para a solução é mantido, o *Beam Search* é um modelo de meta-heurística que executa uma busca em árvore selecionando um grupo de nós em cada passo, exigindo mais poder computacional, e cobrindo um maior número de caminhos para uma ou mais soluções. Esses grupos expandem a busca ao mesmo tempo em que a direcionam até que uma quantidade determinada de passos seja executada culminando com a escolha da melhor solução dentre as encontradas.

Ambas as meta-heurísticas apresentadas nessa seção seguem um caminho bastante parecido. Primariamente é dada uma configuração como entrada que será a raiz da árvore e um valor $\beta$ usado como número de caminhos a serem criados e descartados em cada iteração, na primeira iteração são criados $2\beta$ filhos da raiz da árvore de forma aleatória. Desses filhos, metade é descartada de acordo com a regra da meta-heurística sobrando $\beta$ filhos. Cada filho

restante da etapa anterior gera um filho que, somado com os $\beta$ da etapa anterior, totalizam $2\beta$ candidatos cuja metade será novamente podada. Essa etapa é repetida até que o número total de passos selecionados seja efetuado. Ao final, a melhor solução dentre as $2\beta$ geradas no último passo é retornada como a solução encontrada pela meta-heurística.

Como exemplo, apresentamos a Figura 8, onde executamos três etapas simplificadas em um modelo de *Beam Search* com $\beta = 3$. Na primeira etapa, o nó raiz cria seis filhos. Na segunda etapa, três nós são cortados e três novos nós são gerados, cada um partindo de um dos nós restantes. Na ultima etapa mostrada, mais três nós são cortados e três novos nós são criados partindo dos nós restantes. Caso o exemplo continuasse, ele iria seguir podando e criando novos nós até que acabasse o número de passos designado e eles escolhesse o melhor resultado dentre os nós restantes.

Figura 8 – Exemplificação da execução de um *Beam Search* em três etapas

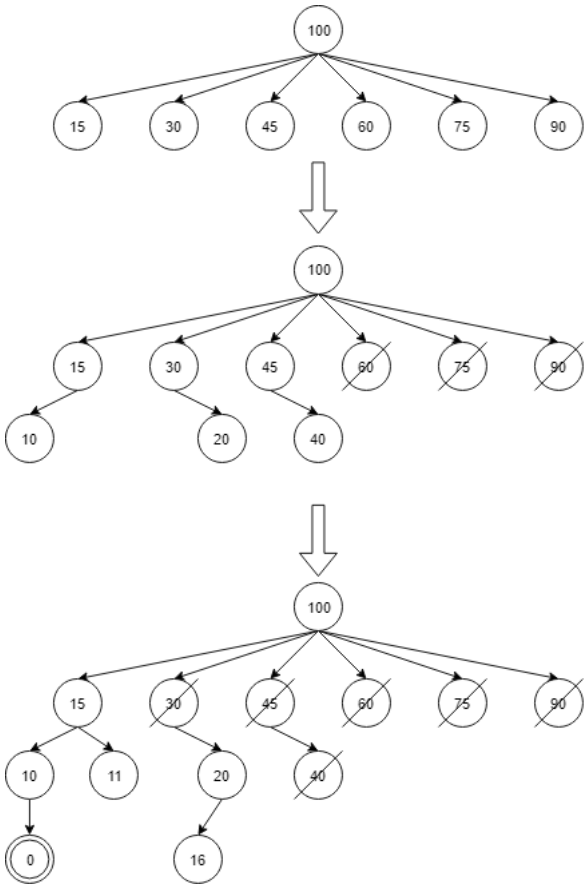


#### *2.3.2.1 Local Beam Search*

Sendo a versão mais simples, o *Local Beam Search* seleciona em cada etapa os $\beta$ filhos com o melhor resultado para compor a etapa seguinte. Entretanto, isso faz com que a *Local Beam Search* se prenda a ótimos locais uma vez que, ao chegar em um, ele não possua nenhum artifício que o faça saltar da posição em que ele está preso para uma outra posição que permita ele continuar se aprimorando.

A Figura 9 apresenta a execução de um *Local Beam Search* de forma bem otimista. A cada passo, ele remove os piores resultados até chegar na solução ótima.

Figura 9 – Exemplificação da execução de um *Local Beam Search* em três etapas




#### *2.3.2.2 Stochastic Beam Search (sbs)*

Como uma forma de resolver o problema do *Local Beam Search* estagnar em um ótimo local, o *Stochastic Beam Search* opta por escolher $\beta$ filhos de forma semi-aleatória onde filhos com resultados melhores possuem uma chance maior de serem escolhidos em detrimento de filhos com resultados piores. Com o intuito semelhante ao do *Simulated Annealing*, a escolha

de um caminho desbalanceado inicialmente pode parecer uma ideia ruim, mas sua verdadeira razão é buscar se distanciar dos ótimos locais e saltar para uma solução melhor.

A Figura 8, apresentada anteriormente, representa bem o comportamento da *Stochastic Beam Search*. Com um comportamento diferente da sua antecessora, a *Stochastic Beam Search* age de maneira mais aleatória gerando uma arvore que não se focaria em um único lado.

## 2.4 CONCLUSÃO

Nesse capítulo foi abordado o modelo de computação *Mapreduce*, no qual podemos identificar o problema do *partitioning skew* a ser resolvido. Esse problema pode ter mais de uma origem e seu tratamento pode diferir ou não de uma mesma solução. Por último, apresentamos o conceito de meta-heurísticas e explicamos as três meta-heurísticas que foram utilizadas na realização da nossa pesquisa.

# 3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados alguns trabalhos relacionados à diminuição do *skew* em ambientes *MapReduce*. Conforme visto no Capítulo 2, existem várias causas que podem levar um sistema a gerar *skew*. Portanto, mais de uma maneira de resolver tal problema. Os trabalhos mostrados aqui são algumas técnicas que outros autores propuseram para solucionar este problema.

## 3.1 ALOCAÇÃO DE RECURSOS

O *MapReduce* assume que os nós são homogêneos em relação aos seus recursos computacionais, o que faz com que nós que recebem mais dados demorem mais tempo para processar a carga alocada. Uma das soluções possíveis seria remanejar mais recursos para os nós que receberem mais dados, de modo a equilibrar o tempo de processamento por meios de uma razão entre a carga alocada e os recursos disponíveis. Os trabalhos descritos nesta seção possuem como objetivo resolver o *skew* por meio do remanejamento dos recursos computacionais tais como vCPUs e memória. Para isso, o sistema identifica os nós sobrecarregados pela utilização de monitores e preditores e realocando recursos para as mesmas.

### *3.1.1 DREAMS: Dynamic Resource Allocation for MapReduce with Data Skew*

DREAMS (LIU *et al.*, 2015) é um *framework* que trabalha em um ambiente de máquinas virtuais com autonomia para redistribuir recursos de maneira dinâmica. Ele inicia seu comportamento como um sistema de *MapReduce* padrão e modifica a distribuição dos recursos ao longo do seu funcionamento. Durante a execução de um problema, tal como um classificador de páginas ou um contador de palavras, o *DREAMS* consegue detectar o *skew* dinamicamente e alocar mais recursos aos *reducers* com as maiores partições para que eles terminem mais rápido. Para isso, ele desenvolve um modelo de predição *online* que estima o tamanho das partições durante a execução através de informações colhidas por monitores alocados em cada nó responsável por executar a tarefa de *map*. Através das predições efetuadas, ele libera mais recursos, memória, disco, ou processamento, para os nós de modo a diminuir as sobrecargas detectadas.

Após a execução da tarefa, ele estabelece um modelo de performance que relaciona o tempo de execução com a alocação de recursos baseado no que foi observado pelo modelo

de predição e pela execução dos nós de *reduce* que tiveram seus recursos alterados. Através do modelo de performance, ele constrói um perfil para a tarefa executada, de modo que, caso a mesma tarefa seja executada novamente, ele aloca e redistribui recursos de forma preemptiva para cada nó, visando evitar os gargalos encontrados pelos registros anteriores do mesmo problema. Tal ação é justificada pelo fato de um mesmo conjunto de dados ou uma mesma tarefa nunca serem executadas uma única vez. Cada execução apresenta uma nova análise daquela situação que é utilizada para aprimorar o perfil daquela dada tarefa. Durante a execução, usando esse modelo, o sistema agenda decisões que alocam a quantidade certa de recursos para a tarefa de redução com base em perfis de tarefas executadas anteriormente, de modo a equalizar o tempo de execução. Ao final da execução, os perfis são atualizados com as novas informações colhidas onde são utilizados em execuções futuras.

### *3.1.2 OPTIMA: On-Line Partitioning skew Mitigation for MapReduce with Resource Adjustment*

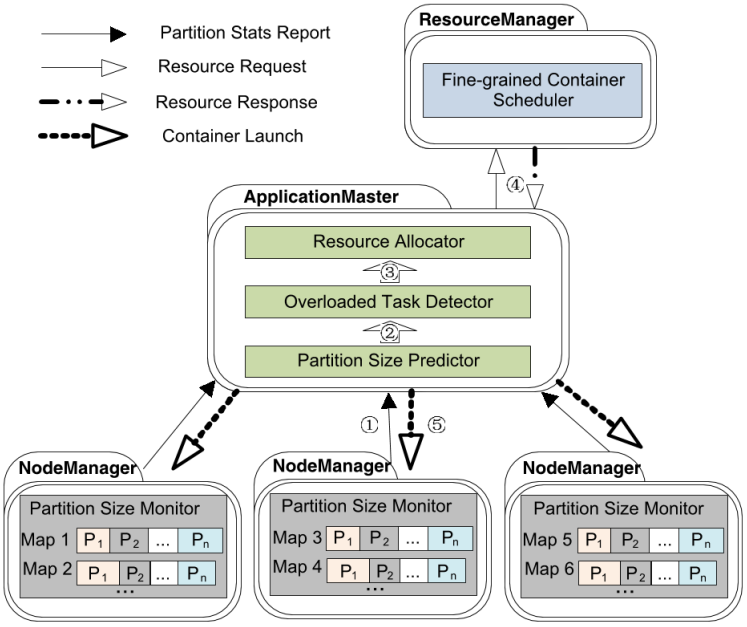
*OPTIMA* (LIU QI ZHANG, 2016), assim como o *DREAMS*, é um *framework* de alocação dinâmica de recursos em ambiente virtual baseado na detecção de tarefas de *reduce* sobrecarregadas por *skew* de particionamento e sucessor do *DREAMS*. Semelhante ao seu antecessor, o *framework* manda as informações sobre a alocação dos dados intermediários para o nó mestre, que as usa para predizer o tamanho das tarefas e identificar possíveis partições sobrecarregadas. Com essa informação, o *OPTIMA* aloca recursos para os nós sobrecarregadas até o máximo de carga permitida pela máquina. Diferente do seu antecessor, não são mais criados perfis para cada tarefa, uma vez que tal ação exige uma quantidade suficiente de recursos e o problema do *partitioning skew* não era resolvido na primeira execução.

São implementados cinco componentes principais como apresentados na Figura 10:1. *Partition Size Monitor*. 2. *Partition Size Predictor*. 3. *Overloaded Task Detector*. 4. *Resource Allocator*. 5. *Fine-grained Container Scheduler*. Durante a execução das tarefas de mapeamento, o *Partition Size Monitor* modifica a função de *report*, responsável por enviar o progresso de cada tarefa do *Task Manager* para o *Job Manager*, do *Hadoop*, para enviar o tamanho dos dados intermediários produzidos para o nó mestre juntamente com o progresso da execução. Com essa informação recolhida das tarefas de mapeamento, o *Partition Size Predictor* usa um modelo de regressão linear para predizer o tamanho das partições de maneira *online*. A partir disso, o *Overloaded Task Detector* é capaz de identificar as tarefas de redução com maior carga e avisar

ao *Resource Allocator* que esses nós precisam de mais recursos. Por último, o *Fine-grained Container Scheduler* aumenta a quantidade de memória e CPU usada pelos nós que precisam de alto desempenho.

Como vantagem, o *OPTIMA* redistribui os recursos melhorando o desempenho dos nós e evitando que as máquinas fiquem ociosas. Entretanto, cada máquina possui um valor máximo de recursos que ela pode receber, delimitada pela capacidade da própria máquina. Após isso, ela pode sobrecarregar uma vez que ela está com uma porção muito grande da tarefa e não pode mais receber recursos. Além disso, o *OPTIMA* não trabalha diretamente com o particionamento dos dados fazendo com que certos nós precisem de bem mais recursos do que outros.

Figura 10 – Arquitetura do *OPTIMA* e seus componentes




## 3.2 DIVISÃO DE PARTIÇÕES

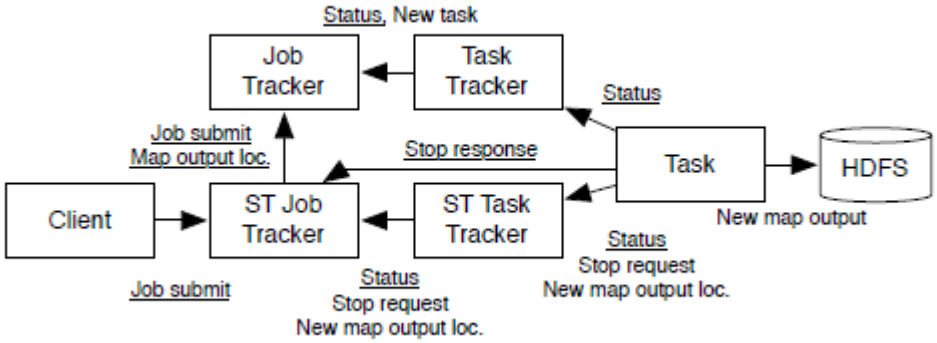
Em um ambiente virtual, quando um nó encerra o processamento da sua carga designada, ele se mantém ocioso até o fim da execução de todos os outros nós trabalhadores e seus recursos computacionais, que poderiam estar a pleno uso, são desperdiçados. O trabalho a seguir apresenta uma solução para aproveitar essas máquinas e diminuir o *skew* causado pela sua ociosidade.

### *3.2.1 SkewTune: Mitigating skew in MapReduce Applications*

O Skewtune (KWON Y., 2012) é uma API compatível com o *Hadoop* que resolve o *skew* reparticionando tarefas em execução toda vez que uma máquina conclui suas atividade e se torna ociosa. Para isso, ele acrescenta ao *Hadoop* dois módulos novos conforme apresentados na Figura 11. O primeiro, *SkewTune Task Tracker*, faz cada tarefa estimar seu tempo de conclusão com base no quanto ela já processou e no tanto de carga que ela acredita ter recebido, e o segundo, *SkewTune Job Tracker*, faz com que as tarefas guardem pequenas informações básicas tais como número de *bytes* processados. Um segundo *job tracker* e *task tracker* análogos aos existentes no *Hadoop* são criados para que, cada vez que um nó concluir suas tarefas, ele sinalize que se tornou disponível para o mestre. Com essa informação, o mestre verifica através de uma troca de mensagens quais nós possuem a previsão para o maior tempo de execução para as tarefas alocadas, e está disponível para parar sua execução para dividir sua carga com as outras máquinas, através do *SkewTune Task Tracker*. Essa máquina encerra a execução da sua fase de *reduce* como se só tivesse recebido a carga que ela já processou, o restante das chaves, presentes no que sobrou da partição, são dividas e migradas para outros nós, sempre respeitando a regra de chaves iguais irem para a mesma localização. Ao final, os resultados daquela partição dividida são ressincronizados e reenviados para o nó original.

A divisão provocada pelo *Skewtune* só ocorre caso o tempo necessário para efetuá-la somado ao novo tempo estimado para a conclusão das tarefas divididas seja menor que o tempo que a tarefa original demoraria para concluir. Caso contrário, a divisão das tarefas não é efetuada. O processo de procurar uma máquina para dividir sua carga com as máquinas ociosas é repetido até que todas as tarefas sejam concluídas ou nenhuma tarefa restante esteja apta a ser dividida. Ao final da execução, os dados das tarefas que foram divididas são reagrupados na máquina que originalmente os havia recebido. Esse processo é necessário para manter a consistência do *Hadoop*, de modo a garantir que todas as instâncias de uma chave estejam na mesma máquina. Apesar de funcionar bem para tarefas complexas, a sobrecarga causada pela análise e divisão de tarefas pequenas gera um custo maior do que simplesmente executar todas as tarefas normalmente, o que acaba por impactar negativamente no tempo de execução, fazendo com que o *skewtune* seja desaconselhável para análises com tarefas pequenas.

Figura 11 – Arquitetura do *skewtune* e seus componentes. As setas vão do remetente para o destinatário, as requisições estão sublinhadas.




## 3.3 MUDANÇA DE PARTICIONAMENTO

Como vimos no capítulo anterior, dentre as causas do *partitioning skew* temos tanto o tamanho das chaves quanto o modo como são distribuídas. O trabalho a seguir trata desses problemas através da mudança do modo como as partições são criadas.

### *3.3.1 Online Load Balancing for MapReduce with Skewed Data Input*

*Online Load Balancing for MapReduce with Skewed Data Input* (LE JIANG-CHUAN LIU, 2014) apresenta um algoritmo guloso *online* que age no particionador distribuindo as chaves de acordo com o tamanho das partições já existentes, dando prioridade às partições menores. Quando uma tupla é mandada para particionamento, primeiro é analisado se a chave dela já foi particionada. Caso ela seja uma chave nova para a função gulosa, o algoritmo procura a partição com o menor número de chaves e a manda para ela. Desse modo, o algoritmo guloso consegue diminuir a diferença entre as partições. Caso a tupla que foi mandada para particionamento possua uma chave já particionada, aquela chave não pode ser enviada para outra partição, forçando a tupla a ir para o mesmo local. O primeiro algoritmo não possui conhecimento prévio dos dados, o que o torna suscetível ao modo como as chaves estão distribuídas no conjunto de dados. Isso acarreta em possíveis escolhas ruins, tais como particionar duas chaves que pareciam ser pequenas, mas que apresentaram um crescimento abrupto próximo do final da fase de particionamento.

Para corrigir a falha do algoritmo guloso, ele cria um segundo algoritmo baseado em amostragem que, como o próprio nome indica, passa a colher uma parcela dos dados

intermediários, em tempo de execução, antes de executar qualquer particionamento. As chaves contidas nessa amostragem são ordenadas pelo número de aparições (tamanho) de maneira decrescente e distribuídas de modo semelhante ao utilizado no algoritmo anterior: se a chave já foi particionada ela é mandada para a mesma partição a qual sua aparição anterior foi enviada. Caso contrário, ela irá para a partição com a menor quantidade de carga. Finalizado o particionamento da amostragem, as chaves seguintes são distribuídas de maneira idêntica ao primeiro algoritmo. Essa amostragem usada como conhecimento prévio permite a identificação das chamadas chaves pesadas que vão causar mais impacto no tempo de execução da função de *reduce* do que as outras chaves. A razão disso é que, uma vez que todas as chaves tenham que ser analisadas e o tempo para analisar uma chave costuma ser proporcional ao tamanho dela, as chaves maiores consomem mais tempo de execução do que as demais chaves. Assim, separando as chaves pesadas em máquinas diferentes, o algoritmo garante um menor tempo de execução para a tarefa como um todo. Entretanto, uma vez que apenas uma amostragem é feita, e somente no início do algoritmo, um crescimento abrupto de outra chave poderia desequilibrar o particionamento gerado pela amostragem, provocando o *skew* que foi procurado evitar.

## 3.4 DISCUSSÃO

Os trabalhos apresentados neste capítulo possuem o mesmo intuito de eliminar o *skew* no *MapReduce*, mas suas execuções, e seus problemas, ocorrem de formas diferentes.

Tanto o *DREAMS* quanto o *OPTIMA* visam controlar a alocação de recursos em um ambiente virtual, reduzindo os recursos de nós ociosos e melhorando o desempenho de nós sobrecarregados. Entretanto, o *DREAMS* requer conhecimento prévio e criação de um conjunto de perfis para melhorar o desempenho de sua abordagem. Mesmo com a remoção dos perfis no *OPTIMA*, ele acaba sofrendo com problemas relacionados a alocação de recursos, mais especificamente quando um nó recebe uma carga demasiadamente grande e já alcançou o teto de recursos que o mesmo poderia receber.

O *SkewTune* não trabalha com o reparticionamento antes da fase de *reduce*. Ao invés disso, ele espera detectar máquinas disponíveis para rebalancear os dados. Como vantagem, ele poupa tempo caso o sistema já estivesse equilibrado. Mas como desvantagem, caso o sistema envie muitas requisições de tarefas para o nó mestre, o *SkewTune* acaba criando um *overhead* na tentativa de parar múltiplas tarefas pequenas tentando dividi-las.

O *Online Load Balance* (OLB) trabalha diretamente na função de particionamento.

Primeiro usando uma vertente gulosa, depois usando uma amostragem. Seu desempenho em equilibrar as partições é excelente quando a distribuição dos dados ocorre de maneira linear. Mas um surto de novas chaves no final do conjunto de dados pode acabar por desequilibrar sua distribuição.

Nossa abordagem adota o modelo de resolução semelhante ao utilizado pelo OLB (LE JIANGCHUAN LIU, 2014). Essa escolha se deve ao fato de fazer alterações diretamente no responsável pelo particionamento dos dados. Nossa contribuição com relação à abordagem citada está nas múltiplas amostragens, que usamos para melhorar o particionamento em situações não lineares, e na utilização de meta-heurísticas, que possuem uma boa capacidade de decisão para ajudar a melhor distribuir os dados.

| Estratégia | Conhecimento prévio | Online | Reparticiona | Uso de amostragens |
|---|---|---|---|---|
| DREAMS | Sim | Sim | Não | Criação de perfis |
| OPTIMA | Não | Sim | Não | Predição de carga |
| SkewTune | Não | Sim | Durante o *reduce* | Não utiliza amostragens |
| OLB | Não | Sim | Durante o particionamento | Amostragem única |
| MALiBU | Não | Sim | Durante o particionamento | Múltiplas amostragens |

Tabela 1 – Comparação entre os trabalhos relacionados

## 3.5 CONCLUSÃO

Este capítulo apresentou alguns trabalhos focados na resolução do problema do *skew* no *MapReduce*. Cada trabalho possui uma metodologia diferente para atacar o problema, não existindo uma solução perfeita ou absoluta. Além disso, as soluções poderiam ser aplicadas em conjunto, criando camadas de proteção para esse problema, mas precisando de cuidados adicionais com sobrecargas para não impactar negativamente na execução. A contribuição apresentada nesta dissertação segue para a criação de novos particionamentos.

# 4 UTILIZAÇÃO DE META-HEURÍSTICAS PARA REPARTICIONAMENTO DE DADOS EM APLICAÇÕES MAPREDUCE

Como vimos nos capítulos anteriores, o *MapReduce* é um modelo computacional de programação paralela que utiliza um *cluster* de máquinas para processar grandes quantidades de dados. Entretanto, a divisão desses dados entre as máquinas pode acarretar em um problema de balanceamento de carga chamado comumente de *data skew*. Dentre as possíveis causas para o *data skew*, destacamos o tamanho de cada partição (*skewed key frequencies*), que ocorre quando uma quantidade muito grande de chaves se concentra em uma mesma partição, o tamanho (número de aparições) de cada chave (*skewed key sizes*) e o tempo de processamento de cada chave (*skewed execution time*). Com exceção do *skew* causado pelo tempo de processamento de cada chave, que depende da complexidade da função de *reduce*, o problema de *data skew* em arquiteturas *MapReduce* pode ser atacado através do balanceamento da divisão dos dados entre as máquinas do *cluster*.

Assim sendo, criamos a seguinte definição para o nosso problema.

**Definição 1** (Melhor particionamento de um conjunto de dados)**.** Seja:

- $C$ um *cluster* de máquinas.
- $D$ um conjunto de dados
- $t =< k, v >$ uma tupla formada por uma chave $k$ e um valor $v$ a partir dos dados de $D$.
- $p_i = \{t_1 =< k_1, v_1 >, t_2 =< k_2, v_2 >, \ldots, t_m =< k_m, v_m >\}$ uma partição formada por um conjunto de tuplas $t$.
- $P = \{p_1, p_2, \ldots, p_n\}$ um particionamento de $D$, onde $n$ é o número de partições definidas pelo usuário.
- $T(p_i) = \{|p_i|\}$ a função que calcula o tamanho de uma dada partição $p_i$ como sendo sua cardinalidade.
- $F(P) = \{\max\{T(p_a) - T(p_b)\} \mid p_a, p_b \in P\}$ a função que calcula o desbalanceamento de um particionamento $P$, que é definido como a maior diferença entre os tamanhos das partições contidas em $P$.
- $R = \{P_1, P_2, \ldots P_w\}$ o conjunto de particionamentos possíveis de $D$ sobre $n$ partições.

Dizemos que $P_{opt}$ é o melhor particionamento de $D$ se e somente se $F(P_{opt}) \leq F(P_y)$, $\forall P_y \in R$, ou seja, $P_{opt}$ é o particionamento com o menor desbalanceamento dentre todos os particionamentos do conjunto $R$.

A função de particionamento implementada neste trabalho é apresentada na Seção 4.1. Na Seção 4.2, apresentamos o MALiBU, nosso componente que substitui o particionamento em uma arquitetura *MapReduce*, balanceando os dados entre as partições. Por fim, na Seção 4.3, apresentamos uma breve conclusão sobre o que foi apresentado neste capítulo.

## 4.1 CRIAÇÃO DA FUNÇÃO DE BALANCEAMENTO

Uma primeira tentativa para encontrar o particionamento ideal seria a estratégia gulosa proposta por (LE JIANGCHUAN LIU, 2014). Como apresentado na Seção 3.3.1, a estratégia gulosa utiliza a função $M(P) = \{p_a \mid T(p_a) \leq T(p_b), \forall p_b \in P\}$ para encontrar a menor partição em $P$ em um dado momento e aloca os dados para a mesma. Porém, a função de particionamento gulosa se atém a um único particionamento o qual ela vai incrementando a cada nova tupla alocada, dispensando a análise de outras configurações de particionamento presentes no conjunto $R$, tornando seu desempenho dependente da ordem em que as tuplas chegam na função de particionamento. Por esse motivo, descartamos o uso da função gulosa e optamos por colher amostras dos dados em tempo de execução antes de efetuar o particionamento dos mesmos.

Uma abordagem para encontrar o particionamento $P_{opt}$ é analisar o conjunto $R$. Entretanto, esse conjunto é dependente do número de partições, do número de chaves diferentes e do tamanho de cada chave, tornando qualquer tentativa de mapear todo o conjunto extremamente exaustiva, principalmente quando o conjunto de dados é muito grande. Uma estratégia para iniciar a busca no conjunto $R$ pelo particionamento ótimo é começar por um particionamento $P_0$ criado de maneira aleatória, ou semi-aleatória, e modificá-lo através de uma função $S(P)$. Essa função $S(P)$ funciona como um passo onde $S(P) = \{P' \mid P \neq P'\}$ onde a diferença entre $P$ e $P'$ é de uma chave $k_j$ que estava em uma partição $p \in P$ e foi realocada para $p' \in P'$, com todas as demais chaves com alocações idênticas em $P$ e $P'$; ou de um par de chaves $k_j$ e $k_f$ tal que $k_j \in p$, $k_f \in q$, com $p, q \in P$ e $k_f \in p$, $k_j \in q$ com $p, q \in P'$, com todas as demais chaves com alocações idênticas em $P$ e $P'$. Ou seja, a função $S(P)$ retorna um particionamento $P'$ alterando a alocação de uma chave de uma partição de $P$ para outra, ou trocando duas chaves de partição.

Uma vez definida a função de passo, precisamos definir como ela será usada. Uma primeira opção poderia ser a criação de uma função $H(P)$ que executaria a função $S(P)$ um número $Y$ de vezes de tal forma que, a cada execução, se $F(P') < F(P)$, isto é, se o desbalanceamento de $P'$ for menor que o desbalanceamento de $P$, então $P'$ substitui $P$ na próxima execução

de $S$. Esse modelo de operação funciona como a heurística *Hill Climb* que escolhe sempre o melhor resultado em detrimento das outras opções, portanto, existe o risco da função encontrar um particionamento que represente um ótimo local $P$, definido por $F(P) < F(P') \; \forall \; P' = S(P)$, de tal forma que esse $P$ não seja o ótimo global no espaço de busca $R$.

Para solucionar o problema encontrado em $H(P)$, optamos pelo uso da criação de uma função de balanceamento $B(P)$, que utiliza uma meta-heurística diferente do *Hill Climb* para balancear a carga utilizando sua capacidade de tomada de decisões como forma de obter um resultado melhor que $H(P)$. Ao todo, foram implementadas três meta-heurísticas (*Simulated Annealing*, *Local Beam Search* e *Stochastic Beam Search*), mas apenas uma é executada pela função de balanceamento. A motivação para a implementação de mais de uma, mesmo não as usando ao mesmo tempo, é para que possamos estudá-las e comparar seus desempenhos com relação ao nosso problema de balanceamento, para fins de pesquisa. Dessa forma, podemos estudar qual meta-heurística melhor se adapta a cada situação e quais os parâmetros ideais para cada uma. Todas as três funcionam de maneira bem semelhante, iniciando com um particionamento $P_0$ e executando a função $S(P)$ por um número $Y$ de vezes. O que as diferencia é o modo como elas agem após a execução do passo. Descrevemos como cada meta-heurística atua a seguir.

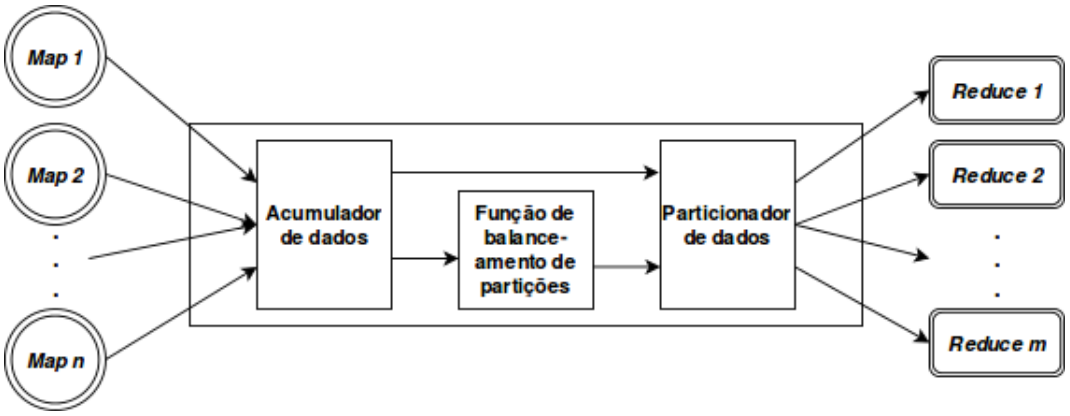
## 4.2 MALIBU: METAHEURISTICS APPROACH FOR ONLINE LOAD BALANCING IN MAPREDUCE WITH SKEWED DATA INPUT

Para que a função de balanceamento que criamos aja antes que a fase de *reduce* seja iniciada, a colocamos durante a fase de *map*, mais especificamente, durante as chamadas da função de particionamento e após a geração de cada tupla $t$. Entretanto, a função de particionamento não guarda o conteúdo das tuplas que ela processa. Ao invés disso, ela o transmite diretamente para as partições $p$ que serão utilizadas pelos nós de *reduce*, tornando o reparticionamento desse conteúdo e a criação do nosso $P_0$ mais complexos. Para contornar essa situação, modificamos a função de particionamento de modo que a função acumule uma determinada quantidade de dados antes de particioná-los para a próxima fase. A quantidade de dados acumulados é dada por um valor de $X\%$ do conjunto de dados definido como *threshold*, onde $0 < X < 100$. Graças a esse acumulador de dados, as duas necessidades inicias para criarmos o $P_0$ a ser utilizado pela nossa função de balanceamento foram satisfeitas. Essas duas necessidades são: obter o conteúdo dos dados a serem particionados; e impedir que os dados sejam particionados sem passar por

$B(P)$.

Uma vez que passamos a acumular parte dos dados a serem particionados, podemos finalmente montar o nosso $P_0$. Para isso, usamos a função padrão *Hash(HashCode mod number of reducers)*, que calcula o valor *hash* da chave e o divide pelo número de nós de *reduce*, como sugestão de partição para cada tupla. Após a execução da função de balanceamento $G(P)$, um particionador de dados usa o resultado $P_{opt}$, encontrado pela função, como base para o particionamento de todas as tuplas produzidas até então. Com isso, inicia-se um ciclo onde o MALiBU utiliza o acumulador de dados para armazenar uma quantidade de $X\%$ do total de dados, cria um modelo de particionamento usando a função de balanceamento e particiona os dados acumulados até aquele momento usando o modelo de particionamento encontrado, como mostrado na Figura 12.

Figura 12 – Representação dos componentes do *MALiBU*




Duas situações podem ocorrer com relação às tuplas que chegam no acumulador de dados. A primeira ocorre quando surge uma tupla com uma chave ainda não particionada. Nesse caso, a tupla será armazenada pelo acumulador de dados, passará pela função de balanceamento e será particionada em seguida, conforme mostrado no fluxo presente na Figura 12. As informações a respeito da alocação de uma chave são armazenadas de modo a sempre enviar as tuplas com mesma chave para uma mesma partição. A segunda situação ocorre quando a tupla possui uma chave já particionada em uma iteração anterior do MALiBU. Quando isso ocorre, o acumulador não armazena aquela tupla, que é imediatamente enviada para o particionador, onde as informações sobre sua alocação são atualizadas.

A principal motivação do incremento do modelo de particionamento com a adição de outros particionamentos feitos em grupos é garantir que os dados intermediários, que surgem

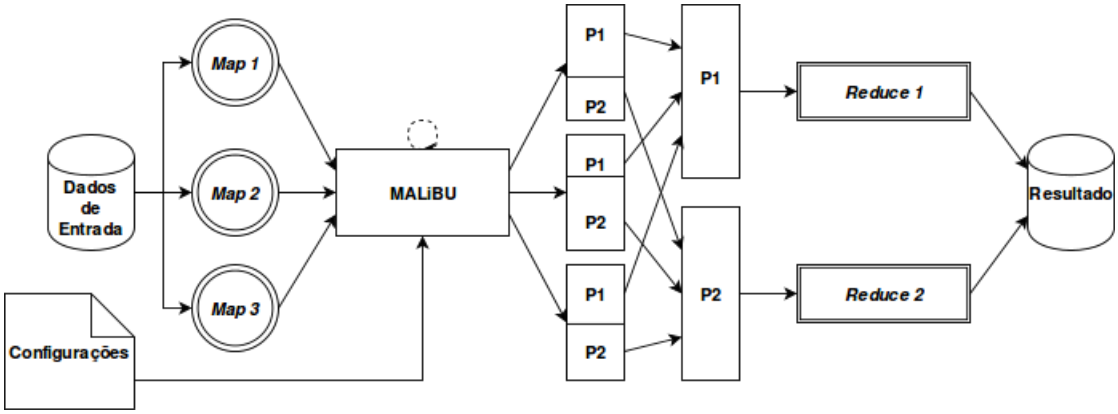
após a execução do primeiro particionamento, não desbalanceiem a distribuição por estarem enviesados. Uma primeira versão do MALiBU, descrita no artigo (PERICINI *et al.*, 2017), acumulava dados somente uma vez e criava apenas um modelo base, enviando as tuplas com chaves novas para a menor partição de uma maneira semelhante ao observado no algoritmo guloso. Como consequência, seu comportamento se tornava extremamente dependente da linearidade da distribuição e da exposição dos dados, isto é, ele necessita que a quantidade de chaves novas e o crescimento do volume das chaves não aumente de forma abrupta. Caso um número muito grande de chaves novas surgisse após o *threshold*, o MALiBU não teria como garantir o particionamento igualitário desses dados.

A introdução do MALiBU numa arquitetura de implementação do modelo *MapReduce* pode ser vista na Figura 13. Observe que o MALiBU atua justamente na definição do particionamento entre as fases de mapeamento e redução do modelo. Como descrito anteriormente, os dados vindos da fase de *map* são acumulados até alcançar o *threshold*, X. Após isso, são usados como entrada pela função de balanceamento e particionados de acordo com a solução obtida pela função. Além disso, conforme é mostrado na figura, o MALiBU necessita de um conjunto adicional de configurações contendo os seguintes parâmetros:

- o número, $Y$, de vezes que a função $S(P)$ será executada, ou seja, o número de passos que serão executados pela meta-heurística;
- o acúmulo de carga do *threshold* $X$;
- número de particionamentos efetuados a cada passo $\beta$;
- a estratégia de balanceamento.

Optamos por usar uma meta-heurística, como explicado na Seção 4.1, devido ao tamanho do conjunto $R$ de possíveis particionamentos, à capacidade de tomada de decisão das meta-heurísticas auxilia na análise de um conjunto muito grande de soluções, e ao controle que nós podemos exercer com relação aos parâmetros usados pela meta-heurística de modo que podemos modificá-las mais facilmente para cada situação. Mais uma vez, como dito anteriormente, apesar da função de balanceamento utilizar apenas uma meta-heurística, implementamos três com o intuito de comparar seus resultados para fins de estudo.

Figura 13 – Representação do *MALiBU*




Em cada passo executado por cada meta-heurística, são geradas uma ou mais soluções intermediárias que, por sua vez, são vizinhas das soluções do passo anterior. Esse conceito de vizinhança é usado para facilitar na criação das soluções seguintes, de modo a melhor poupar recursos, principalmente no uso da *Local Beam Search* e da *Stochastic Beam Search*, que requerem estruturas de dados mais robustas para guardar as informações de cada particionamento efetuado em cada passo. Explicamos, a seguir, o conceito de particionamentos vizinhos usado no experimento na Definição 2.

**Definição 2** (Particionamentos vizinhos)**.** Seja $K = \{k_1, k_2, \ldots, k_n\}$ um conjunto de chaves, $R = \{r_1, r_2, \ldots, r_m\}$ um conjunto de particionamentos e $P_{r_x} = \{p_1, p_2, \ldots, p_m\}$ o conjunto de partições pertencentes a um particionamento $r_x$ qualquer. Dizemos que $P_{r_1}$ é vizinha de $P_{r_2}$ se e somente se uma dessas condições é satisfeita:

- $k_i \in p_x \in P_{r_1}$, $k_j \in p_y \in P_{r_1}$, $k_j \in p_x \in P_{r_2}$ e $k_i \in p_y \in P_{r_2}$, com $1 \leq i < j \leq n$ e $1 \leq x < y \leq m$ para apenas um par de chaves $k_i$ e $k_j$. O particionamento é o mesmo para as chaves restantes em $P_{r_1}$ e $P_{r_2}$.
- $k_i \in p_x \in P_{r_1}$ e $k_i \in p_y \in P_{r_2}$, com $1 \leq i \leq n$ e $1 \leq x < y \leq m$ para apenas uma chave $k_i$. O particionamento é o mesmo para as chaves restantes em $P_{r_1}$ e $P_{r_2}$

As subseções a seguir serão utilizadas para explicar como cada meta-heurística atua para entregar uma solução com particionamento balanceado. Seção 4.2.1 tratará da *Simulated Annealing*. Seção 4.2.2 tratará da *Local Beam Search*. Seção 4.2.3 tratará da *Stochastic Beam Search*.

### *4.2.1 Simulated Annealing*

Na solução baseada no *Simulated Annealing*, o modelo de particionamento inicial criado pelo MALiBU é dado como entrada juntamente com o número de passos, que é definido como temperatura inicial $T$, e do número de partições $R$ que utilizamos no experimento. O modelo de particionamento inicial é chamado de *current* por ser o particionamento atual. Enquanto a temperatura não chegar a zero, ou seja, a meta-heurística executar o número $Y$ de passos definidos, criamos um particionamento vizinho (*neighbor*), respeitando a Definição 2, utilizando a função *generateNeighbor(partitioning)*. Essa função recebe o particionamento atual como entrada, escolhe uma chave de uma partição de maneira aleatória e, ou a troca de posição com outra chave de outra partição, ou a envia para outra partição. Uma vez que um particionamento vizinho foi criado, a função *unbalance(partitioning)* cria um grau de desbalanceamento, baseado na diferença entre todas as partições, para os dois particionamentos a serem comparados. Caso o grau de desbalanceamento do *neighbor* seja menor que o do *current*, o particionamento do *current* é substituído pelo *neighbor*. Caso contrário, é gerado um número aleatório entre 0 e 1 usando a função *random(0,1)* e comparado com o valor de $e^{\frac{b_1-b_2}{t}}$. Caso o valor gerado seja menor, ocorre a substituição do *current* pelo *neighbor*. Caso contrário, o *current* permanece. Após a execução dessas etapas, a temperatura é decrescida em 1 e um novo *neighbor* é gerado.

Ao final, o *current* é retornado como solução dada pela meta-heurística.

**Algoritmo:** Simulated Annealing

```
1  Fn(SimulatedAnnealing(D,T,R))
2  current ← D ;
3  t ← T;
4  enquanto t ≠ 0 faça
5      neighbor ← generateNeighbor(current);
6      b1 ← unbalance(current);
7      b2 ← unbalance(neighbor);
8      se b2 < b1 então
9          current ← neighbor;
10     senão se random(0,1) < e^((b1−b2)/t) então
11         current ← neighbor;
12     t− = 1
13 retorne current;
```

### *4.2.2 Local Beam Search*

Na solução baseada no *Local Beam Search*, a meta-heurística recebe como entrada o modelo de particionamento inicial $D$ provido pelo MALiBU, o número de passos, $I$, a serem executados, o número de partições $R$ usadas na execução, e o número de particionamentos $\beta$ a serem analisados por passo. O modelo $D$ é usado como particionamento inicial, *init*. Dele, são criados $2 * \beta$ particionamentos vizinhos, usando a mesma função *generateNeighbor(partitioning)* vista no Algoritmo da *Simulated Annealing*, e adicionados ao *array temp* que servirá como portador temporário de todos os particionamentos. Do *temp*, são escolhidos os $\beta$ melhores resultados usando a função *selectBest(int,array)* que recebe como entrada o número de resultados a serem escolhidos e o *array* a ser percorrido em busca das soluções, utiliza a função *unbalance(partitioning)* para gerar um grau de desbalanceamento para cada particionamento contido no *array* passado como entrada, e retorna um novo *array* contendo os particionamentos com o menor grau de desbalanceamento. O resultado da função *selectBest(int,array)* é considerado o conjunto de particionamentos atuais e chamado de *current*. Após essa primeira seleção, o número de passos começa a ser decrementado e, para cada passo, o *array temp* receberá o conteúdo do *curent* e gerará um particionamento vizinho para cada particionamento existente no *curent*. O

conteúdo existente no *temp* é podado e transformado em uma nova versão do *current*. Ao final da execução, o melhor resultado existente no ultimo *current* gerado é selecionado como a solução dada pela meta-heurística.

**Algoritmo:** selectBest

```
1  Fn(selectBest(num,current))
2  scoreList ← createArray();
3  para a ← 0 to current.size faça
4      scoreList.add(unbalance(current[a])
5  temp ← createArray();
6  para a ← 0 to num faça
7      lowest ← getLowestScore(scoreList);
8      temp ← current[lowest];
9      current.remove(lowest);
10     scoreList.remove(lowest);
11 retorne temp;
```

**Algoritmo:** Local Beam Search

```
1  Fn(LocalbeamSearch(β,D,T,R))
2  init ←D; temp ← createArray();
3  current ← createArray();
4  para q ← 1 to 2 ∗ β faça
5      temp.add(generateNeighbor(init));
6  current ← selectBest(β,temp);
7  t ← T;
8  enquanto t ≠ 0 faça
9      temp ← createArray();
10     temp ← current ;
11     para q ← 1 to β faça
12         temp.add(current[q]);
13         temp.add(generateNeighbor(current[q]));
14     current ← selectBest(β,temp);
15     t = t − 1;
16 retorne selectBest(1,current);
```

### *4.2.3 Stochastic Beam Search*

A solução baseada na *Stochastic Beam Search* é executada de maneira bem similar à solução apresentada pelo Algoritmo *Local Beam Search*. A única exceção sendo a mudança da função *selectBest(int,array)* para a função *selectSemiRandom(int,array)* que utiliza a função *unbalance(partitioning)* para gerar um grau de desbalanceamento para cada particionamento e, a partir desse grau, elabora uma chance inversamente proporcional ao grau calculado de cada particionamento ser escolhido, isto é, quanto menor for o grau de desbalanceamento, maior é a chance daquele particionamento ser escolhido. A chance de um bom particionamento ser escolhido aumenta conforme o número de particionamentos a serem escolhidos para o *array* de retorno da função diminui, o que favorece os particionamentos mais balanceados. Isso ocorre porque um dos fatores para o calculo da chance de cada particionamento ser escolhido é o número de particionamentos restante no *array*.

**Algoritmo:** selectSemiRandom

```
1  Fn(selectSemiRandom(num,current))
2  scoreList ← createArray();
3  para a ← 0 to current.size faça
4  |  scoreList.add(unbalance(current[a])
5  sum ← 0;
6  para a ← 0 to current.size faça
7  |  sum+ = scoreList[a];
8  chanceList ← createArray();
9  para a ← 0 to current.size faça
10 |  chanceList.add(sum − scoreList[a]);
11 temp ← createArray();
12 para a ← 0 to num faça
13 |  chanceTotal ← sum × (current.size − 1);
14 |  random ← generateNumber(chancetotal);
15 |  iterator ← 0;
16 |  enquanto chanceList[iterator] ≤ random faça
17 |  |  random− = chanceList[iterator];
18 |  |  iterator+ = 1;
19 |  temp.add(current[iterator]);
20 |  current.remove(iterator);
21 |  sum− = scoreList[iterator] scoreList.remove(iterator);
22 |  chanceList.remove(iterator);
23 retorne temp;
```

**Algoritmo:** Stochastic Beam Search

```
1  Fn(StochasticBeamSearch(β,D,T,R))
2  init ← partitioning of the keys of D using Hash(HashCode(D) mod |R|);
3  temp ← createArray();
4  current ← createArray();
5  para q ← 1 to 2 * β faça
6  |  temp.add(generateNeighbor(init));
7  current ← selectSemiRandom(β, temp);
8  t ← T;
9  enquanto t ≠ 0 faça
10 |  temp ← createArray();
11 |  temp ← current;
12 |  para q ← 1 to β faça
13 |  |  temp.add(current[q]);
14 |  |  temp.add(generateNeighbor(current[q]));
15 |  current ← selectSemiRandom(β, temp);
16 |  t = t − 1;
17 retorne selectBest(1, current)
```

## 4.3 CONCLUSÃO

Os três pseudocódigos apresentados demonstram as meta-heurísticas que utilizamos para reduzir o desbalanceamento das partições enviadas para a fase de redução que é causado de maneira não intencional pela fase de particionamento ao não distinguir as chaves, ou não se preocupar com o conteúdo das partições.

A integração ocorre de forma que o usuário pode escolher a meta-heuristica a ser usada pelo MALiBU, que substitui completamente o particionamento executado pelo *MapReduce*, o *threshold* que marca a quantidade de dados a serem acumulados, a quantidade de passos que as meta-heurísticas executam, e o número de particionamentos por passo analisados pela *Local Beam Search* e *Stochastic Beam Search*. Com a substituição da fase de particionamento pelo MALiBU, aumenta a complexidade para o usuário, mas dá a ele um melhor controle sobre os dados a serem analisados pela sua aplicação.

# 5 AVALIAÇÃO EXPERIMENTAL

Nosso principal objetivo com os experimentos realizados é apresentar os resultados alcançados com as meta-heurísticas apresentadas no Capítulo 4, mostrando como é desenvolvida a relação da distribuição de chaves efetuada por elas com a variação do número de chaves acumuladas (*threshold*) $X$, número de passos $Y$, e número de particionamentos por passo $\beta$. Demonstrando como as meta-heurísticas podem resolver o problema do *partitioning skew* no *MapReduce* de maneira mais eficiente que a distribuição por *hash* presente na implementação do *Hadoop YARN* 2.7.2, e que a abordagem gulosa apresentada em (LE JIANGCHUAN LIU, 2014). Devido ao grande número de experimentos realizados, dividimos o capítulo de avaliação experimental em duas seções para facilitar a exposição dos cenários.

Na primeira seção, apresentada na Tabela 2, comparamos a meta-heurística *Simulated Annealing* com a função de particionamento por *hash* e com a abordagem gulosa através de três cenários onde cada cenário utiliza um número de passos $Y$ fixo, onde $Y \in \{10000, 30000, 50000\}$ e apresenta o resultado daquele número de passos com cada variação de *threshold* $X\%$, onde $X \in \{5, 10, 15, 20, 30\}$. Finalizamos a primeira seção com um quarto cenário onde comparamos os resultados com *threshold* $X = 30\%$ com os três valores de temperatura.

Tabela 2 – Organização dos cenários na Seção 5.1

| Y = 10000 | Y = 30000 | Y = 50000 | X = 30% |
|---|---|---|---|
| Cenário 1 | Cenário 2 | Cenário 3 | Cenário 4 |

Na segunda seção, apresentada nas tabelas 3 e 4, comparamos as meta-heurísticas *Local Beam Search* e *Stochastic Beam Search* com os resultados da *Simulated Annealing* apresentados na seção anterior, a função de particionamento por *hash*, e com a abordagem gulosa. Devido ao grande volume de experimentos, apresentamos um total de 45 cenários divididos em 9 grupos de cinco onde cada cenário apresenta uma variação de *threshold*, e cada grupo possui um valor fixo de passos $Y$, e um valor fixo de particionamento por passos $\beta$ onde $\beta \in \{1, 3, 5\}$ que representa quantos particionamentos serão produzidos e descartados a cada passo dado pelas meta-heurísticas *Local Beam Search* e *Stochastic Beam Search*. Ao final, apresentamos os dez melhores resultados dentre todos os testes feitos com 100% do conjunto de dados em comparação com o que se esperaria de um resultado ótimo e do resultado guloso.

Tabela 3 – Organização dos grupos na Seção 5.2

| $\beta \backslash Y$ | Y = 10000 | Y = 30000 | Y = 50000 |
|---|---|---|---|
| $\beta = 1$ | Grupo 1 | Grupo 2 | Grupo 3 |
| $\beta = 3$ | Grupo 4 | Grupo 5 | Grupo 6 |
| $\beta = 5$ | Grupo 7 | Grupo 8 | Grupo 9 |

Tabela 4 – Organização dos cenários nos grupos da Seção 5.2

| X = 05% | X = 10% | X = 15% | X = 20% | X = 30% |
|---|---|---|---|---|
| Cenário 1 | Cenário 2 | Cenário 3 | Cenário 4 | Cenário 5 |

Realizamos um total de 525 experimentos utilizando uma versão do *hadoop YARN* 2.7.2 que modificamos para utilizar o nosso sistema de particionamento no qual é possível escolher entre a abordagem por *hash*, gulosa, ou uma das três meta-heurísticas. Utilizamos 10 máquinas virtuais com 16 GB de RAM, 16 vCPUs e 160 GB de armazenamento. Todas as máquinas virtuais estão rodando no ambiente de nuvem *OpenStack* hospedado no LSBD/UFC. Para a realização dos experimentos utilizamos como base um algoritmo de contagem de palavras sobre um conjunto de dados reais compostos dos dados das avaliações dos livros da loja da Amazon (MCAULEY *et al.*, 2015b; MCAULEY *et al.*, 2015a) de aproximadamente 400MB divididos em arquivos de tamanho $T_{arq} \in \{80MB, 160MB, 240MB, 320MB, 400MB\}$.

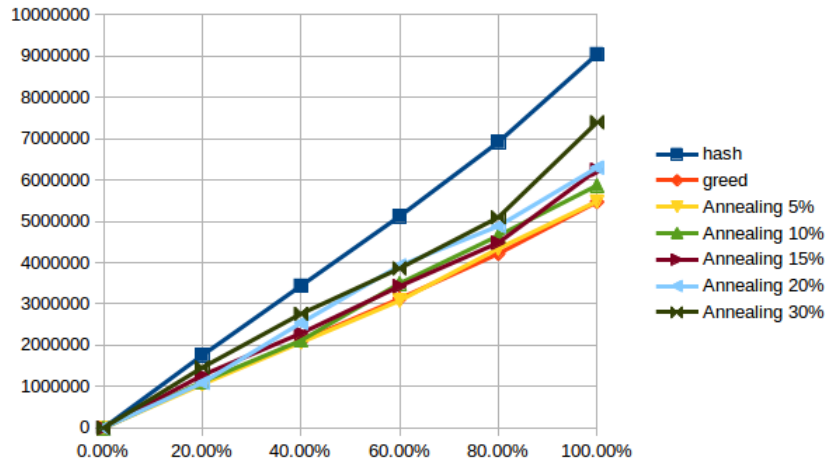
Para executar as meta-heurísticas, acumulamos as chaves geradas como dados intermediários até que um limite dado por um *threshold* de $X\%$, com $X \in \{5, 10, 15, 20, 30\}$, seja alcançado. Após isso, executamos a meta-heurística selecionada onde ela fará um número de interações $Y$ onde $Y \in \{10000, 30000, 50000\}$. Ao finalizar a primeira execução da meta-heurística, todas as chaves acumuladas são particionadas de acordo com o resultado dado pela meta-heurística selecionada e acumulamos novas chaves para repetir o processo quando o mesmo *threshold* for alcançado novamente. Para a medição de um bom particionamento é necessário que a maior partição convirja para um tamanho $T = \frac{N_{keys}}{N_{parts}}$ onde $N_{keys}$ é número total de chaves e $N_{parts}$ é o número de partições.

## 5.1 SIMULATED ANNEALING

Nosso principal objetivo nessa seção é analisar a performance da meta-heurística *Simulated Annealing* de acordo com o número de passos $Y$ e o tamanho do acumulo de chaves limitado pelo *threshold* $X$, onde valores maiores de $X$ e $Y$ implicam em um maior custo compu-

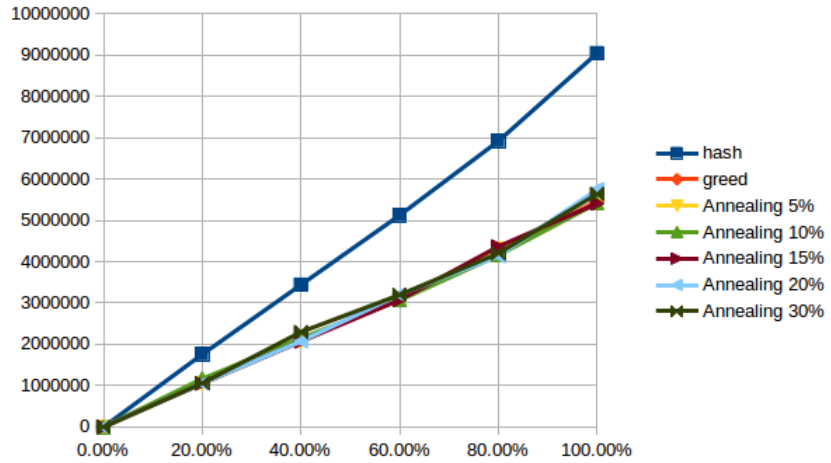
tacional. Como métrica para medir o desempenho do particionamento será utilizada o tamanho da maior partição, isto é, o número de chaves alocadas a ela.

Figura 14 – Comparação da execução do algoritmo *hash* padrão do *hadoop* e do algoritmo guloso com a *simulated annealing* com diversos *thresholds* usando 10000 passos como temperatura. No eixo X é representado o tamanho do conjunto de dados processado indo de 0% a 100%. No eixo Y é representado o tamanho da maior partição em quantidade de chaves



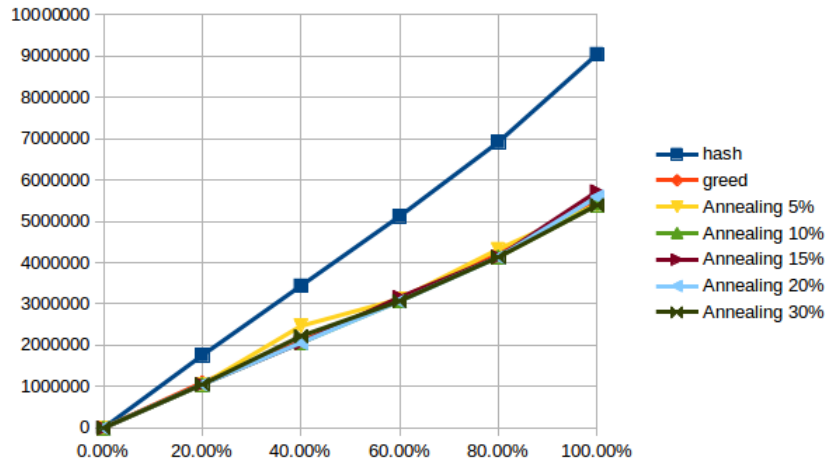

No primeiro cenário representado na Figura 14 variamos o *threshold* utilizado para acumular chaves e fixamos a temperatura no valor $Y = 10000$ como mostrado na Tabela 2. Mesmo o resultado cuja maior partição deu o maior resultado com 10000 passos possui um balanceamento mais satisfatório do que a função de particionamento por *hash*. Além disso, é possível traçar uma relação inicial entre temperatura e *threshold* uma vez que quanto maior o *threshold*, mais complicado se torna o balanceamento. Isso se deve ao fato de quanto mais dados acumulados precisarem ser particionados mais passos serão necessários para equiparar as partições.

Figura 15 – Comparação da execução do algoritmo *hash* padrão do *hadoop* e do algoritmo guloso com a *simulated annealing* com diversos *thresholds* usando 30000 passos como temperatura. No eixo X é representado o tamanho do conjunto de dados processado indo de 0% a 100%. No eixo Y é representado o tamanho da maior partição em quantidade absoluta de chaves




No cenário seguinte, representado na Figura 15, repetimos os testes de variação de *threshold* aumentando a quantidade de passos para $Y = 30000$. O número de passos já é o suficiente para convergir todas as partições para um valor próximo ao ideal. Essa mudança do tamanho das partições em comparação com o que foi observado no cenário anterior confirma que o aumento de temperatura é essencial para alcançar um bom resultado quando o acumulo de chaves é muito grande.

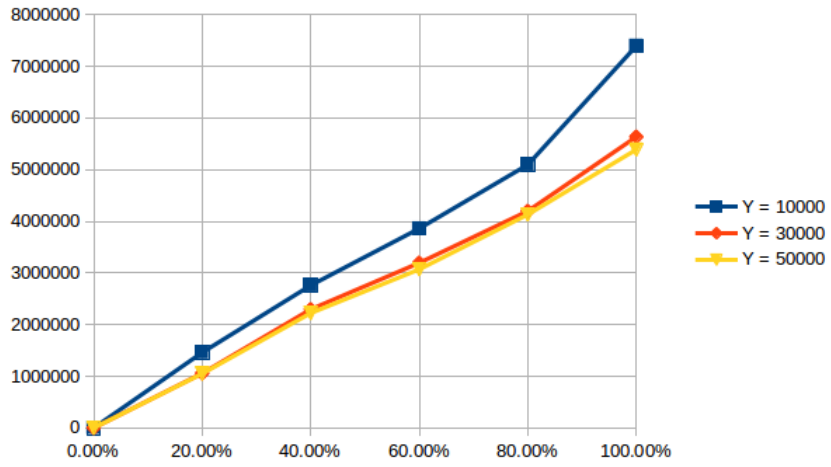
Figura 16 – Comparação da execução do algoritmo *hash* padrão do *hadoop* e do algoritmo guloso com a *simulated annealing* com diversos *thresholds* usando 50000 passos como temperatura. No eixo X é representado o tamanho do conjunto de dados processado indo de 0% a 100%. No eixo Y é representado o tamanho da maior partição em quantidade absoluta de chaves



No último cenário da *Simulated Annealing*, representado na Figura 16, aumentamos a quantidade de passos para 50000; mesmo com o aumento do número de passos, os resultados não apresentam uma diferença significativa do resultado anterior. Isso caracteriza que, não apenas deve existir um valor máximo para o número de passos, como também que um número muito grande de passos não é garantia de que a conversão das partições alcançará um resultado melhor. Isso se deve ao fato de estarmos trabalhando com algoritmos não determinísticos.

Para melhor ilustrar a questão da temperatura, trazemos a Figura 17, onde comparamos os três valores de $Y$. Com uma escala levemente reduzida em comparação às figuras anteriores, é possível observar melhor a conclusão obtida pelos cenários de temperatura fixa onde a diferença da maior partição com o número de passos $Y = 10000$ e $Y = 30000$ é muito maior do que entre os valores de $Y = 30000$ e $Y = 50000$ mostrando que um número exageradamente grande ou pequeno de passos pode não ser necessário para o bom desempenho da meta-heurística e que é necessário buscar um equilíbrio para esse fator.

Figura 17 – Comparação dos resultados dos três números de passos com *threshold* = 30%

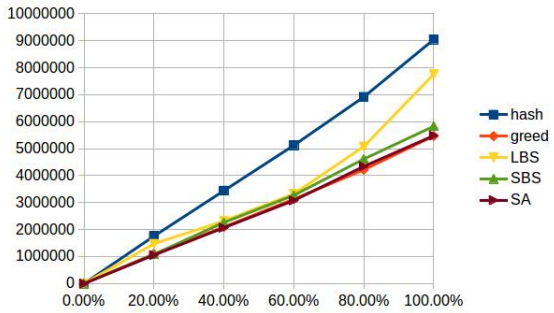



## 5.2 *LOCAL BEAM SEARCH* E *STOCHASTIC BEAM SEARCH*

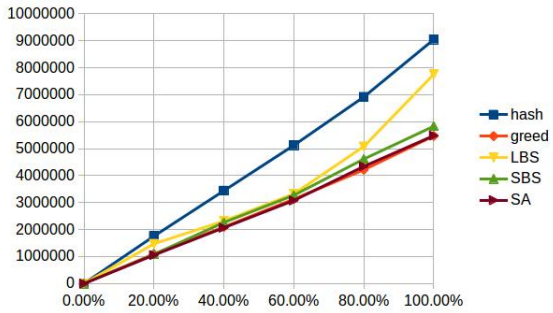
Para analisar o desempenho das meta-heurísticas *Local Beam Search* e *Stochastic Beam Search*, é necessário observar o comportamento do número de passos e do *threshold* citados na seção anterior bem como do número de particionamentos por passo $\beta$ onde $\beta \in \{1,3,5\}$. A apresentação será feita em 9 grupos de 5 cenários onde cada grupo terá o número de passos $Y$ e o número de particionamentos por passo $\beta$ fixado e mostrará um valor diferente do *threshold* $X$ para cada cenário no grupo conforme demonstrado nas tabelas 3 e 4. A comparação em cada cenário será feita entre as três meta-heurísticas, o algoritmo guloso e o método *hash* padrão.

Ao final, mostraremos os melhores resultados dentre todos os cenários em comparação com o método guloso e o método *hash*.

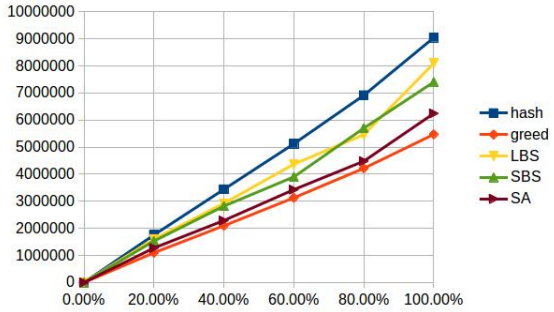
Figura 18 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



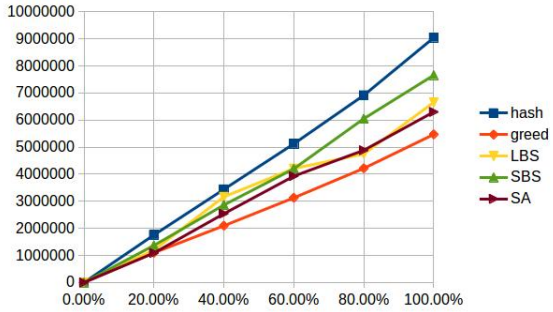

(a) $\beta$=1, passos = 10000, *threshold* = 05%



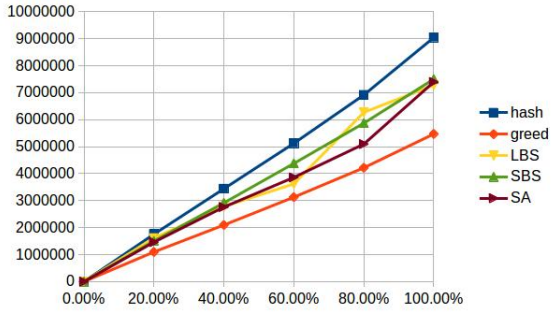

(b) $\beta$=1, passos = 10000, *threshold* = 10%




(c) $\beta$=1, passos = 10000, *threshold* = 15%




(d) $\beta$=1, passos = 10000, *threshold* = 20%

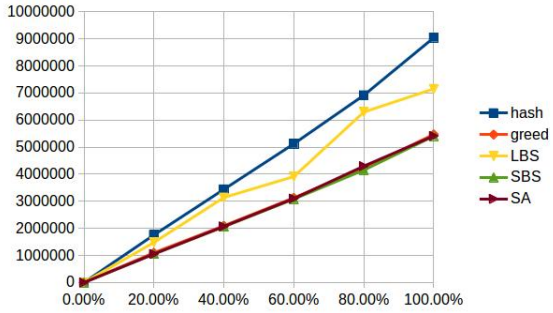



(e) $\beta$=1, passos = 10000, *threshold* = 30%

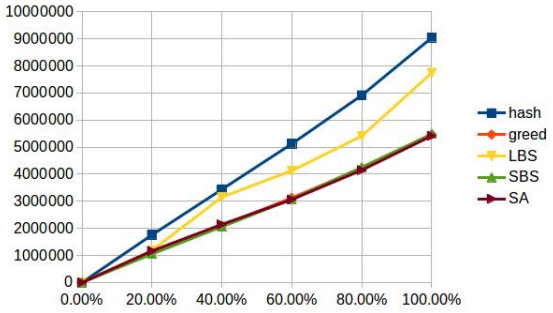
Nos primeiros cinco cenários, representados na Figura 18, fixamos o número de passos $Y = 10000$ e o número de particionamentos por passo $\beta = 1$. É possível notar que, independente do valor dado para o *threshold*, a ausência de uma quantidade significativa de passos e de um valor maior para o número de particionamentos por passo faz com que a *local beam search* não possua uma boa convergência conforme o aumento do número de chaves. Em contrapartida, a *Stochastic Beam Search* consegue um bom rendimento com *thresholds* mais

baixos de maneira similar ao que já havia sido observado na *Simulated Annealing* na seção anterior. Entretanto, a falta de um fator que diminua as chances da *Stochastic Beam Search* escolher o resultado ruim perante o bom, semelhante ao que ocorre na *Simulated Annealing*, acarreta em uma má distribuição das chaves com *threshold* mais altos.

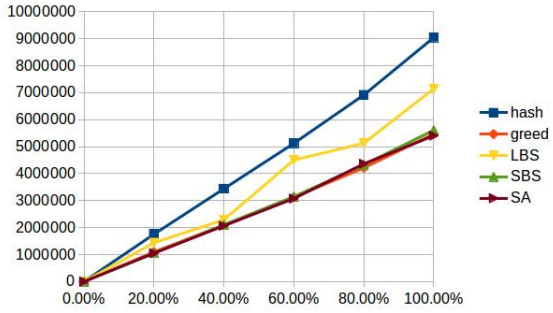
Figura 19 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



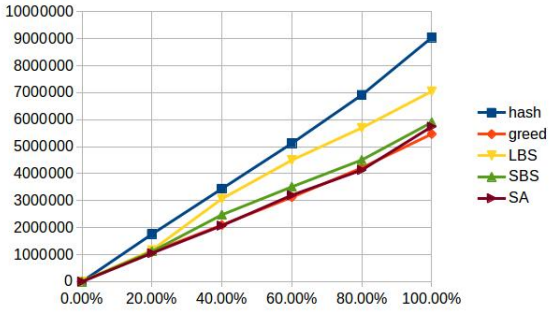

(a) $\beta$=1, passos = 30000, *threshold* = 05%



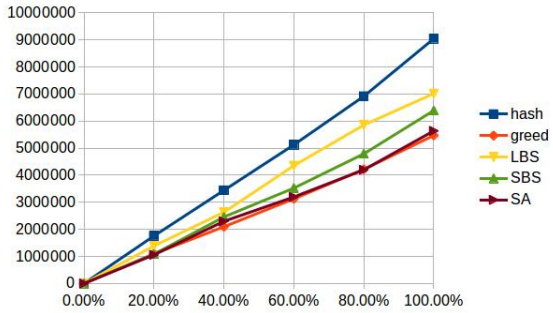

(b) $\beta$=1, passos = 30000, *threshold* = 10%




(c) $\beta$=1, passos = 30000, *threshold* = 15%




(d) $\beta$=1, passos = 30000, *threshold* = 20%

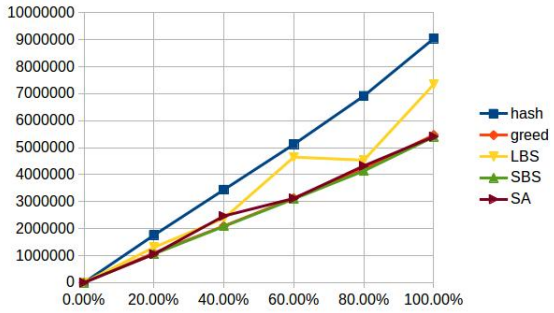



(e) $\beta$=1, passos = 30000, *threshold* = 30%

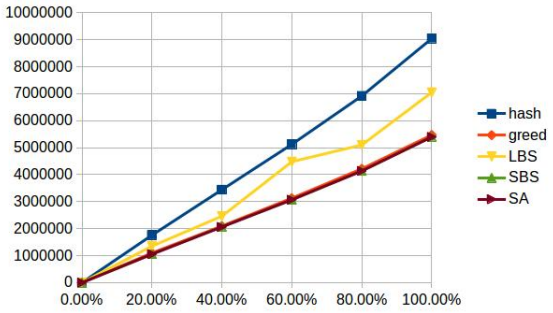
Nos cinco cenários seguintes, representados na Figura 19, aumentamos a temperatura para 30000 e mantemos $\beta = 1$. Notamos que, mesmo sem variar o número de particionamentos por passo, a *Stochastic Beam Search* alcança resultados satisfatórios em todos os cenários exceto o de *threshold* mais alto o que demonstra uma leve dificuldade em convergir. Em contrapartida,

a *Local Beam Search* ainda não possui recursos suficientes para convergir a um valor desejado, com seus resultados apresentando apenas uma pequena melhora com relação aos resultados anteriores.

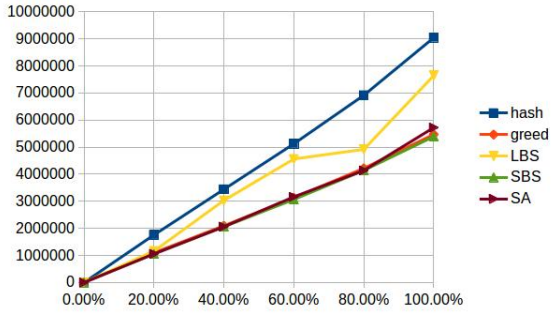
Figura 20 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



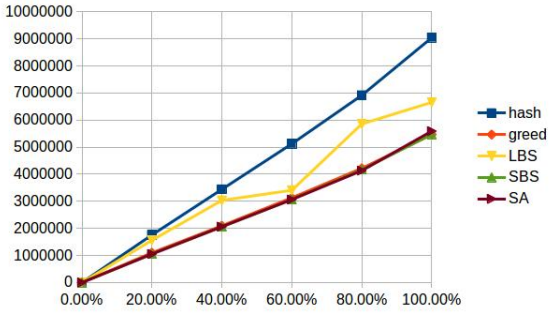

(a) $\beta$=1, passos = 50000, *threshold* = 05%



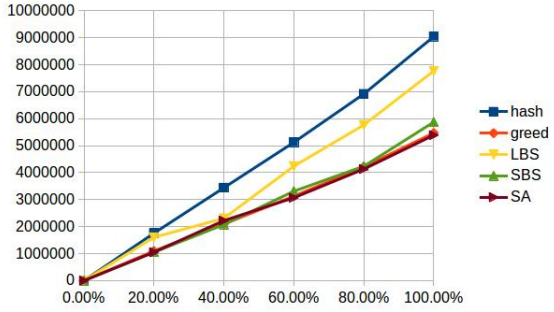

(b) $\beta$=1, passos = 50000, *threshold* = 10%




(c) $\beta$=1, passos = 50000, *threshold* = 15%




(d) $\beta$=1, passos = 50000, *threshold* = 20%

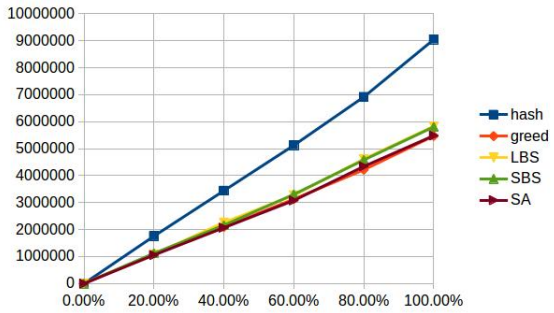



(e) $\beta$=1, passos = 50000, *threshold* = 30%

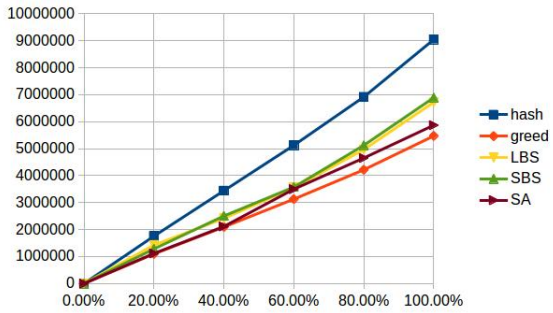
Os cinco últimos cenários com $\beta = 1$, representados na Figura 20, são realizados com 50000 passos. Os resultados apresentados pela *Local Beam Search* não diferem dos resultados mostrados nos experimentos anteriores, mostrando que apenas o número de passos não é um fator impactante para que essa meta-heurística consiga um bom desempenho. Entretanto, a *Stochastic Beam Search* passa a convergir em todos os cenários, mostrando que essa meta-heurística, mesmo

com uma quantidade de particionamentos por passo pequena, é possível convergir para um bom resultado, se a quantidade de passos for alta o suficiente.

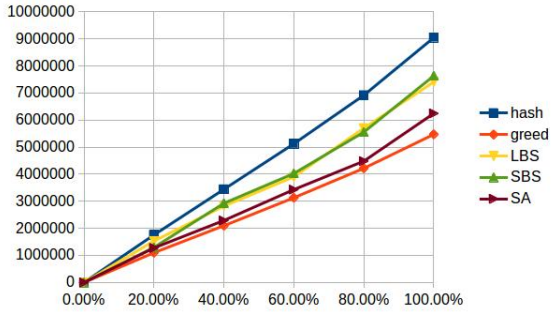
Figura 21 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



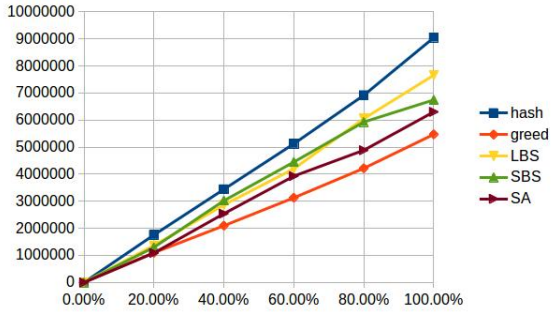

(a) $\beta$=3, passos = 10000, *threshold* = 05%



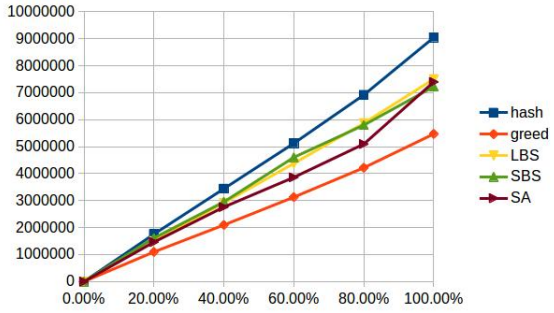

(b) $\beta$=3, passos = 10000, *threshold* = 10%




(c) $\beta$=3, passos = 10000, *threshold* = 15%




(d) $\beta$=3, passos = 10000, *threshold* = 20%

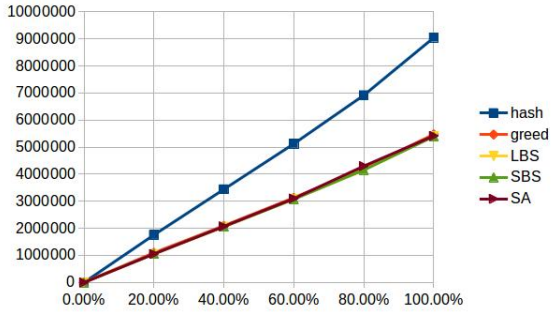



(e) $\beta$=3, passos = 10000, *threshold* = 30%

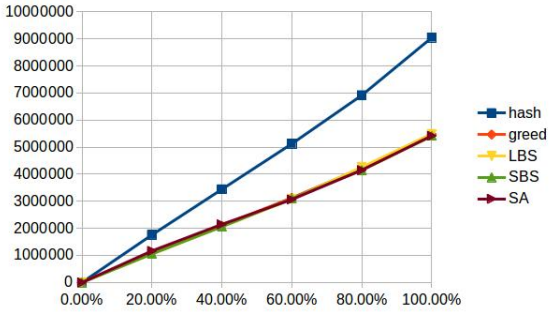
Nos cinco cenários seguintes, representados na Figura 21 alteramos o valor de $\beta$ para 3 e regredimos o número de passos para 10000. Diferente do que foi observado nos primeiros cinco experimentos com essa mesma quantidade de passos, a *Local Beam Search* passa a ter um comportamento mais parecido com a *Stochastic Beam Search* mostrando assim que o número de particionamentos por passo é um fator importante para adquirir bons resultados com a *Local Beam Search*. Além disso, os resultados obtidos pela *Stochastic Beam Search* não apresentam

uma diferença tão significativa, mostrando que, diferentemente da sua antecessora, o número de particionamentos por passo não é tão significativo para a *Stochastic Beam Search* quando o número de passos é baixo.

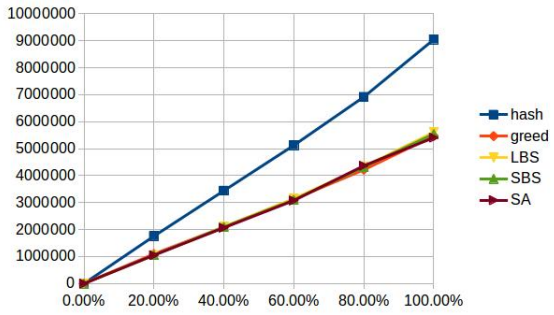
Figura 22 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



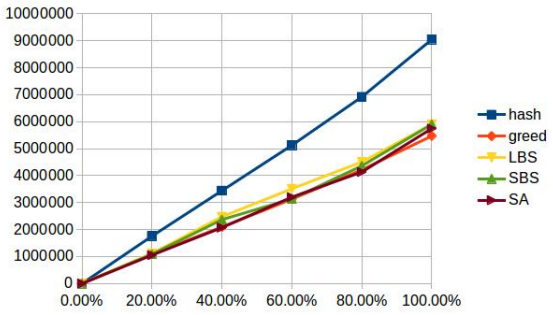

(a) $\beta$=3, passos = 30000, *threshold* = 05%



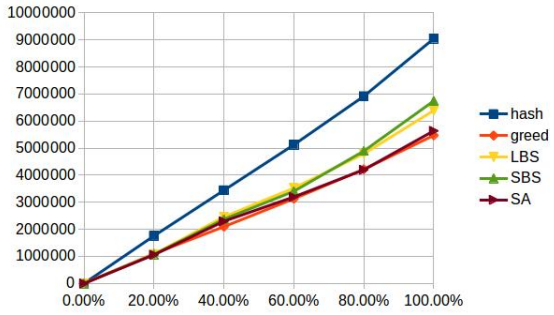

(b) $\beta$=3, passos = 30000, *threshold* = 10%




(c) $\beta$=3, passos = 30000, *threshold* = 15%



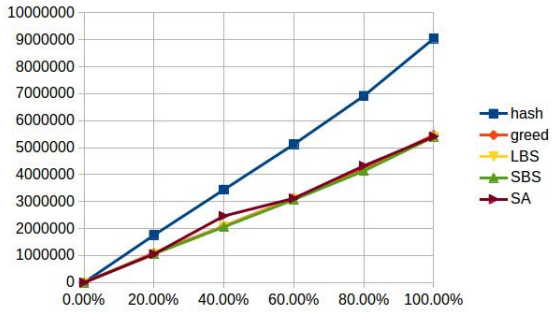

(d) $\beta$=3, passos = 30000, *threshold* = 20%



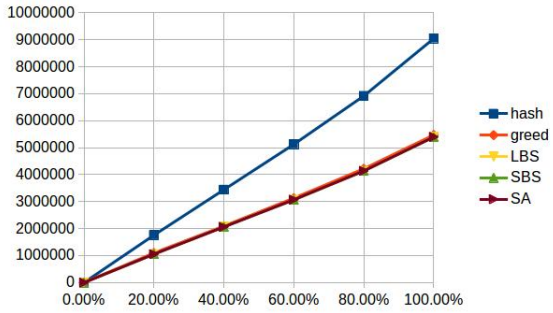

(e) $\beta$=3, passos = 30000, *threshold* = 30%

Nos cinco cenários com $\beta = 3$ e número de passos = 30000, representados na Figura 22, a *Local Beam Search* repete o comportamento visto no grupo anterior, obtendo resultados semelhantes aos da *Stochastic Beam Search*. Enquanto isso, a *Stochastic Beam Search* apresenta uma sutil melhora em relação aos resultados com $Y = 30000$ e $\beta = 1$.

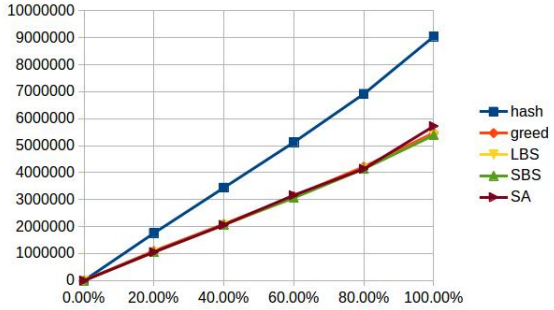
Figura 23 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



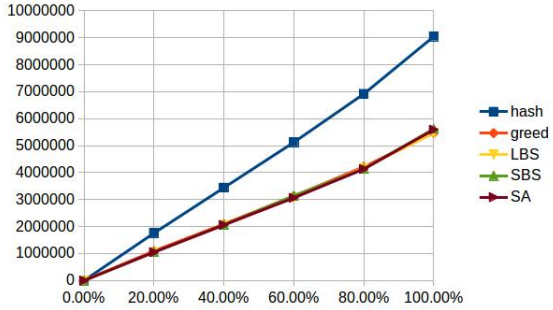

(a) $\beta$=3, passos = 50000, *threshold* = 05%



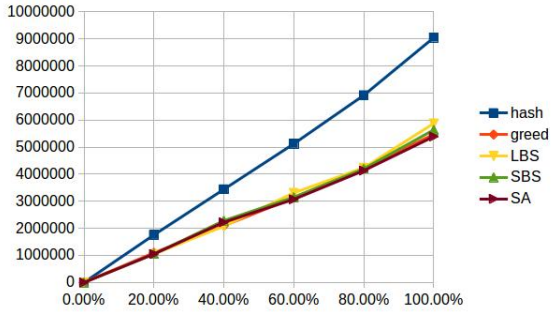

(b) $\beta$=3, passos = 50000, *threshold* = 10%




(c) $\beta$=3, passos = 50000, *threshold* = 15%



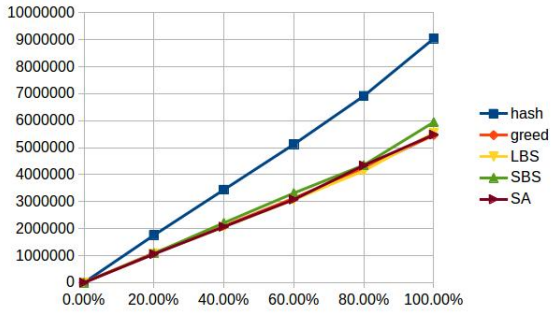

(d) $\beta$=3, passos = 50000, *threshold* = 20%



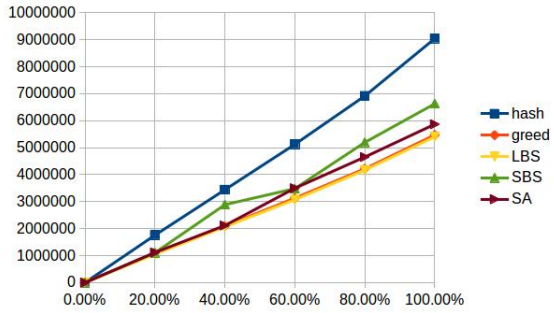

(e) $\beta$=3, passos = 50000, *threshold* = 30%

Nos últimos cinco cenários com $\beta = 3$, representados na Figura 23, elevamos a temperatura para o valor de 50000 e todos os experimentos convergiram para um valor satisfatório, um resultado que era esperado pela *Local Beam Search* pelo modo como ela se aproximava da *Stochastic Beam Search* e pelo que a *Stochastic Beam Search* já havia alcançado com $\beta = 1$ e mesmo número de passos.

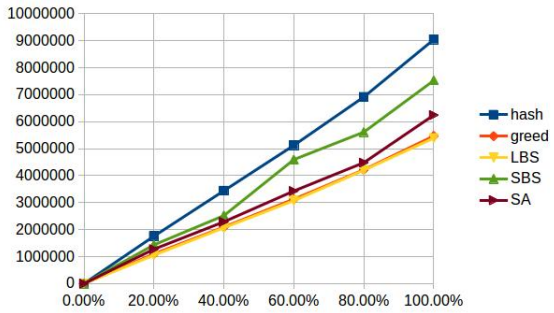
Figura 24 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



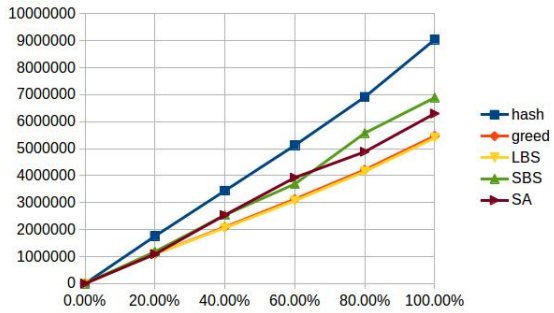

(a) $\beta$=5, passos = 10000, *threshold* = 05%



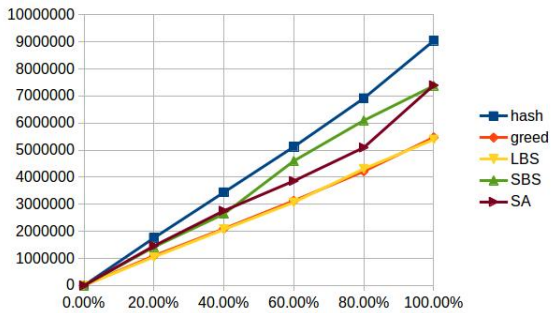

(b) $\beta$=5, passos = 10000, *threshold* = 10%




(c) $\beta$=5, passos = 10000, *threshold* = 15%




(d) $\beta$=5, passos = 10000, *threshold* = 20%

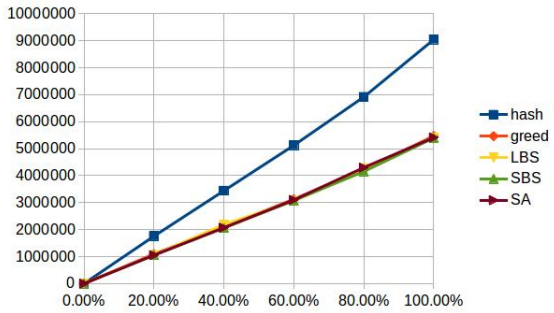



(e) $\beta$=5, passos = 10000, *threshold* = 30%

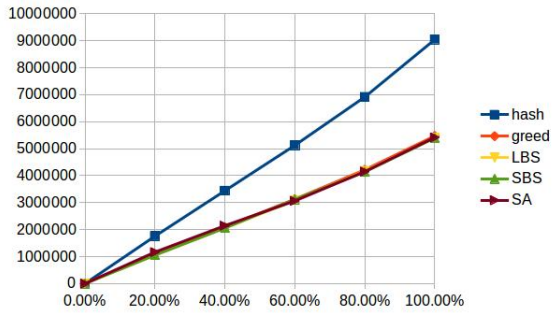
Os primeiros cinco cenários com o valor $\beta = 5$ são representados na Figura 24. Neles, retornamos o número de passos para 10000 e podemos constatar em definitivo que, para esse problema, o número de particionamentos por passo influencia mais no desempenho da *Local Beam Search* do que o número de passos ou o *threshold*. Até mesmo o tamanho do arquivo não parece gerar um problema para essa meta-heurística, que obteve os piores resultados em quase todos os cenários com $\beta = 1$. Entretanto, para a *Stochastic Beam Search* podemos chegar a conclusão de que o valor do $\beta$ é menos relevante que o número de passos. Isso ocorre porque, como a *Stochastic Beam Search* possui uma chance de escolher caminhos piores para a fase seguinte, os ganhos de uma interação boa podem ser mitigados por uma interação ruim, mesmo

que os resultados melhores possuam uma chance mais alta de serem escolhidos.

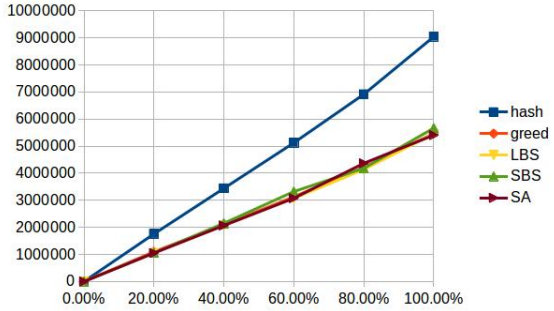
Figura 25 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



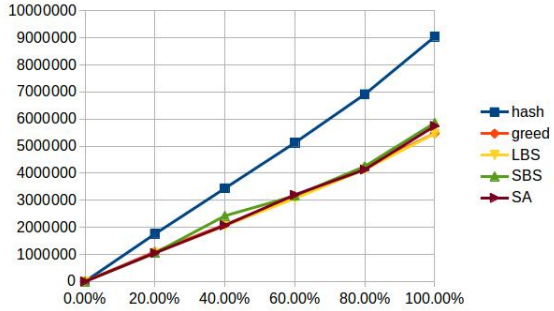

(a) $\beta$=5, passos = 30000, *threshold* = 05%



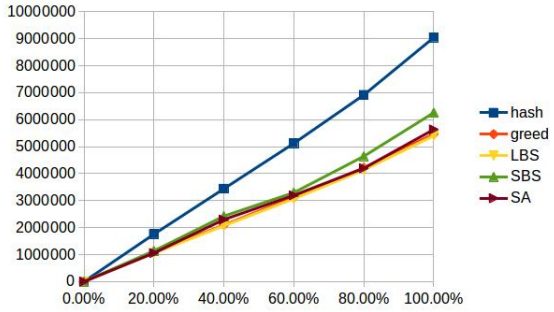

(b) $\beta$=5, passos = 30000, *threshold* = 10%




(c) $\beta$=5, passos = 30000, *threshold* = 15%



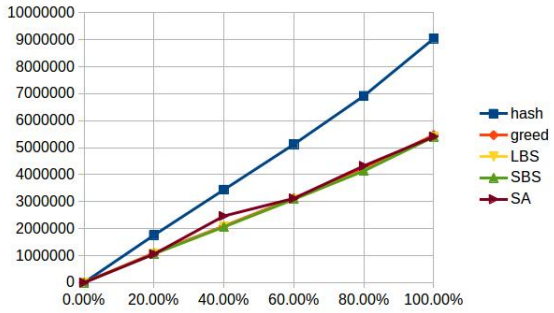

(d) $\beta$=5, passos = 30000, *threshold* = 20%



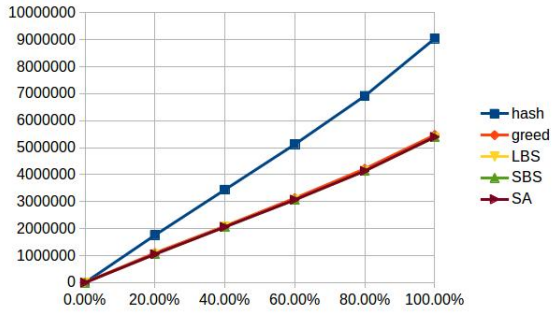

(e) $\beta$=5, passos = 30000, *threshold* = 30%

Nos cinco cenários representados na Figura 25 aumentamos o número de passos para 30000. Não ocorreram alterações com relação aos resultados anteriores da *Local Beam Search* e, assim como já foi observado nos outros resultados de temperatura semelhante, a *Stochastic Beam Search* converge para um bom resultado, com exceção de um leve desvio no *threshold* mais alto.

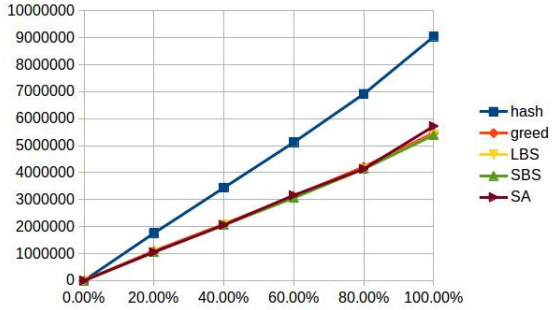
Figura 26 – Comparação entre as meta-heurísticas com variação de *threshold*. Eixo X representa o tamanho do conjunto de dados em porcentagem. Eixo Y representa o tamanho da maior partição em quantidade absoluta de chaves



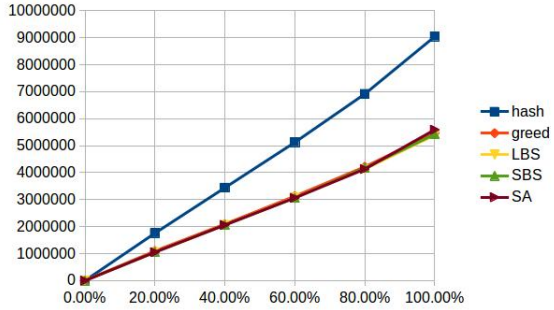

(a) $\beta$=5, passos = 50000, *threshold* = 05%



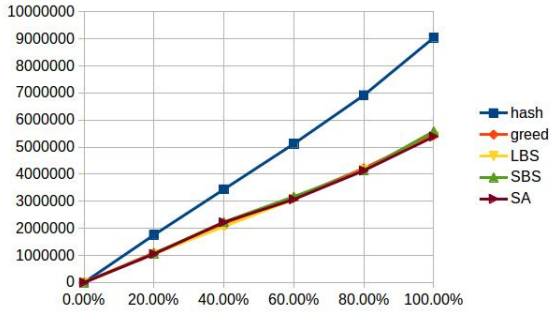

(b) $\beta$=5, passos = 50000, *threshold* = 10%




(c) $\beta$=5, passos = 50000, *threshold* = 15%




(d) $\beta$=5, passos = 50000, *threshold* = 20%

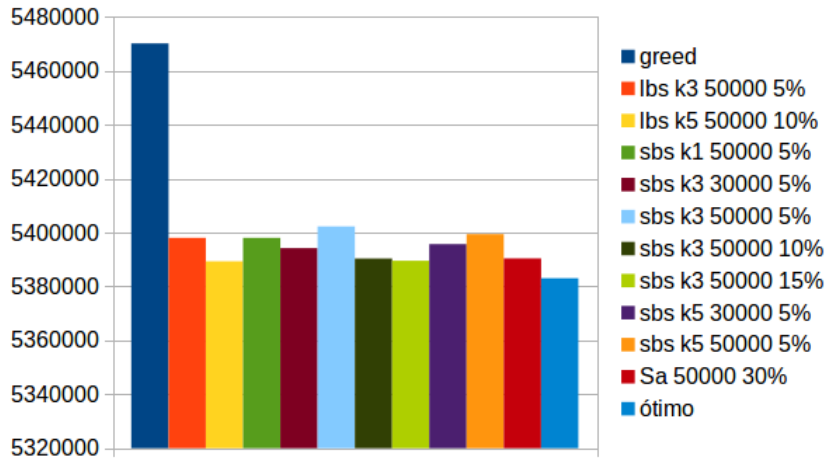



(e) $\beta$=5 passos = 50000, *threshold* = 30%

Nos últimos cinco cenários, representados na Figura 26, o número de passos foi elevado para 50000 e ambas as meta-heurísticas se aproximam do resultado ideal tendo em vista o fato de todos os cenários convergirem para valores melhores ou iguais que o resultado do algoritmo guloso, o que era um resultado esperado, uma vez que em todos os experimentos com $\beta = 5$ a *Local Beam Search* obteve resultados ótimos independente dos outros dois fatores, e, em todos os experimentos com número de passos igual a 50000, a *Stochastic Beam Search* obteve resultados ótimos também independente dos outros fatores.

No gráfico na Figura 27, trazemos uma comparação entre os dez melhores dos 105 resultados das três meta-heurísticas com 100% do conjunto de dados,o resultado do algoritmo

Figura 27 – Comparação do resultado do algoritmo guloso e do resultado ótimo com os dez melhores resultados obtidos




guloso (*greed*), e o resultado ótimo. Apesar da variação dos valores que cada cenário pode obter devido à natureza não determinística das meta-heurísticas, essa comparação é importante para que se possa perceber a distância entre os resultados. Em um experimento com pouco mais de 53 milhões de chaves, a diferença do algoritmo guloso para os melhores resultados foi de pouco mais de 60 mil chaves, e a dos melhores resultados para o ótimo foi de pouco menos de 20 mil.

## 5.3 CONCLUSÃO

Neste capítulo, foram apresentados os resultados obtidos através dos experimentos com as três meta-heurísticas. A meta-heurística *Simulated Annealing*, por ser mais simples, conseguiu convergir para um bom resultado com uma quantidade razoável de passo e um *threshold* baixo. A meta-heurística *Stochastic Beam Search* com número de particionamento por passos igual a 1 possui um comportamento levemente diferente da *Simulated Annealing* devido à seleção entre o novo particionamento de cada passo e o particionamento já existente ser feito de maneira menos controlado, não buscando sempre o melhor. Quando é usado um número suficiente de passos o comportamento é quase que idêntico e quando é aumentado o número de particionamentos por passo a meta-heurística tende a apresentar melhores resultados, mesmo com temperaturas mais baixas. Por fim, a meta-heurística *Local Beam Search* precisa de um maior número de particionamentos por passo para apresentar alguma melhora, mas o aumento desse fator pode bastar para que ela alcance um bom desempenho. Os resultados obtidos foram dependentes do conjunto de dados e do problema de *wordcount*, mas espera-se que não ocorra uma variação grande dos resultados em problema semelhantes uma vez que esse problema de

otimização pode ser repetido em outros cenários.

# 6 CONSIDERAÇÕES FINAIS

## 6.1 CONCLUSÃO

O aumento do volume de dados produzidos globalmente trouxe um gargalo para o processamento de dados. Uma máquina de grande porte não era o suficiente ou era cara demais para ser utilizada. Tendo em mãos a disponibilidade de usar máquinas de pequeno porte para parte do processamento, deixou-se de procurar apenas por aumentar os recursos computacionais das máquinas e passou-se a agrupar essas máquinas em *clusters* para que juntas resolvessem o problema de maneira mais eficiente usando técnicas de paralelismo e divisão-e-conquista. Apesar de acelerar o processamento desses grandes volumes de dados graças à divisão desses dados em blocos menores, essa estratégia possui problemas de otimização. Uma vez que o resultado final é dependente do desempenho individual de cada máquina, garantir que todas trabalhem nas mesmas condições torna-se uma medida fundamental para o bom funcionamento das mesmas. Caso contrário, quando uma máquina demora mais para computar do que as outras, essa máquina passa a ser responsável pelo tempo total da execução uma vez que as outras estão ociosas.

Com isso em mente, a busca por estratégias que executem um particionamento balanceado ganha uma atenção devido à dificuldade em garantir um bom desempenho; dentre as estratégias buscadas encontra-se: executar um plano de particionamento antes de iniciar a fase de *Map*; aumentar os recursos das máquinas que ficarem sobrecarregadas; remover tarefas de máquinas sobrecarregadas e passá-las para outras máquinas sempre que ficarem ociosas; particionar todos os dados intermediários antes de começar a fase de *Reduce*; particionar uma parte inicial dos dados intermediários; deixar que o sistema particione o resto usando como base esse primeiro bloco de dados particionados.

A estratégia apresentada nessa dissertação segue a última ideia citada no parágrafo anterior procurando usar meta-heurísticas como controladoras do particionamento, uma vez que as meta-heurísticas conseguem vasculhar o conjunto de respostas de uma maneira mais ampla. Escolhemos três meta-heurísticas simples que não necessitam de tantos recursos computacionais. *Simulated Annealing* sempre escolhe o novo passo caso ele seja melhor, caso contrário ela permite uma chance de se seguir pelo passo pior com a justificativa de tentar sair do caminho de um ótimo local na procura do ótimo global. A chance do algoritmo seguir pelo caminho ruim diminui conforme o número de passos diminui e o caminho seguido pelo algoritmo vai se estabilizando até que os passos acabem e o particionamento final seja decidido. *Local Beam*

*Search* funciona como uma *Hill Climb* olhando por diversos caminhos aleatórios a cada passo na tentativa de achar os melhores caminhos a se seguir. Ela descarta metade dos caminhos que são analisados a cada interação, os considerando como resultados ruins, e cria um novo caminho para cada particionamento restante usando-os como base, escolhendo apenas o melhor de todos ao final da execução. *Stochastic Beam Search* funciona como uma evolução da *Local Beam Search* que atribui um peso para cada resultado e uma chance dele ser escolhido baseada no peso atribuído. Essa chance é maior para os resultados melhores. Isso ajuda a meta-heurística a manter um grupo de caminhos bons. A possibilidade de um resultado ruim ser escolhido ocorre pelo mesmo motivo descrito na *Simulated Annealing*, a possibilidade de evitar um ótimo local em prol de um ótimo global.

Nossa estratégia foi comparada com a estratégia gulosa de (LE JIANGCHUAN LIU, 2014) em um conjunto de dados real colhido por (MCAULEY *et al.*, 2015b; MCAULEY *et al.*, 2015a). Até certo ponto, a estratégia gulosa serve como um limitante inferior devido ao seu bom desempenho. Esse bom desempenho se deve à estrutura do conjunto de dados que possui uma grande variação de chaves de tamanhos diferentes, o que possibilita a montagem de um bom particionamento pelo método guloso mesmo quando a maioria das chaves já foi encaminhada para as suas devidas partições. Essa mesma estrutura permite que a *Local Beam Search* alcance melhores resultados quando executa mais de um particionamento por passo, e atrapalha parcialmente as outras duas meta-heurísticas devido a ambas terem uma tendência de fazer algumas escolhas ruins ao longo de sua execução. Entretanto, mesmo com esses desvios, ambas conseguem convergir para bons resultados com uma quantidade suficientemente grande de passos.

## 6.2 TRABALHOS FUTUROS

A estratégia do *MapReduce* se desenvolveu como base para o problema da computação em *cluster* e o *Hadoop* é apenas um dos modos como ela foi posta em prática. Com a evolução de como a estratégia é aplicada são necessários estudos para identificar o *partitioning skew* em outras implementações do *MapReduce* e possíveis formas de resolvê-los.

A maneira como o sistema armazena os dados intermediários é um grande gargalo que força a utilização de mais recursos computacionais do que o necessário para aplicar qualquer técnica de particionamento. Otimizar a estrutura interna do armazenamento é crucial para melhorar o desempenho das estratégias aqui citadas.

As três meta-heurísticas abordadas nesse trabalho não são as únicas capazes de diminuir o impacto do *partitioning skew*. Assim como o *wordcount* não é o único a apresentar esse tipo de problema. Um estudo entre o desempenho de outras meta-heurísticas para esse tipo de problema e a utilização de outros problemas inicias, tais como *pagerank*, ajudaria na escolha de novas opções de particionamento para cada caso. Finalmente, testamos as meta-heurísticas utilizando apenas um problema dentre vários existentes. A mudança do problema, do conjunto de dados e dos parâmetros poderiam dar resultados completamente diferentes do que foi apresentado nesse trabalho.

# REFERÊNCIAS

ATTA, F.; VIGLAS, S. D.; NIAZI, S. Sand join—a skew handling join algorithm for google's mapreduce framework. In: IEEE. **Multitopic Conference (INMIC), 2011 IEEE 14th International**. [S.l.], 2011. p. 170–175.

DAS, D. A. **Layout of Integrated Circuits using Simulated annealing**. 2014. Disponível em: <https://www.slideshare.net/deviarchanadas16/simulated-annealing-39938986>.

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

FACEBOOK. **Under the Hood: Scheduling MapReduce jobs more efficiently with Corona**. 2012. Disponível em: <https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920/>.

GUFLER, B.; AUGSTEN, N.; REISER, A.; KEMPER, A. Handling data skew in mapreduce. **Closer**, v. 11, p. 574–583, 2011.

HE, B.; FANG, W.; LUO, Q.; GOVINDARAJU, N. K.; WANG, T. Mars: a mapreduce framework on graphics processors. In: ACM. **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. [S.l.], 2008. p. 260–269.

IBRAHIM, S.; JIN, H.; LU, L.; WU, S.; HE, B.; QI, L. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In: IEEE. **Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on**. [S.l.], 2010. p. 17–24.

ISARD, M.; BUDIU, M.; YU, Y.; BIRRELL, A.; FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In: ACM. **ACM SIGOPS Operating Systems Review**. [S.l.], 2007. v. 41, n. 3, p. 59–72.

KWON, Y.; BALAZINSKA, M.; HOWE, B.; ROLIA, J. A study of skew in mapreduce applications. **Open Cirrus Summit**, 2011.

KWON Y., B. M. H. B. R. J. Skewtune: mitigating skew in mapreduce applications. **Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data**, ACM, p. 25–36, 2012.

LE JIANGCHUAN LIU, F. E. D. W. Y. Online load balancing for mapreduce with skewed data input. **IEEE Conference on Computer Communications**, 2014.

LIU QI ZHANG, R. B. Y. L. B. W. Z. Optima: On-line partitioning skew mitigation for mapreduce with resource adjustment. **Journal of Network and Systems Management**, 2016.

LIU, Z.; ZHANG, Q.; ZHANI, M. F.; BOUTABA, R.; LIU, Y.; GONG, Z. Dreams: Dynamic resource allocation for mapreduce with data skew. In: IEEE. **Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on**. [S.l.], 2015. p. 18–26.

MCAULEY, J.; PANDEY, R.; LESKOVEC, J. Inferring networks of substitutable and complementary products. In: ACM. **Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. [S.l.], 2015. p. 785–794.

MCAULEY, J.; TARGETT, C.; SHI, Q.; HENGEL, A. van den. Image-based recommendations on styles and substitutes. In: ACM. **Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval**. [S.l.], 2015. p. 43–52.

OKCAN, A.; RIEDEWALD, M. Processing theta-joins using mapreduce. In: ACM. **Proceedings of the 2011 ACM SIGMOD International Conference on Management of data**. [S.l.], 2011. p. 949–960.

PERICINI, M. H.; LEITE, L. G.; MACHADO, J. C. MALiBU: Metaheuristics approach for online load balancing in mapreduce with skewed data input. In: SBC. **Anais do XXXV Simpósio Brasileiro de Redes de Computadores**. [S.l.], 2017. p. 358–371.

RAMAKRISHNAN, S. R.; SWART, G.; URMANOV, A. Balancing reducer skew in mapreduce workloads using progressive sampling. In: ACM. **Proceedings of the Third ACM Symposium on Cloud Computing**. [S.l.], 2012. p. 16.

RANGER, C.; RAGHURAMAN, R.; PENMETSA, A.; BRADSKI, G.; KOZYRAKIS, C. Evaluating mapreduce for multi-core and multiprocessor systems. In: IEEE. **2007 IEEE 13th International Symposium on High Performance Computer Architecture**. [S.l.], 2007. p. 13–24.

RUSSELL, S. J.; NORVIG, P.; CANNY, J. F.; MALIK, J. M.; EDWARDS, D. D. **Artificial intelligence: a modern approach**. [S.l.]: Prentice hall Upper Saddle River, 2003. v. 2.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. In: IEEE. **Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on**. [S.l.], 2010. p. 1–10.