IVAN LUIZ PICOLI

UMA ABORDAGEM DE CLASSIFICAÇÃO NÃO SUPERVISIONADA DE CARGAS DE TRABALHO DE SISTEMAS ANALÍTICOS EM APACHE HADOOP ATRAVÉS DE ANÁLISE DE LOG

CURITIBA

2015

IVAN LUIZ PICOLI

UMA ABORDAGEM DE CLASSIFICAÇÃO NÃO SUPERVISIONADA DE CARGAS DE TRABALHO DE SISTEMAS ANALÍTICOS EM APACHE HADOOP ATRAVÉS DE ANÁLISE DE LOG

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Eduardo Cunha de Almeida

CURITIBA

2015

Picoli, Ivan Luiz
Uma abordagem de classificação não supervisionada de cargas de trabalho de sistemas analíticos em apache hadoop através de análise de log / Ivan Luiz Picoli. – Curitiba, 2014
58 f. : il.; tabs.

Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Eduardo Cunha de Almeida
Bibliografia: p.54-58

1. Apache (Programa de computador). 2. Banco de dados. I. Almeida, Eduardo Cunha de. II. Título

CDD 005.71369

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Ivan Luiz Picoli, avaliamos o trabalho intitulado, *"Uma Abordagem de Classificação Não Supervisionada de Cargas de Trabalho de Sistemas Analíticos em Apache Hadoop através de Análise de Log"*, cuja defesa foi realizada no dia 26 de fevereiro de 2015, às 14:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela: (X) **aprovação** da candidata. ( ) **reprovação** da candidata.

Curitiba, 26 de fevereiro de 2015.

Prof. Dr. Eduardo Cunha de Almeida
DINF/UFPR – Orientador

Prof. Dr. Tarcizio Alexandre Bini
UEPG - Membro Externo

Prof. Dr. Luis Carlos Erpen de Bona
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

"Tape is dead, disk is tape, flash is disk, RAM locality is king;

Main Memory DB is going to be common."

Jim Gray (2006)

AGRADECIMENTOS

Agradeço primeiramente aos espíritos de luz que me acompanham em minha jornada e me ajudam na minha formação e missão. Agradeço a minha mãe por ter me criado e me dado a melhor educação que poderia ter, a minha mãe novamente e meu irmão por me apoiarem, acreditarem e me motivarem a continuar me aperfeiçoando, e pelas tantas vezes que me orientaram e me indicaram o melhor caminho em momentos que precisei.

Agradeço aos meus amigos Paulo, Silvio, Victor, Andrew, Bruno e Felipe os quais dividi meu tempo nos últimos anos e em meio a tantas festas, parcerias e risos me fizeram ver que ninguém é feliz sem ter amigos. Agradeço a todos os meus amigos por fazerem parte de minha vida e me ajudarem a viver em um círculo social divertido e feliz para conseguir estudar.

Agradeço aos Cowboys de los Andes Paulo, Tony, Victor e Silvio por me acompanharem nas loucas viagens pela América do Sul, viagens que me fizeram abrir a mente para novas situações e costumes, além de aprimorarem minha tomada de decisões. Acredito que essa experiência será de grande valia para minha carreira, pois aumentar a visão sobre o mundo, com certeza, enriquecerá futuras pesquisas.

Agradeço meus amigos e chefes Leandro e Ivan que sempre me auxiliaram e me motivaram durante meus estudos. Agradeço ao meu orientador Eduardo por acreditar, entender e me auxiliar em tudo que precisei durante o mestrado, também agradeço ao professor Didonet pelo auxílio e apoio prestado nos últimos anos. Agradeço a todos os professores que um dia dividiram seus conhecimentos comigo.

Agradecimentos especiais ao Ramiro, que sempre esteve junto nas pesquisas e programação, e que mesmo a distância me orientou e me ajudou em tudo que precisei durante o mestrado. Também agradeço aos outros alunos de meu orientador, Tiago, Jorge e Simone por contribuirem no grupo de pesquisas em banco de dados e pelas conversas sobre estudos futuros.

Agradeço à Capes e ao Serpro pelas bolsas de estudo durante o mestrado.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

RESUMO

MapReduce vem sendo utilizado amplamente na área de processamento de dados e Data Warehouse. Entre as muitas implementações do MapReduce disponíveis nos dias de hoje, o Apache Hadoop é o mais popular e possui código aberto. Hadoop também é usado amplamente como motor de muitos sistemas de processamento de consultas baseados em SQL, como o Apache Hive e PIG. Nesses sistemas, desenvolvedores podem processar consultas baseadas em SQL utilizando a escalabilidade do MapReduce. Neste contexto, desempenho e escalabilidade estão diretamente ligados aos parâmetros de configuração, que determinam o consumo de recursos e a eficiência do processamento. Hoje, as abordagens de ajuste de parâmetros analisam as tarefas durante sua execução e geram configurações de parâmetros baseadas em dados contidos em arquivos de log. Apesar de aumentar o desempenho, essa abordagem não é capaz de associar tarefas MapReduce similares para aplicar a configuração necessária. Assim, se tem dois problemas: (1) tarefas MapReduce recebem otimizações através de regras preestabelecidas sem se preocupar com a melhor alocação de recursos; (2) sequências de tarefas MapReduce, como planos de consulta do Hive, recebem a mesma otimização mesmo que diferentes tarefas consumam diferentes recursos (problema nomeado de 'otimização uniforme'). A consequência de ambos os problemas é a perda de desempenho, e o aumento do tempo de resposta e do consumo de recursos. Nesta dissertação apresenta-se uma abordagem que classifica tarefas MapReduce para aplicar otimizações através da similaridade de recursos automaticamente. Essa abordagem é capaz de gerar grupos de tarefas que possuam consumo de recursos similares. Cada grupo criado é associado a uma otimização específica que é aplicada às novas tarefas MapReduce. As duas principais contribuições são: (1) uma nova tarefa MapReduce recebe uma otimização apropriada sem a intervenção humana; (2) Tarefas de diferentes grupos recebem diferentes otimizações (abordagem chamada de 'otimização granular'). Experimentos mostram que nossa abordagem reduz o tempo de resposta em até 20% no melhor caso quando o benchmark TPC-H é executado no Hive e Hadoop.

Palavras-chave: Apache Hadoop, MapReduce, Otimização de Banco de Dados, Otimização Hive, Aprendizado de Máquina, Análise de Log

ABSTRACT

MapReduce has been extensively used for data processing and analytics over the past years. Among the many MapReduce implementations available to date, the Apache Hadoop is one of the most popular due to its open source nature. Hadoop has been also used as data processing back-end by many SQL-like query processing systems, such as Apache Hive and PIG. In these systems, developers can leverage the declarative nature of query languages with the scalability of MapReduce processing. The scalability of Hadoop directly depends on proper performance tuning in order to squeeze computer resources for efficient data processing. To date, any Hadoop tuning approach relies on monitoring the execution of running programs for computing tuning setups based on execution data in log files. While this approach can boost performance, it does not allow associating similar Hadoop programs to a similar tuning setup. This leads to two problems: (1) any upcoming program receives a single pre-computed tuning without any concern to what are the best computing resources to be allocated; (2) chains of running MapReduce programs, such as Hive query plans, tend to receive the same tuning setup, even if they eventually consume different computing resources (this problem is called 'uniform tuning'). The direct consequence of both problems is that they lead to poor performance, increase in response time and in resource consumption. This dissertation presents an approach for classifying MapReduce programs with similar resource consumption into groups. For each group, a specific tuning setup is associated that can be then re-applied to any upcoming program. This approach provides two main contributions: (1) an upcoming program receives a proper tuning on-the-fly without human intervention; (2) programs from different groups receive different tuning setups (this approach is called 'fine-grained tuning'). Empirical experimentation shows that this approach reduces response time in 20% in the best case scenario when running the TPC-H benchmark on Hive and Hadoop.

Keywords: Apache Hadoop, Apache Hive, MapReduce, Database Tuning, HiveQL Tuning, Machine Learning, Log Analysis

CAPÍTULO 1

INTRODUÇÃO

Nos últimos anos, empresas como Facebook, Google e Twitter enfrentam grandes desafios ao processar e ao armazenar grandes quantidades de dados. Segundo a empresa Cisco, em 2015 os dados de redes sociais e transmissão de vídeos em alta resolução [16] atingirão 1.0 ZB, o que equivale a 87125 PB por mês [34].

Recentes técnicas de processamento distribuído vem sendo aprimoradas e implementadas nas mais diversas plataformas para acompanhar o crescimento exponencial de dados e gerenciar os problemas relativos a esse crescimento, aos quais se resumem em processamento, armazenamento e gerenciamento. Um dos paradigmas mais conhecidos é o *MapReduce* [8]. Desenvolvido para simplificar o processamento distribuído, o *MapReduce* torna os problemas de escalabilidade transparentes, possibilitando que um programa de computador seja executado de forma distribuída sem a preocupação com os diversos fatores que dificultam esse tipo de programação, como falhas de rede e a consistência dos dados que se encontram espalhados pela rede.

Processos que necessitam de escalabilidade e plataformas baseadas em nuvem podem ser reduzidos à tarefas de *MapReduce*, juntamente com o ecossistema de aplicações envolvido a partir dessa tecnologia emergente [16]. Uma das principais implementações do *MapReduce* é o Hadoop [3]. O desempenho das tarefas do Hadoop está ligado diretamente aos parâmetros de configuração, e o correto ajuste desses parâmetros faz com que as aplicações aloquem recursos computacionais distribuídos de maneira mais eficiente.

O Hadoop possui características como transparência, escalabilidade e tolerância a falhas, e suas aplicações necessitam do ajuste de parâmetros de configuração para que obtenham um desempenho melhor. A melhoria de desempenho através do ajuste de

parâmetros é uma estratégia utilizada em diversas ferramentas de otimização [14] [17].

O desempenho de tarefas no Hadoop é afetado por diversos fatores, como por exemplo a leitura e escrita em disco que torna-se custoso devido ao sistema de arquivos estar fragmentado na rede. Tarefas com diferentes usos de recursos requerem configurações diferentes em seus parâmetros para que haja diminuição no tempo de resposta da aplicação. Um software autoajustável, o qual ajusta seus parâmetros de acordo com a carga de trabalho submetida é uma solução para a diminuição do tempo de processamento [11].

A dissertação é dividida em quatro partes. O Capítulo 1 é a introdução e a apresentação do problema, desafios e contribuições. O Capítulo 2 é o estado da arte, onde é apresentada a situação atual nas pesquisas de otimização do *MapReduce*, também é apresentada a arquitetura do *MapReduce* e sistemas construídos sobre o mesmo. No Capítulo 3 apresenta-se a abordagem proposta e sua implementação, o *Chameleon*. No Capítulo 4 apresenta-se os experimentos realizados e os trabalhos futuros.

1.1 Problema

Para que um software de autoajuste de parâmetros seja capaz de identificar os valores corretos, faz-se necessário a implementação de regras que classificam as tarefas. Alguns softwares de otimização existentes analisam o possível consumo de recurso computacional antes mesmo da execução propriamente dita; porém, essa classificação não mostra o real consumo de recursos que a tarefa irá utilizar e sim uma projeção por meio de regras impostas. Esse fato gera o seguinte problema:

- Classificar tarefas no Hadoop antes da execução exige a extração de informações através de regras criadas a partir de ambientes específicos, resultando em dados menos precisos do que se a análise fosse realizada após a execução. Ao trabalhar com dados incertos, as tarefas podem ser classificadas erroneamente, e os parâmetros de configuração receberão valores inapropriados ou menos eficazes.

1.2 Desafios

Neste trabalho mostra-se uma abordagem extensível para qualquer sistema baseado em *MapReduce* que seja capaz de gerar arquivos de log de execução das tarefas. Sendo assim, não se sabe que tipo de informações estarão disponíveis para análise. Como se desconhece as informações que serão analisadas, os dados devem ser classificados a partir de seus próprios valores.

Em uma consulta de banco de dados, muitas vezes cada operador da consulta se transforma em uma tarefa *MapReduce* e esse conujunto de tarefas gera o plano de consulta [20]. Cada tarefa possui um diferente consumo de recursos das máquinas, exigindo diferentes parâmetros de configuração. Em sistemas existentes é aplicada a otimização uniforme, onde todas as tarefas da consulta recebem a mesma otimização. O objetivo é realizar a otimização granular utilizando similaridade do uso de recursos na aplicação da otimização.

Simplificando os aspectos relacionados à contribuição pode-se dizer que na implementação da abordagem proposta foi aprimorado a abordagem de autoajuste existente [19] utilizando aprendizado de máquina e análise de log para classificar uma tarefa *MapReduce* no Hadoop. Tarefas classificadas que possuam consumo de recursos computacionais parecidos podem receber a mesma otimização.

1.3 Contribuição

Para que as tarefas no Hadoop sejam ajustadas de forma mais precisa, sugere-se encontrar o consumo real de recursos computacionais que as tarefas exigem do sistema, auxiliando em um melhor ajuste dos parâmetros em futuras execuções. Para tal, propõem-se uma abordagem de classificação de tarefas no Apache Hadoop através das informações de monitoramento de consumo de recursos persistidos em arquivos de log. Realizou-se a análise dos arquivos de log através do uso do algoritmo de classificação não supervisionado K-Means. A abordagem irá gerar como resultado grupos que determinam o consumo de

recursos computacionais que foram utilizados pelas tarefas atribuídas a esses grupos. A hipótese é que grupos de tarefas que possuem o mesmo consumo de recurso computacional podem receber a mesma configuração de parâmetros de desempenho, então, através da abordagem proposta qualquer tarefa do Hadoop submetida para execução e que se assemelhe às tarefas já classificadas a partir de algum mecanismo de análise prévia [11], poderão receber o mesmo ajuste de parâmetros. A Tabela 1.1 mostra os desafios que surgem a partir das contribuições propostas.

Contribuição	**Desafio**
Aplicar a otimização antes da execução da tarefa	Como adaptar a otimização das tarefas antes de sua execução?
Utilizar os arquivos de log para classificar tarefas concluídas	Como otimizar novas tarefas antes de sua execução se a classificação é realizada em tarefas concluídas?
Utilização de algoritmo de classificação para agrupar logs	Como analisar logs com dados não rotulados?
Assumiu-se que tarefas com consumos de recursos similares, podem receber a mesma otimização	Como agrupar tarefas que possam receber a mesma otimização?

Tabela 1.1: Desafios relacionados às contribuições propostas

CAPÍTULO 2

ESTADO DA ARTE

A grande quantidade de dados que emerge diariamente nos servidores das grandes empresas requerem processamento do tipo *OLAP (On-line Analytical Processing)* [7] para extração de conhecimento em diversas atividades de negócio. O *MapReduce* auxilia neste tipo de processamento utilizando uma plataforma distribuída, porém há um grande desafio em manter um bom desempenho neste tipo de sistema. Considerem-se várias tarefas chegando a um *cluster* (máquinas executando o *Hadoop*) onde cada tarefa possui características diferentes como o tamanho dos dados de entrada e o consumo de recursos de memória, disco, rede e processador. A mudança dos parâmetros de configuração afeta diretamente o desempenho da execução das tarefas. Na seção 2.1 aborda-se a arquitetura do *MapReduce*, na seção 2.2 aborda-se os sistemas construídos sobre o *MapReduce*, na seção 2.3 aborda-se alguns sistemas de otimização de desempenho, na seção 2.4 aborda-se a arquitetura do *AutoConf* e na seção 2.5 uma discussão com a finalidade de expôr o problema abordado.

2.1 MapReduce

O paradigma de programação *MapReduce*, desenvolvido em 2004 pela Google Inc. [8] foi projetado para simplificar as tarefas de processamento distribuído que necessitavam de escalabilidade linear. As principais características do sistema são a transparência quanto à programação distribuída, dentre elas se encontram o gerenciamento automático de falhas, de redundância dos dados e de transferência de informações; balanceamento de carga; e a escalabilidade. O *MapReduce* é baseado na estrutura de dados em formato de chave/valor, esse tipo de estrutura vem sendo cada vez mais utilizada entre os novos sistemas de banco

de dados. Entre os sistemas que utilizam chave/valor estão o *HyperDex* (mapeamento multidimensional) [10], o *Dynamo* (uso na *Amazon.com*) [9] e o *Hadoop*.

O *Hadoop* é uma implementação de código aberto do *MapReduce*, hoje gerenciado e repassado à comunidade de desenvolvedores através da Fundação Apache [3]. Existem outros sistemas baseados em *MapReduce*, como por exemplo o *Disco* [22], Spark [4], Hadapt [1] e Impala [6].

2.1.1 Funções *Map* e *Reduce*

Uma tarefa ou *Job* é um programa *MapReduce* sendo executado em um *cluster* de máquinas. Os programas são desenvolvidos utilizando as diretivas *Map* e *Reduce*, que em linguagem de programação são métodos definidos pelo programador. Abaixo é detalhado o modelo de cada primitiva.

- **Map:** Processa cada registro da entrada, gerando uma lista intermediária de pares chave/valor;
- **Reduce:** A partir da lista intermediária, combina os pares que contenham a mesma chave, agrupando os valores;

Para exemplificar o modelo *MapReduce* será utilizado um programa que realiza a contagem da frequência de palavras em um arquivo texto. Na Figura 2.1 seguem os exemplos de função *Map* e *Reduce*:

```
map(String key, String value):
  // key: nome do documento
  // value: conteúdo do documento
  for each word w in value:
    EmitIntermediate(w, 1);
```

```
reduce(String key, Iterator values):
  // key: palavra
  // value: lista de quantidades
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Figura 2.1: Primitivas *Map* e *Reduce* contando a frequência de palavras em um texto

A função *map* recebe por parâmetro uma chave (nome do documento) e um valor (conteúdo do documento), para cada palavra emite um par intermediário onde a chave

é a palavra e o valor é o inteiro 1. A função *reduce* recebe como parâmetro uma chave (a palavra) e o valor (no caso, uma lista de inteiros 1 emitidos por *map*), em seguida a variável *result* recebe a soma dos valores contidos na lista *values* e por fim a função emite o valor total de ocorrências da palavra contida em *key*.

2.1.2 Hadoop

O *Hadoop* é um *framework* de código aberto desenvolvido na linguagem Java, C e Bash. O *framework* disponibiza bibliotecas em Java (.jar) para desenvolvimento de aplicações *MapReduce*, onde o programador poderá criar suas próprias funções *Map* e *Reduce*. Detalhes de instalação e configuração podem ser visualizados em [24].

O *Hadoop* possui o *HDFS (Hadoop Distributed File System)* [29] como sistema de arquivos, sendo responsável pela persistência e consistência dos dados distribuídos, também possui transparência em falhas de rede e replicação dos dados. Para que o *Hadoop* processe informações, essas devem estar contidas em um diretório do *HDFS*. Os dados de entrada são arquivos de texto, que serão interpretados pelo *Hadoop* como pares de chave/valor de acordo com o programa definido pelo desenvolvedor *MapReduce*. Após o processamento da tarefa, a saída é exportada em um ou vários arquivos de texto armazenados no *HDFS* e escritos no formato chave/valor.

O software é dividido em dois módulos, o sistema de arquivos global denominado *HDFS* e o módulo de processamento. Como arquitetura do sistema distribuído há uma máquina que coordena as demais denominada *master*, e as outras máquinas de processamento e armazenamento denominadas *slaves* ou *workers*. Assim, têm-se quatro denominações:

- **Datanode**: cada nodo de armazenamento de dados do *HDFS*;
- **Namenode**: processo que gerencia o *HDFS* e os *Datanodes*;
- **TaskTracker**: cada nodo de processamento de tarefas de *MapReduce*;

- **JobTracker**: processo executado no *master* que gerencia o processamento das tarefas do *MapReduce*, gerencia todos os *TaskTracker*.

A Figura 2.2 mostra a arquitetura do *Hadoop* e seus módulos:

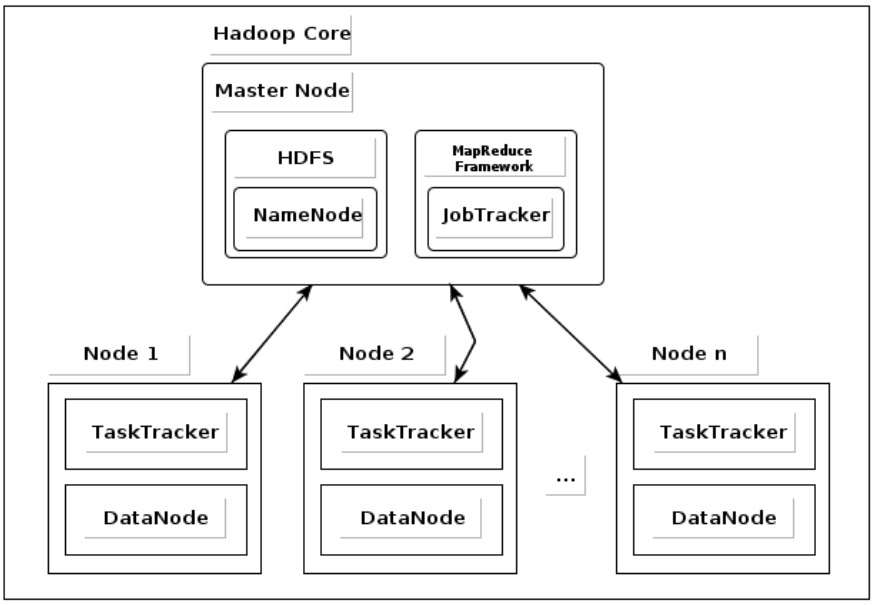


Figura 2.2: Arquitetura do Hadoop e seus módulos

Algumas tarefas importantes executadas pelo nodo *master* são descritas abaixo:

- Controla as estruturas de dados necessárias para a gerência do *cluster*, como o endereço das máquinas de processamento, a lista de tarefas em execução e o estado das mesmas;

- Para cada tarefa de *Map* e *Reduce*, armazena seu estado, podendo ser *idle*(em espera), *in-progress*(sendo processada) ou *completed*(completa);

- Armazena as regiões e tamanhos dos arquivos onde estão localizados os pares intermediários de chave/valor;

- Controla todos os *workers*(máquinas que executam *Datanode* e *Tasktracker*), verificando se estão ativos através de estímulos por *ping*.

2.1.3 Fluxo de Informações

A Figura 2.3 mostra o fluxo de informações no processamento *MapReduce*, usando como exemplo a contagem de palavras. O processamento inicia-se a partir de arquivos armazenados no *HDFS*. Esses arquivos são divididos em partes de acordo com o tamanho definido nos parâmetros de configuração. Uma das máquinas é definida para executar o processo *master*, responsável por atribuir tarefas de *Map* e *Reduce* às demais denominadas *workers*, que por sua vez lêem sua partição de entrada, produzindo as chaves e valores que serão passadas para a função *Map* definida pelo usuário. Pares intermediários de chave e valor são armazendos em *buffer* na memória e periodicamente salvos em disco em regiões de memória que são enviadas ao processo *master*.

Workers responsáveis pelas funções *Reduce* buscam chaves idênticas nos pares intermediários, agrupando-as para processamento. Após a execução da função, é armazenado no sistema de arquivos o resultado final de cada *Reduce*. É interessante ressaltar que é gerado um arquivo para cada tarefa de *Reduce* executada, pois usam-se os arquivos de saída como entrada de outros tarefas de *MapReduce*. Após a finalização da tarefa, o *framework* retorna o resultado ao programa do usuário que requisitou a tarefa.

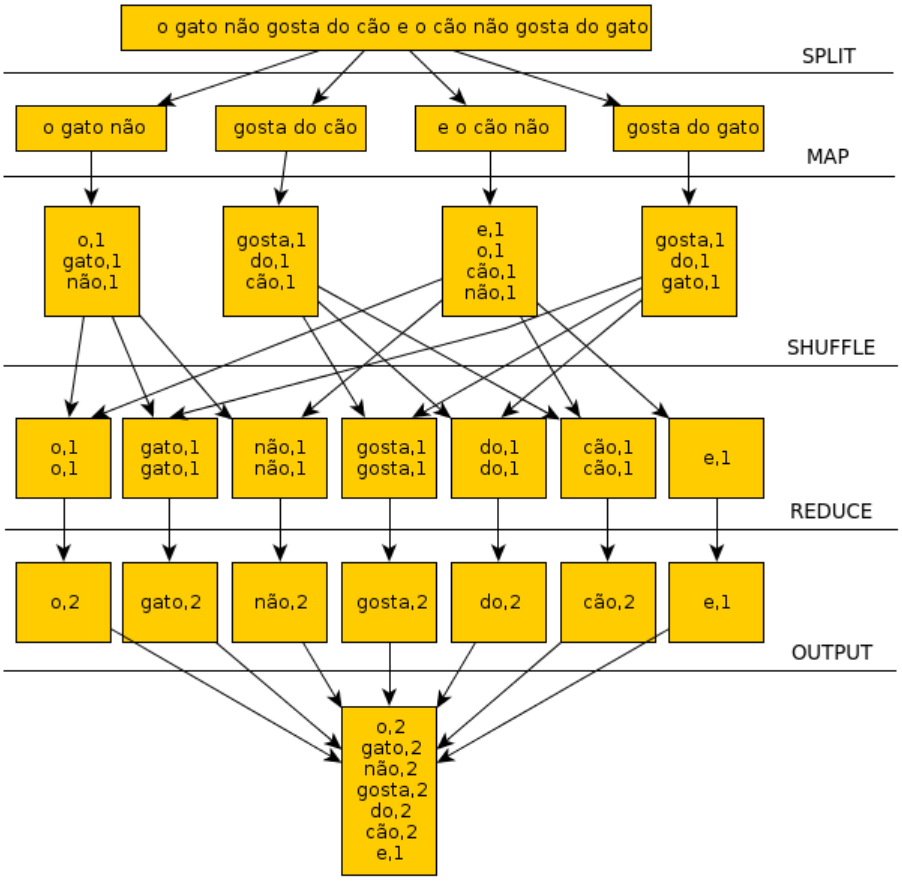


Figura 2.3: Fluxo de informações durante um programa *MapReduce* para contar palavras

Abaixo seguem as fases descritas na Figura 2.3:

- SPLIT - Fase em que a entrada é dividida entre os *workers* disponíveis;
- MAP - Execução das funções *Map*;
- SHUFFLE - Fase em que pode ocorrer o agrupamento das saídas dos *Maps* pela chave, facilitando os trabalhos de *Reduce*;
- REDUCE - Execução das funções *Reduce*;
- OUTPUT - Criação dos dados de saída contendo o resultado da tarefa

2.1.4 Melhorias no Desempenho do Hadoop

O *Hadoop* possui alguns processos que auxiliam no tempo de resposta. Essa seção aborda sobre controle de falhas, armazenamento local e o que são os *Stragglers*. Também aborda-se nesta seção os principais parâmetros de configuração. Seguem algumas melhorias de otimização de desempenho:

- Controle de Falhas - Tarefas de *Map* e *Reduce* são reexecutadas no caso de falha, e todos que encontram-se com tarefas de *Reduce* são informados. Existem *checkpoints* do processo *master* contendo seu estado atual, e caso haja falha do mesmo então poderá ser reiniciado no estado em que se encontrava;

- Armazenamento Local - Dados são armazenados em discos rígidos locais, resultando em menor consumo de banda de rede. Os arquivos de entrada são divididos em partes de tamanho definido e replicados em um número definido de máquinas. Tarefas de *Map* são atribuídas para *workers* que já contenham em seu disco a parte que será utilizada, economizando ainda mais a banda de rede. Tarefas de *Reduce* podem ser atribuídas à máquinas que possuam dados locais produzidos pelos *Maps*; porém, dependerá da disponibilidade das máquinas em questão de *slots* (número de *Maps* e *Reduce* executados concorrentemente na mesma máquina);

- *Stragglers* - *Straggles* são máquinas ativas mas que demoram demais para executar uma tarefa pois possuem algum problema, como disco rígido muito lento. O *MapReduce* resolve esse problema executando tarefas de *backup*. Quando o trabalho está perto do final, as tarefas de *Map* e *Reduce* que estejam em progresso são enviadas para outros *workers*, e assim que algum deles terminar a tarefa é marcada como completa, evitando que as máquinas lentas retardem o término do processo;

2.1.5 Parâmetros de Configuração

O *Hadoop* possui mais de 200 parâmetros de configuração que servem de diretrizes para execução das tarefas, a alteração dos valores desses parâmetros influencia no desempenho do *cluster Hadoop*. Na Tabela 2.1 seguem alguns dos principais parâmetros que quando alterados mudam drasticamente o tempo de resposta da tarefa. Pode-se visualizar detalhes e a lista completa de parâmetros relevantes em [36].

Parâmetro	**Valor Padrão**	**Descrição**
dfs.replication	2	Número de replicação dos blocos
dfs.block.size	64 mb	Tamanho dos blocos em disco
io.sort.mb	100 mb	Tamanho do *buffer* de memória ao ordenar dados
maximum.map.tasks	2	Máximo de tarefas de *map* por *slave* em paralelo
maximum.reduce.tasks	2	Máximo de tarefas de *reduce* por *slave* em paralelo
mapred.job.shuffle.merge.percent	0.66	Porcentagem da memória total alocada para armazenar saídas de *map* em memória

Tabela 2.1: Alguns dos principais parâmetros de configuração do Hadoop

A Figura 2.4 mostra alguns valores de parâmetros determinados no arquivo de configuração do *Hadoop*.

```
1  <configuration>
2    <property>
3        <name>dfs.replication</name>
4        <value>3</value>
5    </property>
6    <property>
7        <name>maximum.map.tasks</name>
8        <value>2</value>
9    </property>
10 </configuration>
```

Figura 2.4: Arquivo de configuração do Hadoop determinando valores de alguns parâmetros

2.2 Sistemas construídos sobre o MapReduce

A necessidade de refazer um programa *MapReduce* para diferentes visões, como no caso de consultas que variam valores e que adicionam ou removem colunas, cria um ambiente de reescrita de código, aumentando os gastos e o tempo com programação. Soluções foram criadas para solucionar esse fato e agilizar a reescrita de código. Sistemas como o *Hive* e o *Pig* [21] possuem uma linguagem declarativa, possibilitando criar uma consulta sobre os dados armazenados no *HDFS*.

O Sistema de Data Warehouse *Hive* possui a linguagem *HiveQL*. Internamente o *Hive* cria um *DAG* (Gráfico Acíclico Dirigido) que contém uma árvore de execução de tarefas *MapReduce* que representa o plano de execução da consulta [19]. Essas tarefas transcrevem os operadores da consulta em programas *MapReduce*, onde cada programa é um estágio da consulta como um todo. A Figura 2.5 mostra um exemplo de consulta *HiveQL*.

```
FROM (SELECT a.status, b.school, b.gender
      FROM status_updates a JOIN profiles b
           ON (a.userid = b.userid and
               a.ds='2009-03-20' )
      ) subq1
INSERT OVERWRITE TABLE gender_summary
                       PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender
INSERT OVERWRITE TABLE school_summary
                       PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1) GROUP BY subq1.school
```

Figura 2.5: Exemplos de consulta *HiveQL* gerada pelo Facebook [5]

Pode-se observar na Figura 2.5 a manipulação de duas tabelas e a execução de uma junção. Após, é populada a tabela *gender-summary* que conterá o resultado do agrupamento pelo atributo *gender*. E por fim, é populada a tabela *school-summary* contendo o resultado do agrupamento pela escola. O *DAG* desta consulta serão programas *MapReduce*, os quais podem receber diferentes valores de parâmetros de desempenho, fazendo com que o ajuste de parâmetros seja realizado a nível de estágio na consulta *HiveQL*.

As Figuras 2.6 e 2.7 mostram parte da consulta 16 do TPC-H e o *DAG* de programas

MapReduce gerado pelo *Hive*, respectivamente.

```
1  INSERT OVERWRITE
2  TABLE q16_parts_supplier_relationship
3  SELECT p_brand, p_type, p_size,
4     COUNT(distinct ps_suppkey) AS supplier_cnt
5  FROM (SELECT *
6     FROM q16_tmp
7     WHERE p_size = 49
8     OR p_size = 14 OR p_size = 23
9     OR p_size = 45 OR p_size = 19
10    OR p_size = 3 OR p_size = 36
11    OR p_size = 9 ) q16_all
12 GROUP by p_brand, p_type, p_size
13 ORDER by supplier_cnt DESC,
14          p_brand, p_type, p_size;
```

Figura 2.6: Parte da consulta 16 do TPC-H [19]

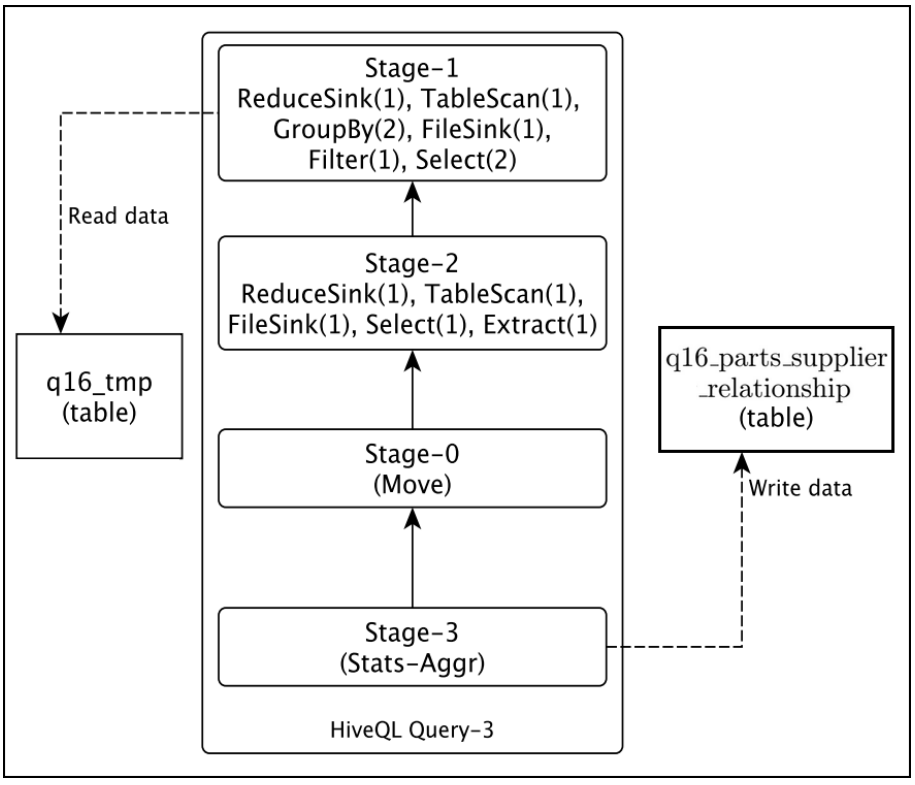


Figura 2.7: *DAG* gerado pelo *Hive* representando a consulta da Figura 2.6 [19]

Na Figura 2.7 as linhas contínuas representam as dependências entre os estágios e as linhas pontilhadas representam operações de leitura e escrita nas tabelas.

Nas seções seguintes são apresentadas diferentes arquiteturas de autoajuste dos parâmetros e de desempenho do *Hadoop*, dentre elas se encontram o *StarFish* [14] (oti-

mização de parâmetros), *Stubby* [17] (análise de fluxo de trabalho) e *AutoConf* [11] (modificação de parâmetros em tempo de execução).

2.3 Abordagens de Ajuste de Parâmetros

O ajuste de parâmetros pode ser aplicado para otimização; porém, pode ser feito de duas formas: pelo administrador do *cluster* ou automaticamente, por uma ferramenta de autoajuste. A segunda alternativa é mais complexa, pois é necessário que as tarefas sejam classificadas e os parâmetros sejam ajustados antes da execução. Os parâmetros geralmente já estão predefinidos no sistema de otimização, como no *Starfish*, mas sua arquitetura apenas gera um perfil da tarefa que foi executada e não aplica o ajuste em tempo de execução. O administrador deverá ajustar esses parâmetros manualmente para que nas próximas tarefas o sistema esteja otimizado.

2.3.1 Geração de Perfis

A geração de perfis consiste em coletar informações sobre a execução de uma tarefa e armazená-las para uso futuro. Essa técnica é importante, pois através de buscas e análises nas informações coletadas é possível ajustar os parâmetros de sistemas *MapReduce* [14].

Starfish é um sistema que foi criado para determinar um perfil às tarefas do *Hadoop* e oferecer ao administrador do sistema melhores configurações de parâmetro. O *Starfish* possui um módulo chamado *Optimizer* que analisa os perfis gerados pelo módulo *Profiler* e sugere melhores valores de parâmetros. Para realizar a coleta de dados das tarefas em tempo de execução o *Starfish* utiliza o BTrace [25].

O *Stubby* por utilizar o *Starfish* internamente possui características semelhantes na geração de perfis [17]; porém, além de gerar perfis para as tarefas também realiza outras otimizações como análise do fluxo de execução. Ao trabalhar com consultas complexas, as consultas são divididas em estágios, onde cada estágio torna-se uma tarefa *MapReduce*.

Nesse caso, as tarefas realizam a execução dos operadores da consulta [11]. Otimizar o fluxo de execução resultará em ganho de desempenho e melhor aproveitamento das máquinas, já que alguns operadores devem ser executados antes que outros.

Um problema nesse tipo de abordagem é a geração de *overhead* para extração do perfil, já que é necessário o monitoramento das máquinas durante a execução da tarefa *MapReduce*.

2.3.2 Análise de Log

Análise de log é uma técnica utilizada para extrair conhecimento sobre um grupo de ações realizadas por algum programa de computador durante um período de tempo, na web por exemplo, geralmente captura-se logs de acesso às páginas [30]. No caso do *Hadoop* os arquivos de logs contém estatísticas de consumo de recursos de máquina, como o uso do processador, disco e memória.

O uso de logs para descoberta de padrões de comportamento é uma opção que auxilia na otimização. O *Hadoop* possui um módulo que gerencia o histórico de execução de suas tarefas chamado *JobHistory*, no decorrer da execução de uma tarefa esse módulo cria logs que são constituídos por dois arquivos, um arquivo de configuração (xml) e outro contendo o histórico de execução detalhado. É a partir desses logs que torna-se possível o uso de um algoritmo de classificação para encontrar um comportamento comum no consumo de recursos computacionais.

As informações mais relevantes a serem extraídas se encontram armazenadas nos contadores gerados pelo *Hadoop*, como por exemplo a quantidade de *bytes* lidos do *HDFS*. A Figura 2.8 mostra alguns exemplos de parâmetros de configuração utilizados pela tarefa, e a Figura 2.9 mostra o histórico de um *Map* contendo um contador, ambas as figuras foram extraídas de arquivos de log.

```
15 <property><name>mapred.job.reuse.jvm.num.tasks</name><value>1</value></property>
16 <property><name>dfs.block.access.token.lifetime</name><value>600</value></property>
17 <property><name>dfs.replication.interval</name><value>3</value></property>
18 <property><name>mapreduce.workflow.name</name>
19 <value>
20 insert overwrite table q20_potential_part_promotion
21 select
22   s_name, s_address
23 from
24   supplier s join nation n
25   on
26     s.s_nationkey = n.n_nationkey
27     and n.n_name = 'CANADA'
28   join q20_tmp4 t4
29   on
30     s.s_suppkey = t4.ps_suppkey
31 order by s_name</value></property>
```

Figura 2.8: Exemplos de parâmetros de configuração armazenados em arquivo de log

Na Figura 2.8 pode-se observar nas linhas 15, 16 e 17 parâmetros contendo valores inteiros, porém, a partir da linha 18 observa-se um parâmetro contendo uma consulta, esse parâmetro é o nome da tarefa. Parâmetros contendo valores alfanuméricos que impossibilitam a conversão do valor para um número decimal não serão utilizados para classificação.

```
 9 Job JOBID="job_201402141153_0001" JOB_STATUS="RUNNING" .
10 Task TASKID="task_201402141153_0001_m_000000" TASK_TYPE="MAP" START_TIME="1392387666530" SPLI
   TS="/default-rack/HalfMoska-Junior" .
11 MapAttempt TASK_TYPE="MAP" TASKID="task_201402141153_0001_m_000000" TASK_ATTEMPT_ID="attempt_
   201402141153_0001_m_000000_0" START_TIME="1392387666537" TRACKER_NAME="tracker_HalfMoska-Juni
   or:localhost/127\.0\.0\.1:33640" HTTP_PORT="50060" LOCALITY="NODE_LOCAL" AVATAAR="VIRGIN" .
12 MapAttempt TASK_TYPE="MAP" TASKID="task_201402141153_0001_m_000000" TASK_ATTEMPT_ID="attempt_
   201402141153_0001_m_000000_0" TASK_STATUS="SUCCESS" FINISH_TIME="1392387668075" HOSTNAME="/de
   fault-rack/HalfMoska-Junior" STATE_STRING="" COUNTERS="{(FileSystemCounters)(FileSystemCounte
   rs)[(HDFS_BYTES_READ)(HDFS_BYTES_READ)(10635)][(FILE_BYTES_WRITTEN)(FILE_BYTES_WRITTEN)(63488
   )]}{(org\.apache\.hadoop\.mapreduce\.lib\.input\.FileInputFormat$Counter)(File Input Format C
   ounters )[(BYTES_READ)(Bytes Read)(10525)]}{(org\.apache\.hadoop\.mapred\.Task$Counter)(Map-R
   educe Framework)[(MAP_OUTPUT_MATERIALIZED_BYTES)(Map output materialized bytes)(8818)][(COMBI
   NE_OUTPUT_RECORDS)(Combine output records)(432)][(MAP_INPUT_RECORDS)(Map input records)(322)]
   [(PHYSICAL_MEMORY_BYTES)(Physical memory \\(bytes\\) snapshot)(191180800)][(SPILLED_RECORDS)(
   Spilled Records)(432)][(MAP_OUTPUT_BYTES)(Map output bytes)(11885)][(CPU_MILLISECONDS)(CPU ti
   me spent \\(ms\\))(370)][(COMMITTED_HEAP_BYTES)(Total committed heap usage \\(bytes\\))(16796
   8768)][(VIRTUAL_MEMORY_BYTES)(Virtual memory \\(bytes\\) snapshot)(1001029632)][(COMBINE_INPU
   T_RECORDS)(Combine input records)(701)][(MAP_OUTPUT_RECORDS)(Map output records)(701)][(SPLIT
   _RAW_BYTES)(SPLIT_RAW_BYTES)(110)]}" .
```

Figura 2.9: Histórico de execução de um *Map* extraído de arquivo de log

Na Figura 2.9, pode-se observar na linha 9 o identificador do *Job* e seu estado. Na linha 10 observa-se uma subtarefa de *Map*, seu identificador e horário de início em milisegundos. Nas linha 11 e 12 observa-se as fases de execução do *Map* e seu histórico, contendo os contadores e as informações a serem extraídas para classificação.

Foram encontradas 63 informações possíveis de participar de uma classificação por um algoritmo nos logs do Hadoop. Porém, concluiu-se que para determinar o workload de um *cluster* Hadoop é necessário classificar as informações de uso de recursos, como memória, disco, rede e cpu. Sendo assim, considera-se essas as mais relevantes.

Sistemas como o Mochi [31] e o Rumen [12] analisam os logs do Hadoop com a finalidade de mostrar ao administrador informações relevantes sobre o ambiente, como tempo total de execução, volume de dados processados e tarefas falhas.

2.4 *AutoConf*

AutoConf [11] é uma ferramenta desenvolvida para otimização de consultas do sistema de *Data Warehouse Apache Hive* [32]. O *Hive* é construído sobre o *Hadoop*, e internamente analisa a consulta e a divide em estágios onde cada estágio é uma tarefa no *Hadoop*. O *AutoConf* realiza a análise dos operadores utilizados em cada estágio da consulta e extrai o que chama-se de **assinatura de código**, por exemplo, se uma consulta possui um operador *TableScan* e um *GroupBy*, então é atribuído a assinatura de código referente a essa estrutura. Cada assinatura possui uma configuração de parâmetros associada, a qual é aplicada em tempo de execução no *Hadoop* antes das tarefas serem executadas. Ou seja, os estágios da consulta serão executados após o autoajuste dos parâmetros, e cada estágio poderá receber valores de parâmetros diferentes de acordo com os operadores da consulta, por exemplo, no caso de um operador que exige leitura de disco, parâmetros associados à otimização de disco serão aplicados [11]. Os grupos de tarefas classificadas a partir das assinaturas de código recebem o nome de **grupos de intenção**. A Figura 2.10 mostra a arquitetura do *AutoConf* e sua integração com o ecossistema do *Hadoop*.

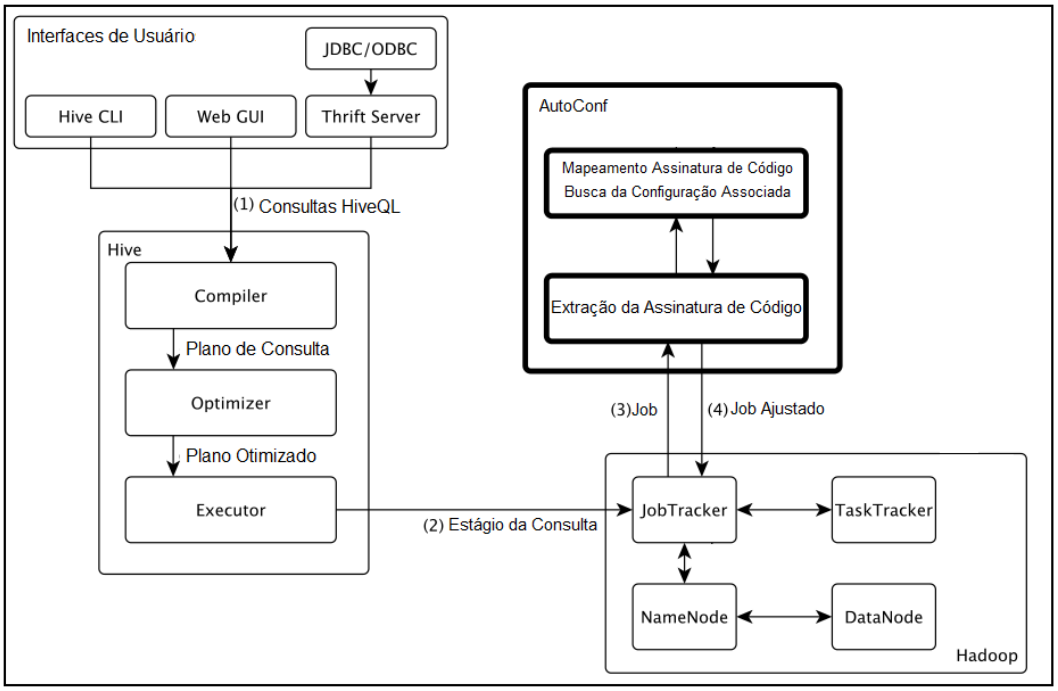


Figura 2.10: AutoConf em meio ao ecossistema do Hive e Hadoop

- Em (1) as Interfaces de Usuário submetem uma consulta *HiveQL*;
- Após o *Hive* gerar e otimizar o plano de consulta, em (2) os estágios da consulta (tarefas) são enviados ao *Hadoop*;
- O *JobTracker* antes de ordenar a execução em (3) envia a consulta ao *AutoConf*;
- Após a extração da assinatura de código e a associação com a configuração correta, em (4) o *AutoConf* envia ao *JobTracker* os valores de parâmetros a serem ajustados;
- Por fim o *JobTracker* ajusta os parâmetros e ordena a execução da tarefa que agora possui as configurações ajustadas.

O *AutoConf* realiza a geração de perfil nas tarefas, pois analisa o texto *HiveQL* submetido e dele extrai os operadores, após, gera a assinatura de código que se pode chamar de perfil do estágio da consulta através do qual está associada uma configuração que será aplicada a tarefa.

A integração desta abordagem de classificação com o *AutoConf* é uma forma de criar um sistema mais robusto, onde é possível que as configurações se autoajustem de acordo

com os recursos computacionais reais utilizados e encontrados após a análise dos logs. E autoajustando as tarefas gerados pelo *Hive*, se está otimizando as consultas *HiveQL* submetidas.

2.5 Discussão

Comparando as tecnologias citadas, pode-se perceber os diferentes tipos de abordagens com a mesma finalidade. O objetivo é identificar funcionalidades que podem ser aprimoradas se forem aplicados a análise de log e algoritmos de classificação. Na Tabela 2.2 observa-se a comparação entre algumas arquiteturas citadas e a abordagem proposta.

Arquitetura	**Descrição**
Starfish	Gera os perfis das tarefas e encontra valores ótimos de parâmetros; porém, o ajuste deve ser manual e a tarefa apenas poderá ser otimizada na segunda vez que for executada. Também gera *overhead*, pois monitora as máquinas durante a execução
Stubby	Utiliza o Starfish, segue os mesmos princípios.
AutoConf	Analisa os operadores das consultas do Hive e classifica as tarefas por regra de acordo com o número de operadores. Aplica o autojuste antes de iniciar a execução da tarefa.
Abordagem Proposta	Aliado ao AutoConf que provê o ajuste automático antes da execução, utiliza algoritmos de classificação não supervisionada para analisar os arquivos de log. Classifica as tarefas de acordo com o consumo de recursos computacionais e aplica valores de parâmetros voltados para a necessidade do *cluster*.

Tabela 2.2: Comparação entre as arquiteturas

Analisando a Tabela 2.2, observa-se que o *Starfish* e o *Stubby* classificam as tarefas durante a execução pelo monitoramento das máquinas, diferente do *AutoConf* que classifica antes da execução analisando o texto *HiveQL*. Ambas as tecnologias não utilizam aprendizado de máquina. Quanto à forma de ajuste de parâmetros, apenas o *AutoConf* realiza automaticamente (ajusta os parâmetros antes da tarefa ser executada, o AutoConf trabalha exclusivamente com as tarefas geradas pelo *Hive*). A abordagem auxilia no autoajuste; porém, também pode ser utilizada em sistemas de ajuste manual. A coleta de informações de logs não é realizada no *AutoConf*, seu objetivo é classificar as tarefas baseado em regras na contagem dos operadores *HiveQL*.

Esta abordagem procura cobrir os pontos fracos de cada arquitetura estudada, disponibilizando o uso de algoritmos de classificação não supervisionados para determinar padrões em tarefas executadas no *Hadoop*. Também auxilia sistemas que classificam as tarefas antes de sua execução, substituindo as regras predefinidas de classificação por uma análise profunda nos arquivos de logs utilizando aprendizado de máquina.

2.5.1 Grupos de Comportamento

O *AutoConf* aborda as tarefas antes da execução utilizando assinaturas de código para classificação e geração de grupos chamados grupos de intenção, citados na seção 2.4. Estendeu-se a abordagem aumentando a representatividade desses grupos, encontrando a realidade do comportamento das tarefas através dos logs, nomeando os novos grupos de **grupos de comportamento**. Os novos grupos serão mais representativos devido aos problemas encontrados no *AutoConf* e no estado da arte, como seguem:

- Os arquivos de log do *Hadoop* contêm dados que não se encontram antes da execução da tarefa, como por exemplo, a quantidade total de bytes lidos e escritos em disco;
- O uso de regras para classificação gera predições de consumo de recursos, predições são menos representativas do que os dados armazenados em arquivos de log;
- Os grupos e perfis criados antes da execução da tarefa são pouco representativos em termos do comportamento real das tarefas, pois utilizam regras.

A melhoria proposta é recriar e ajustar os grupos gerados pelos otimizadores que abordam as tarefas antes de sua execução, por exemplo, as assinaturas de código do *AutoConf*, substituindo por grupos mais consistentes que representam o consumo real de recursos das máquinas, os grupos de comportamento. A abordagem proposta utiliza a análise de logs para encontrar os grupos de comportamento, onde cada recurso (informação encontrada no log) torna-se uma dimensão ao utilizar um algoritmo de aprendizado de máquina. Determinar grupos de comportamento às tarefas analisando os logs é uma forma

de determinar sua configuração. O foco do estudo é encontrar estes grupos e associar uma configuração de parâmetros respectivamente.

CAPÍTULO 3

UMA ABORDAGEM DE CLASSIFICAÇÃO NÃO SUPERVISIONADA DE ANÁLISE DE LOG

Através da classificação de informações sobre o consumo de recursos computacionais de tarefas, busca-se auxiliar o ajuste de parâmetros de desempenho em sistemas de processamento de dados *MapReduce*. A abordagem de classificação proposta neste trabalho é baseada no agrupamento de tarefas *MapReduce* que possuam consumo de recursos computacionais parecidos. Pode-se dizer que os principais consumos de recursos são: ciclos de processador, consumo de memória, banda de rede e vazão de dispositivo de armazenamento. A hipótese é que grupos de tarefas que possuam comportamento similar quanto ao consumo de recursos computacionais podem receber a mesma configuração de parâmetros de desempenho. Os grupos são calculados através do algoritmo de classificação não supervisionado K-Means, que busca encontrar padrões desconhecidos registrados em arquivos de log. A abordagem proposta possui as seguintes vantagens comparado com os trabalhos relacionados:

1. É capaz de ajustar os valores de parâmetros de acordo com um ou mais recursos em específico;
2. Permite uma classificação das tarefas *MapReduce* através do histórico de consumo de recursos;
3. Auxilia no ajuste de parâmetros de sistemas *MapReduce*, como: Hive, Spark e PIG.

Os recursos computacionais utilizados para classificação são dinâmicos, possibilitando que o administrador do sistema que implemente a abordagem possa selecionar os recursos desejados. Como resultado final tem-se um histórico da carga de trabalho do sistema

no período de logs analisado, com essa informação será possível também identificar mudanças na carga de trabalho, facilitando a descoberta de gargalos em determinadas datas e horários. As informações geradas pelo algoritmo de classificação auxiliarão no auto-ajuste de parâmetros *MapReduce*, uma vez que as regras para definição dos valores de parâmetros poderão ser aplicadas sobre esse resultado que equivale a dados reais, e não a uma projeção feita por uma análise de código ou geração de perfil por predição e regra.

Propõe-se uma abordagem genérica, habilitando uma interface de entrada dinâmica com intuito de estender a abordagem além do *Hive* e *Hadoop*. O usuário poderá criar novos formatos de dados de entrada de logs, possibilitando o uso da abordagem em outros e futuros *frameworks* de processamento distribuído *MapReduce*, como o *Pig+Hadoop* [21] e *Spark* [4]. Além da entrada genérica, propõe-se não especificar o uso de um algoritmo, e sim, propor uma abordagem extensiva para qualquer algoritmo de classificação não supervisionado. Realizar a classificação com vários algoritmos pode trazer benefícios no resultado, pois as informações dos logs podem diferenciar de um sistema para outro e o desempenho de cada algoritmo pode ser diferente. A seção 3.1 apresenta os algoritmos de classificação, a seção 3.2 mostra a abordagem proposta e sua implementação com o algoritmo K-Means.

3.1 Algoritmos de Classificação

A classificação é uma das técnica de mineração de dados [27], sua função é classificar os dados de forma a determinar agrupamentos de dados com características comuns. As seções 3.1.1 e 3.1.2 mostram os tipos de classificação e a seção 3.1.2.1 mostra o uso do algoritmo *K-Means* [15] na análise de logs [23].

3.1.1 Classificação Supervisionada

Usa-se classificação supervisionada quando sabe-se a natureza dos dados a serem analisados e sabe-se a qual classes se deseja que as informações sejam classificadas. Os dados

são entregues ao algoritmo já com um rótulo, pode-se dizer que já se encontram com uma classe predeterminada. Um exemplo de algoritmo de classificação supervisionada é o *K-Nearest Neighbor (K-NN)* [35], e um exemplo do *K-NN* em banco de dados é o uso do algoritmo na execução de junções em consultas. O uso desse tipo de algoritmo para efetuar junções é uma forma especial e custosa, e um grupo de pesquisa da Universidade de Singapura mostra uma abordagem de junção com *K-NN* utilizando o *MapReduce* [18]. Pode-se verificar que para junções, os dados já estão preclassificados, pois a junção inicia-se a partir de dois grupos de registros, ou seja, os registros já pertencem a uma classe.

Logs possuem informações estáticas que não serão alteradas em momento algum, com isso é possível utilizar essas informações para autoclassificá-las com um algoritmo de classificação. Porém, os logs são registros não rotulados (i.e., não sabe-se qual a natureza da informação que será encontrada). Não se pode prever qual será o resultado e padrões que serão gerados pelos algoritmos, assim, não se pode predefinir classes aos logs. Como não se conhece os padrões e classes que serão encontrados, utilizou-se classificação não supervisionada.

3.1.2 Classificação não Supervisionada

Quando não se conhece a natureza dos dados e não se sabe que tipos de padrões pode-se encontrar ou se quer buscar, utiliza-se classificação não supervisionada. Apenas os dados puros são passados e é o algoritmo que determina quais os valores dos atributos das classes através da análise dos dados recebidos, sem informar nenhum dado prévio de qual classe pertencem os registros (no caso, os logs). Esse tipo de algoritmo possui um custo de processamento mais elevado, porém, encontra padrões que não se conhecia anteriormente, diferentemente da classificação supervisionada que é necessário um conhecimento prévio desses padrões (i.e., dados rotulados). Exemplos de classificação não supervisionada são os algoritmos *K-Means*, algoritmos de redes neurais [28] e agrupamentos hierárquicos [13]. Na seção 3.1.2.1 apresenta-se o algoritmo *K-Means* e seu uso no tratamento dos logs do *Hadoop* para classificação das tarefas.

3.1.3 K-Means

O algoritmo *K-means* é capaz de classificar as informações de acordo com os próprios dados de entrada, esta classificação é baseada na análise e na comparação entre as distâncias dos registros em valores numéricos a partir dos centróides dos grupos. Por exemplo, considere-se seis grupos, o registro será classificado para o grupo onde a distância do centróide é mais próxima. Desta maneira, o algoritmo automaticamente fornecerá uma classificação sem nenhum conhecimento preexistente dos dados. Nesta seção demonstra-se como são calculados os valores dos centróides dos grupos e as distâncias entre os registros.

O usuário deve fornecer ao algoritmo a quantidade de classes desejadas. Para geração dessas classes e classificação dos registros o algoritmo faz uma comparação entre cada valor de cada linha por meio da distância, onde geralmente é utilizada a distância euclidiana [26] para calcular o quão distante uma ocorrência está da outra. Após o cálculo das distâncias o algoritmo calcula centróides para cada uma das classes. Considere os pontos P e Q, onde:

$P = \{p_1, p_2, ..., p_n\}$ e $Q = \{q_1, q_2, ..., q_n\}$

A equação abaixo representa o cálculo da distância euclidiana entre os pontos P e Q em um espaço n-dimensional:

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Uso com logs: No caso de logs, supõe-se que está se trabalhando com apenas duas dimensões (total de bytes lidos e total de bytes escritos), essas são as dimensões contidas no array P e Q da equação descrita. Supõe-se que P seja o ponto do centróide de uma das classes no espaço bidimensional e Q seja o ponto para um determinado log no espaço bidimensional. Os valores das dimensões do centróide são iguais às médias das dimensões de todos os logs classificados na classe de tal centróide, logo, a fórmula nos trará a distância do log em relação à média de todos da mesma classe.

Cada tipo de informação que um log possui é uma dimensão para o algoritmo. Se

os logs possuem, por exemplo, três tipos de informação, pode-se dizer que o *K-Means* trabalhará com três dimensões. O *K-means* pode ser descrito em cinco passos:

- **Passo 1: Fornecer valores para os centróides:** Neste passo os k centróides devem receber valores iniciais que geralmente são valores aleatórios. Também é importante colocar todos os pontos em um centróide qualquer para que o algoritmo possa iniciar seu processamento.

  Uso com logs: Os centróides possuem as mesmas dimensões dos logs, portanto, são os valores das dimensões dos centróides que determinam o consumo médio de recursos computacionais. No exemplo, a média de bytes lidos e escritos no *HDFS*.

- **Passo 2: Gerar uma matriz de distância entre cada ponto e os centróides:** Neste passo a distância entre cada ponto e os centróides é calculada, é nesse ponto que a maior parte dos cálculos do algoritmo é efetuada, pois se há n pontos e k centróides, então, serão calculadas $n*k$ distâncias.

  Uso com logs: No exemplo de logs de duas dimensões visto anteriormente, nesse ponto é calculada a distância euclidiana exemplificada, lembrando que cada log é um ponto no espaço n-dimensional, serão calculadas (número de logs * número de classes) distâncias.

- **Passo 3: Colocar cada ponto nas classes de acordo com a sua distância do centróide da classe:** Os pontos são classificados de acordo com sua distância dos centróides de cada classe, ou seja, os pontos pertencerão aos centróides que estão mais próximos pela distância calculada. É importante saber que o algoritmo termina se nenhum ponto for classificado a uma classe diferente durante a iteração.

  Uso com logs: Seguindo o exemplo com logs, é nesse ponto que o log será classificado de acordo com a distância entre cada classe. O log será classificado para a classe que possuir o centróide com valores mais próximos. Com isso, se for proposto um ajuste de parâmetros para cada classe de acordo com o consumo de recursos computacionais encontrados, pode-se definir, então, um ajuste de parâmetros igual

para todos os logs que participam da mesma classe.

- **Passo 4: Calcular os novos centróides para cada classe:** Os valores das coordenadas dos centróides são recalculados, para cada classe que possui mais de um ponto o novo valor dos centróides é calculado a partir da média de cada atributo de todos os pontos que pertencem a esta classe.

  Uso com logs: No decorrer do algoritmo, os logs mudam de classe conforme se assimilam ao consumo de recursos computacionais dos centróides. É importante dizer que os valores das dimensões dos logs não se alteram, já que por exemplo, se os bytes lidos de um log foram 1.000.000 ele sempre será 1.000.000, quem terá o valor de bytes alterados são os centróides.

- **Passo 5: Repetir até a convergência:** O algoritmo retorna ao passo 2 repetindo iterativamente o refinamento do cálculo das coordenadas dos centróides.

  Uso com logs: Nesse último passo, se nenhum log sofrer alteração de classe, o algoritmo termina e cada log terá sido classificado de acordo com seu consumo de recursos, podendo então receber um ajuste de parâmetros em uma execução futura ou auxiliar no autoajuste de outras tarefas *MapReduce*.

Dessa maneira, tem-se uma classificação que atribui apenas uma classe a cada ponto (log). No Algoritmo 1, pode-se observar os cinco passos do algoritmo K-Means em um estudo de caso utilizando logs.

No Algoritmo 1, os vetores C e LC armazenam o resultado final da classificação. C contém as classes que representam o consumo dos recursos, onde são representados pelas dimensões dim_x. LC contém a referência para os logs e a qual classe eles pertencem após a classificação. A linha 6 do Algoritmo 1 contém um laço que determina o fim do algoritmo, e entre as linhas 7 e 18 contém as instruções para cálculo das distâncias e dos valores de centróides. A figura 3.1 contém um exemplo de execução do K-Means com duas dimensões (*bytes* lidos e escritos).

Algorithm 1 Algoritmo K-Means em um estudo de caso ao analisar logs

1: k = número de classes desejadas
2: $L = \{l_0, ..., l_n\} \rightarrow$ Conjunto de logs a serem classificados onde $l = \{dim_0, ..., dim_n\} \rightarrow$ Log contendo suas dimensões (e.g., atributos escolhidos para análise)
3: $C = \{c_1, ..., c_k\} \rightarrow$ Centróides (e.g., classes) onde $c = \{dim_0, ..., dim_n\} \rightarrow$ Centróide contendo suas dimensões (e.g, atributos escolhidos para análise)
4: $LC = [\{l_0, c\}, ..., \{l_n, c\}] \rightarrow$ Conjunto relacionando os logs a um centróide
5: $C \leftarrow$ todos os centróides iniciam suas dimensões com valores aleatórios entre o valor mínimo e máximo encontrados entre os logs;
6: **while** Houver alterações de classe entre os logs na última iteração **do**
7: **for** $l_i \leftarrow L$ **do** // Itera os logs
8: $D = \{d_0, ..., d_n\} \rightarrow$ Distâncias entre os centróides e o log
9: **for** $c_x \leftarrow C$ **do** // Itera os centróides
10: $D_x \leftarrow$ cálculo da distância euclidiana entre o log l_i e o centróide c_x;
11: **end for**
12: $LC_i \leftarrow \{l_i, D$ com menor valor (e.g., centróide mais próximo do log)$\}$
13: **end for**
14: **for** $c_i \leftarrow C$ **do** // Recalcula os valores dos centróides
15: **for** $dim_x \leftarrow c_i$ **do** // Itera as dimensões do centróide
16: $dim_x \leftarrow$ Dimensão dim_x do centróide recebe a média entre os valores da dimensão dim_x dos logs que foram classificados para a classe c_i
17: **end for**
18: **end for**
19: **end while**

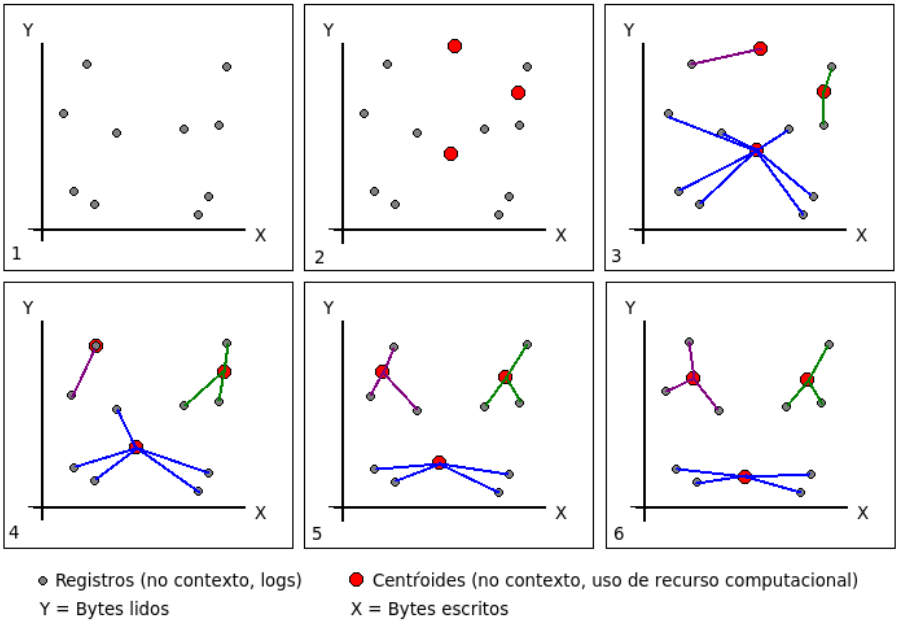


Figura 3.1: Exemplo de execução do K-Means com duas dimensões

Pode-se observar na Figura 3.1 a mudança da posição dos centróides no decorrer dos 6 estágios e a redistribuição dos registros(e.g, logs) nas classes de acordo com suas distâncias euclidianas.

3.1.4 O valor de k no K-Means

As tarefas *MapReduce*, quando executadas no mesmo *cluster* podem consumir quantidades diferentes de recursos entre elas, gerando um espaço de valores entre o mínimo e o máximo consumido. Por exemplo, se as tarefas forem classificadas através do recurso *cpu*, haverão tarefas espalhadas pelos diversos grupos de comportamento ao longo do espaço de valores entre o mínimo e máximo encontrados no uso de *cpu*. O objetivo final é ajustar os valores de parâmetros relacionando com esses grupos que definem o consumo de *cpu*, portanto, quanto maior o número de grupos (valor de k) maior a granularidade obtida. Sendo assim, pode-se obter um ajuste mais fino quando aumenta-se o k, pois assim, se aumenta o número de configurações atreladas aos grupos de comportamento.

3.1.5 Abordagem de Classificação de Logs MapReduce

O objetivo é apresentar uma abordagem de classificação não supervisionada de consumo de recursos das tarefas *MapReduce* no *Hadoop* através da análise de logs. Propõe-se nesse trabalho a possibilidade de uso de diversos algoritmos de classificação na análise dos logs, ou seja, uma abordagem extensível de classificação onde é possível modificar o algoritmo utilizado. Além de possibilitar a entrada dinâmica de logs, habilitando o uso da abordagem para vários sistemas baseados em *MapReduce* que gerem arquivos de log de suas tarefas. Para isso, deve-se superar os seguintes desafios:

1. Informações dos arquivos de logs não são rotuladas;
2. Padrões de consumo de recursos das tarefas *MapReduce* são desconhecidos;
3. O desempenho dos algoritmos é desconhecido, pois não se conhece os dados contidos nos logs.

Até o momento, apenas o *K-Means* e a entrada de logs para o Hadoop foram implementados. Como contribuição propõem-se na arquitetura: o mecanismo de troca de algoritmo e da entrada de logs. Na Figura 3.2 pode-se observar os componentes da abordagem proposta destacados pela linha vermelha pontilhada, em meio ao ecossistema de sistemas baseados em *MapReduce*.

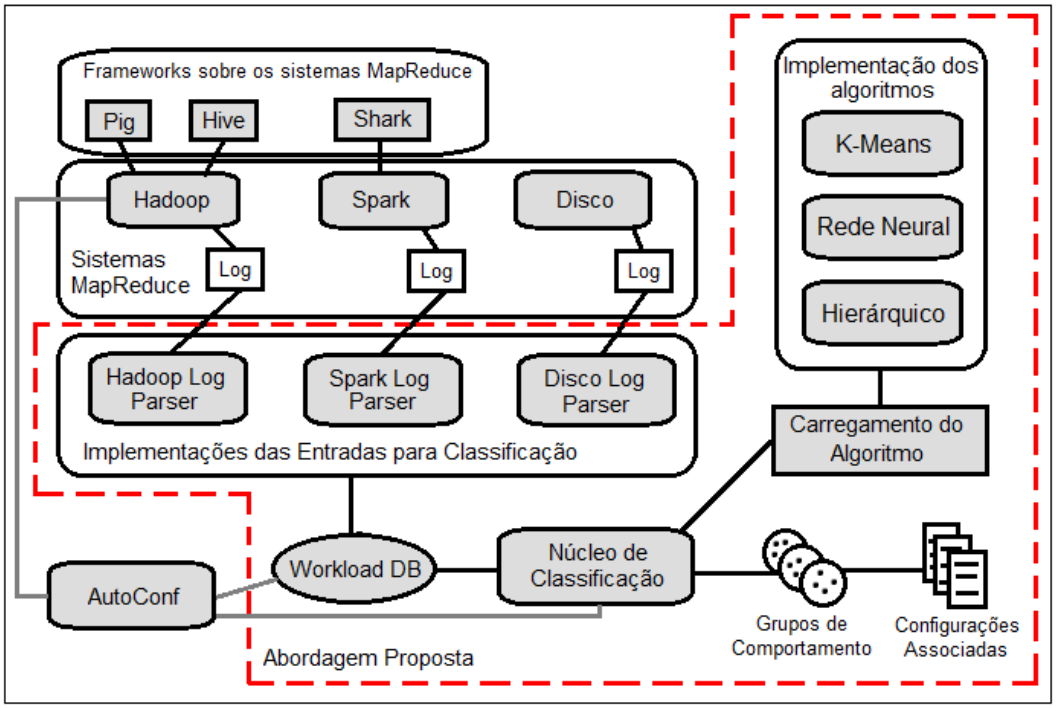


Figura 3.2: Abordagem proposta em meio ao ecossistema de sistemas MapReduce

Cada sistema baseado em *MapReduce* deverá ter sua entrada, estruturando os logs de maneira correta no repositório. O repositório será acessado pelo Núcleo de Classificação, que carregará o algoritmo escolhido através do Carregamento do Algoritmo. Como resultado da classificação serão obtidos os grupos de comportamento e cada grupo terá sua própria configuração de parâmetros. Essa configuração é definida por um módulo a parte ou pelo usuário, pois cada sistema pode possuir parâmetros diferentes. Na Figura 3.2 pode-se visualizar também onde o *AutoConf* se enquadra no ecossistema.

3.2 Chameleon

Na implementação da arquitetura, integrou-se a abordagem proposta com a arquitetura do *AutoConf* e nomeou-se o novo software de *Chameleon* [20]. Essa estratégia irá complementar o *AutoConf* auxiliando na otimização de consultas do *Hive* não somente pela assinatura de código, mas também através da comparação da tarefa com os grupos de comportamento gerados pela abordagem, ajustando os parâmetros de acordo com o consumo de recursos computacionais descritos pelos arquivos de log. O *Chameleon* possui

apenas a implementação do algoritmo K-Means e a entrada de logs para o *Hadoop*. Nessa seção será demonstrado o funcionamento e como as arquiteturas são integradas. A Figura 3.3 mostra a arquitetura do *Chameleon* e onde se enquadra no ecossistema do *Hadoop*.

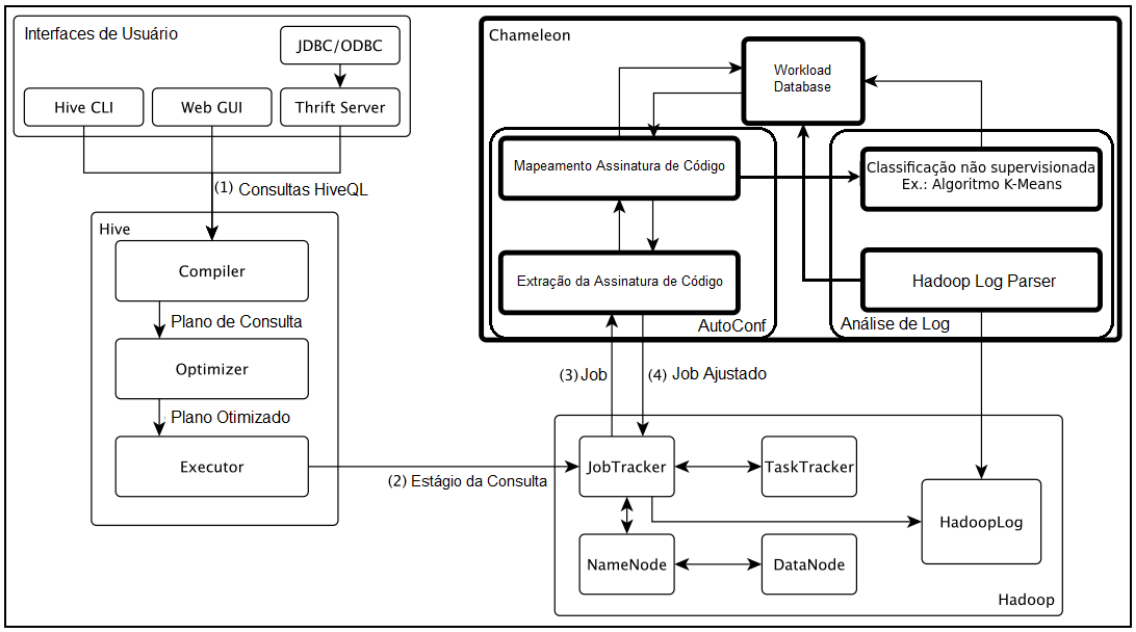


Figura 3.3: *Chameleon* e o ecossistema do *Hadoop*

Como extensão da Figura 2.10, a Figura 3.3 adiciona a Análise de Log ao ecossistema, onde dois módulos distintos trabalham separadamente. O *Hadoop Log Parser* é a implementação dos dados de entrada para o *Hadoop*, mas é possível ser implementado para outros frameworks *MapReduce*. O outro módulo é o algoritmo K-Means implementado. O *Workload Database* é um dos pontos de interação entre as arquiteturas, *O Hadoop Log Parser* usa-o para armazenar os logs que encontra, e por sua vez o *K-Means* busca os logs e salva seus resultados na mesma base de dados.

3.2.1 *Workload Database*

O *Workload Database* ou simplesmente *Workload DB* é uma base de dados relacional contendo os logs do *Hadoop* estruturados de forma que outros módulos possam acessá-los mais facilmente. A base de dados contém informações sobre a execução da tarefa e também quais foram os parâmetros utilizados no momento da execução. A Figura 3.4

mostra o modelo da base de dados.

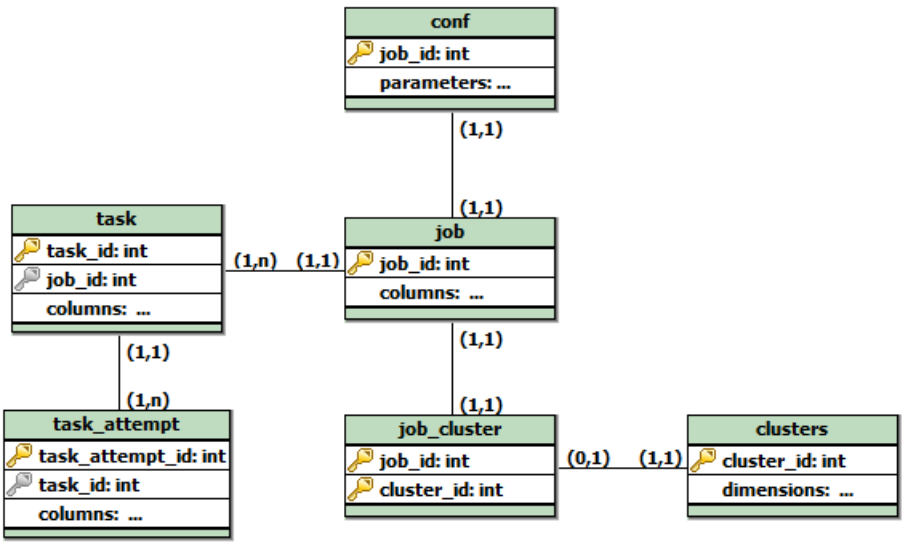


Figura 3.4: Modelo da base de dados do *Workload DB*

Na Figura 3.4 a entidade *job* é a principal, que armazena as informações únicas de cada log das tarefas executadas, e a entidade *conf* armazena as configurações de parâmetros de diferentes tipos que foram utilizadas na execução das tarefas. As entidades *task* e *task-attempt* armazenam o histórico de execução, nelas se pode encontrar os contadores que são utilizados pelo algoritmo de classificação. Após a execução do algoritmo *K-Means*, os valores dos grupos de comportamento são armazenados na entidade *clusters* e por fim, cada log é classificado em um grupo de comportamento através da entidade *job-cluster*.

3.2.2 *Hadoop Log Parser*

Implementação da entrada de logs para o *Chameleon*, tem a função de coletar os logs do *Hadoop*, criar a estrutura de tabelas e popular o *Workload DB* com os logs. Esse módulo pode ser executado no modo *loop*, proporcionando a verificação de novos logs conforme o *Hadoop* processa novas tarefas. Algumas características fazem com que o *LogParser* torne-se um módudo inteligente na coleta e estruturação dos logs, pois além do modo *loop* é possível determinar se os logs serão coletados do histórico do *Hadoop* ou a partir de uma pasta determinada pelo administrador. Outras características são a

verificação de logs já existentes evitando a duplicação; e a exclusão de logs corrompidos ou incompletos, possibilitando uma maior consistência do *Workload DB*.

3.2.3 *Clustering*

Implementação do algoritmo de classificação, o *Clustering* ou Algoritmo K-Means como na Figura 3.3 é o módulo responsável por gerar os chamados grupos de comportamento, que determinam quais os recursos computacionais consumidos pelo *Hadoop* em determinadas tarefas, no caso do *Chameleon* são os recursos usados pelas consultas do *Hive*. O *Clustering* possui diversas configurações que podem ser alteradas pelo administrador. A Figura 3.5 mostra o arquivo de configuração do módulo.

```
<configuration>
  <!— Time among refreshs (min)—>
  <property>
    <name>refresh_time</name>
    <value>5</value>
  </property>
  <!— Number of K-means classes—>
  <property>
    <name>number_of_classes</name>
    <value>12</value>
  </property>
  <!— k-means iterations—->
  <property>
    <name>clustering_execs</name>
    <value>20</value>
  </property>
</configuration>
```

Figura 3.5: Arquivo com algumas configurações do *Clustering*

- **refresh-time**: tempo de espera do *Clustering* para realizar uma classificação dos logs.
- **number-of-classes**: valor de k do algoritmo de classificação ou a quantidade de grupos de comportamento que deseja que os logs sejam agrupados.
- **clustering-execs**: número de execuções do algoritmo para uma mesma classificação, quanto maior valor, mais precisa é a média dos valores encontrados pelo

algoritmo.

Outra característica importante é a possibilidade de escolha das dimensões que deseja utilizar para a classificação. Foram encontradas 63 tipos de informações possíveis de se extrair dos logs do *Hadoop*, e o *Clustering* possui um arquivo de configuração que determina quais delas serão utilizadas, possibilitando que o administrador modele o *Chameleon* de acordo com as necessidades de hardware de seu sistema. O número de informações escolhidas é o número de dimensões que o algoritmo de classificação irá trabalhar.

3.2.4 *Viewer*

Para visualização dos grupos gerados pelo *Clustering* foi desenvolvido o módulo *Viewer*, que se trata de uma *interface web* onde é possível a visualização dos grupos de comportamento gerados pelo *K-Means* e o histórico do consumo de recursos. A Figura 3.6 mostra um exemplo de grupos de comportamento gerados pelo *Chameleon*, a Figura 3.7 mostra um exemplo de dimensões utilizadas com o consumo do recurso computacional encontrado pelo algoritmo e a Figura 3.8 mostra o histórico do consumo de um recurso no decorrer do tempo. Todas as figuras foram extraídas do *Viewer*.

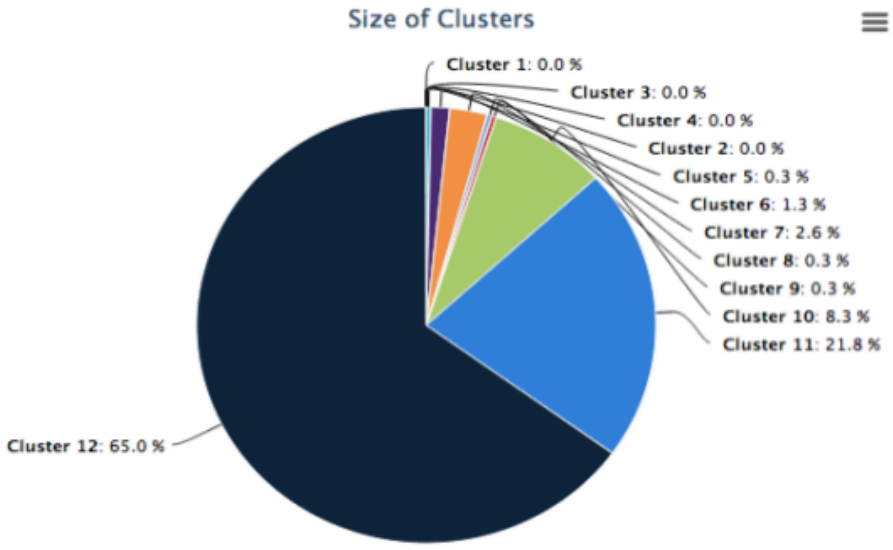
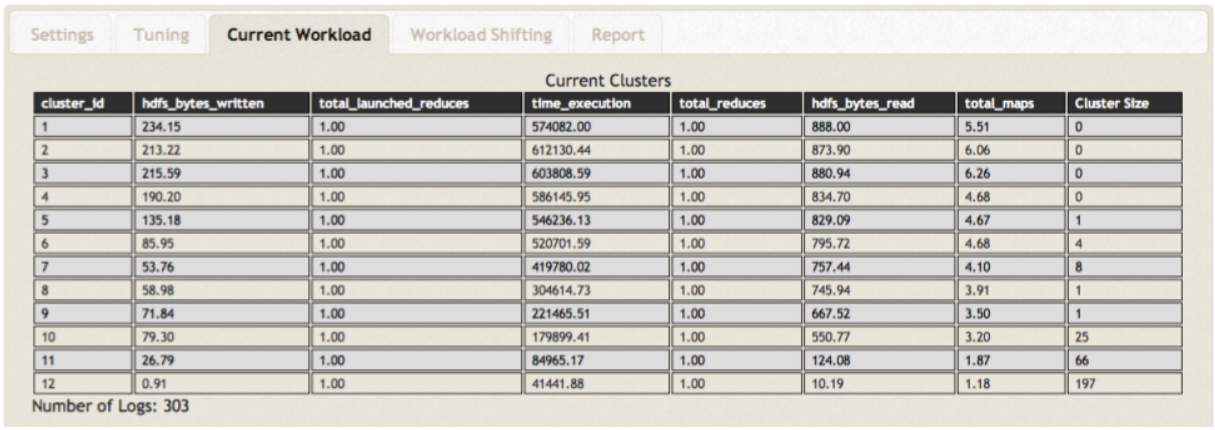


Figura 3.6: Grupos de comportamento gerados pelo *Chameleon*



cluster_id	hdfs_bytes_written	total_launched_reduces	time_execution	total_reduces	hdfs_bytes_read	total_maps	Cluster Size
1	234.15	1.00	574082.00	1.00	888.00	5.51	0
2	213.22	1.00	612130.44	1.00	873.90	6.06	0
3	215.59	1.00	603808.59	1.00	880.94	6.26	0
4	190.20	1.00	586145.95	1.00	834.70	4.68	0
5	135.18	1.00	546236.13	1.00	829.09	4.67	1
6	85.95	1.00	520701.59	1.00	795.72	4.68	4
7	53.76	1.00	419780.02	1.00	757.44	4.10	8
8	58.98	1.00	304614.73	1.00	745.94	3.91	1
9	71.84	1.00	221465.51	1.00	667.52	3.50	1
10	79.30	1.00	179899.41	1.00	550.77	3.20	25
11	26.79	1.00	84965.17	1.00	124.08	1.87	66
12	0.91	1.00	41441.88	1.00	10.19	1.18	197

Number of Logs: 303

Figura 3.7: Grupos de comportamento com as dimensões utilizadas

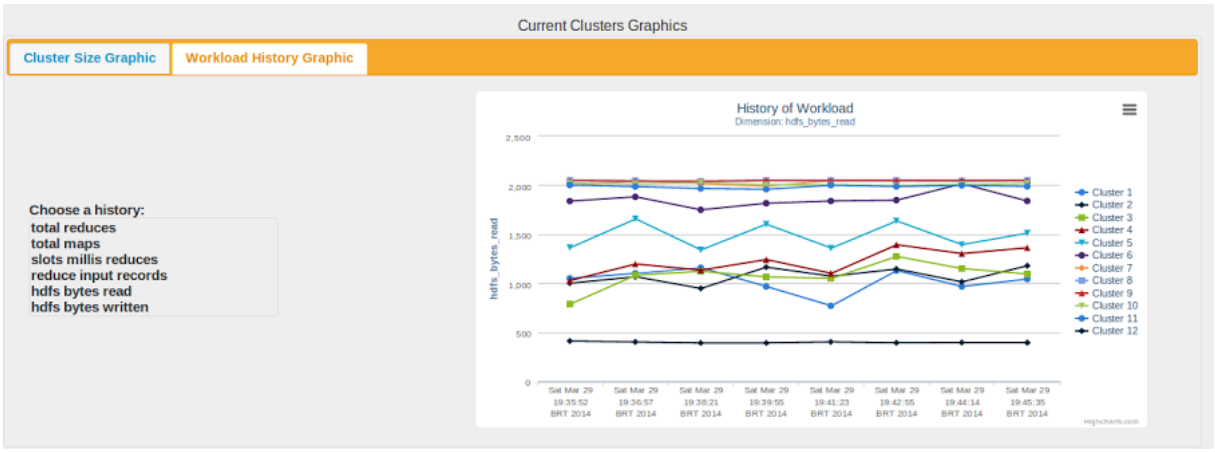


Figura 3.8: Histórico de bytes lidos durante 10 minutos de execução do *Chameleon*

3.2.5 Integração com o *AutoConf*

Tendo como base o funcionamento dos módulos e o objetivo do *Chameleon* na otimização das consultas do *Hive*, fez-se necessário a criação de uma integração entre o *AutoConf* e o *Clustering*. A conexão foi realizada após a identificação da assinatura de código e antes da aplicação da nova configuração. Nesse instante, o *AutoConf* realiza uma chamada remota a um método do *Clustering*, o qual é responsável por determinar qual será a configuração que deverá ser aplicada a partir dos grupos de comportamento e não mais a partir dos grupos de intenção gerados a partir da assinatura de código. O método pode ser visualizado no Algoritmo 2, que mostra o processo de ajuste utilizando os grupos de comportamento.

Algorithm 2 Algoritmo para aplicação da Configuração a partir dos grupos de comportamento

```
    I = {j_0, ..., j_n} → Tarefas com assinaturas de códigos idênticas
2:  C = {c_0, ..., c_n} → Contém a quantidade de tarefas em cada grupo de comportamento,
    todos os valores se iniciam em 0
    procedure GETTUNING(I)
4:      for j_i ← I do
            x ← extrai o índice do grupo de comportamento de j_i
6:          c_x ← c_x + 1
        end for
8:      m ←busca o índice do grupo de comportamento com maior valor em C
        return m
10: end procedure
```

Antes de se explicar detalhadamente, é necessário saber que o *Clustering* gerencia duas novas tabelas no *Workload DB*, uma das tabelas contém os grupos de comportamento atuais e seus valores de dimensões, a outra possui o identificador dos logs já processados e qual é o grupo de comportamento que ele pertence.

No Algoritmo 2, ao chamar o método remoto o *AutoConf* envia ao método um histórico de tarefas (I) que utilizaram a mesma configuração da tarefa a ser ajustada, ou seja, que possuam a mesma assinatura de código (grupo de intenção). Como essas tarefas de histórico já foram completadas, possuem logs que estarão armazenados no *Workload DB*. Então, o *Clustering* gera uma lista chave/valor (C) onde a chave é o identificador do grupo de comportamento e o valor é a quantidade de logs que foram enviados pelo *AutoConf* e que estão relacionados ao grupo. Esses dados são adquiridos ao analisar as duas tabelas citadas e pode-se visualizar a geração da lista no laço entre as linhas 4 e 7 do Algoritmo 2.

O retorno do método remoto é o identificador do grupo de comportamento o qual a tarefa a ser ajustada se enquadra, ou seja, o grupo que mais obteve ocorrências de tarefas vindas do grupo de intenção (linha 8 e 9 do Algoritmo 2). Em seguida, o *AutoConf* aplica a configuração relacionada ao grupo de comportamento encontrado.

3.2.6 Visão Geral sobre o Autoajuste do Chameleon

O autoajuste possui duas fases. Na primeira, ocorre a extração dos operadores da consulta do *Hive* e a partir da assinatura de código é gerado o grupo de intenção. Na segunda fase, através da classificação são gerados os grupos de comportamento que determinam os recursos reais consumidos pelas tarefas. Através da comparação entre os grupos de intenção da tarefa a ser ajustada e os grupos de comportamento, é determinada uma configuração melhor baseada nos recursos computacionais utilizados que foram encontrados pelo algoritmo K-Means ao analisar o histórico de tarefas já concluídas (logs).

CAPÍTULO 4

EXPERIMENTOS

Esta seção é dividida em duas partes. Na seção 4.1, apresenta-se os resultados preliminares que mostram o algoritmo K-Means classificando logs do *Hadoop*. O objetivo destes testes iniciais é verificar o comportamento do algoritmo em relação aos logs do *Hadoop* e verificar o uso das dimensões(recursos) corretas para classificação no *Chameleon*. A seção 4.2 apresenta os experimentos finais onde comparou-se o desempenho do *Hive* e o *AutoConf* contra o *Chameleon*.

4.1 Validação do K-Means

Esta seção mostra os experimentos realizados para validação do uso do K-Means no presente estudo. O objetivo do experimento é verificar o comportamento do algoritmo K-Means ao modificar as dimensões e variar o número de classes utilizando a mesma carga de arquivos de log. A seção 4.1.1 mostra os primeiros experimentos e seus resultados, a seção 4.1.2 mostra experimentos variando as dimensões e o valor de k. Nos experimentos foram extraídos 1505 logs de tarefas executadas em um *cluster* com 4 GB de memória RAM por máquina, 10 máquinas com 2 processadores Intel Core 2 Duo 3.00 GHz cada e discos rígidos de 7200 rpm SATA com 500 GB. Os logs foram gerados através da execução do benchmark TPC-H [33] adaptado para *HiveQL*.

4.1.1 Variação do valor de k

Foram determinadas quatro dimensões para classificação das tarefas: Número de Reduces; Tempo total de execução; Número de Maps; Total de dados de entrada nos Maps

(MB). Os experimentos foram realizados com 8, 12 e 16 grupos de comportamento. A tabela 4.1 demonstra os resultados obtidos com 12 grupos.

Grupo	**Número de Maps**	**Número de Reduces**	**Tempo de Execução (seg)**	**Dados de Entrada nos Maps (MB)**	**Tamanho do Grupo**	**Porcentagem do Grupo**
1	166	44	504,5	2044,4	5	0,33 %
2	170	45	543,8	2041,3	7	0,47 %
3	159	40	428,2	2042,2	11	0,73 %
4	148	35	336,5	2032,2	18	1,20 %
5	155	39	340,9	2040,4	24	1,60 %
6	150	37	331,3	2036,9	27	1,80 %
7	148	39	348,4	2034.0	31	2,07 %
8	117	30	322,9	1965,7	35	2,33 %
9	44	10	169,6	1772,4	52	3,47 %
10	30	7	103,5	1688,1	93	6,20 %
11	20	5	58,7	1314,9	208	13,88 %
12	3	1	16,3	76,2	988	65,91 %

Tabela 4.1: Experimento 1 - Grupos do Algoritmo K-Means no agrupamento de Logs do Hadoop com 12 grupos

Analisando a Tabela 4.1, verifica-se que a maioria dos logs foram classificados apenas em um grupo, mesmo após modificar o número de classes. A predominância foi de tarefas com o tamanho dos dados de entrada menores, mostrando que tarefas que exigem mais recursos não foram comuns, pois quanto menor o dado de entrada, menos recursos serão consumidos.

4.1.2 Variação dos dados de entrada

Foram realizados novos experimentos onde foi incluída a dimensão *Output* em MB e alguns logs foram filtrados pois possuiam zero *bytes* de entrada (identificou-se que logs de tarefas com erro possuiam essa característica). A classificação foi feita iterando entre 4 e 32 grupos em um total de 932 logs. As Tabelas 4.2 e 4.3 demonstram os resultados obtidos com 12 e 23 grupos de comportamento respectivamente.

Analisando os novos testes, verificou-se que a predominância de apenas um grupo

Grupo	Número de Maps	Número de Reduces	Tempo de Execução (seg)	Dados de Entrada nos Maps (MB)	Dados de Saída nos Maps (MB)	Porcentagem do Grupo
1	237	63	614	1957,59	1937,10	0,34 %
2	162	43	544	2030,51	1971,29	0,68 %
3	154	41	528	2041,04	1952,04	1,13 %
4	152	40	481	2036,33	1923,76	1,70 %
5	150	40	400	2032,14	1800,84	2,38 %
6	149	39	355	2027,32	1803,18	3,28 %
7	145	38	337	2024,26	1817,67	4,08 %
8	129	34	329	1990,94	1898,39	5,21 %
9	94	25	277	1900,87	1865,75	6,80 %
10	48	12	158	1648,10	1474,78	11,33 %
11	25	6	81	1569,89	1040,45	23,67 %
12	10	2	36	643,32	284,60	39,41 %

Tabela 4.2: Experimento 2 - Grupos do Algoritmo K-Means no agrupamento de Logs do Hadoop com 12 grupos

diminuiu; porém, ainda continua. Como a carga de logs é a mesma, apenas com a diferença da implementação do filtro de logs de tarefas falhas, os logs predominantes são com dados menores de entrada e saída dos *Maps*. Provavelmente esse fenômeno ocorre por conta das tarefas executadas que provêm do benchmark TPC-H adaptado para *HiveQL*.

Comparando os resultados obtidos com os experimentos realizados pela Yahoo [2], verificou-se que um determinado grupo predominou em ambos os experimentos, formando um grupo com mais de 60 por cento dos logs no experimento inicial. Depois de adicionar uma dimensão, conforme o aumento do número de k os grupos tornam-se menores; porém, a predominância continua no grupo com tarefas com menor consumo de recursos em relação aos dados de entrada dos *Maps*. Ao inserir uma nova dimensão (Saída dos Maps em MB), pode-se perceber uma diminuição na predominância do maior grupo, mas também nesse caso, a predominância continua. Deve-se considerar que os logs utilizados são de experimentos realizados repetitivamente, simulando o processamento de uma mesma consulta várias vezes com cargas de dados diferentes através do TPC-H. Pode-se validar o uso do K-Means por possuir resultados similares aos do Yahoo [2].

Grupo	**Número de Maps**	**Número de Reduces**	**Tempo de Execução (seg)**	**Dados de Entrada nos Maps (MB)**	**Dados de Saída nos Maps (MB)**	**Porcentagem do Grupo**
1	228	60	610	1554,98	1527,73	0,00 %
2	210	53	588	1764,33	1727,43	0,11 %
3	191	51	553	1988,36	1939,64	0,22 %
4	172	45	563	2026,01	1968,26	0,33 %
5	157	41	510	2038,80	1888,48	0,33 %
6	153	40	492	2043,47	1889,98	0,56 %
7	152	40	471	2038,15	1827,12	0,67 %
8	154	40	480	2036,05	1891,13	0,78 %
9	150	40	445	2025,62	1830,58	1,00 %
10	152	40	420	2029,12	1824,43	1,11 %
11	148	39	364	2020,05	1757,70	1,45 %
12	153	40	367	2030,08	1826,06	1,67 %
13	142	38	324	2011,57	1806,09	2,01 %
14	141	37	335	2013,80	1855,74	2,34 %
15	130	34	318	1977,89	1862,54	2,68 %
16	119	31	318	1968,63	1927,39	3,12 %
17	98	26	292	1919,84	1907,41	3,79 %
18	81	21	242	1882,47	1796,52	4,46 %
19	46	12	157	1726,49	1567,24	6,24 %
20	28	7	94	1494,13	1102,99	10,14 %
21	23	6	71	1430,48	872,93	14,16 %
22	15	4	48	914,23	534,72	18,28 %
23	9	2	34	622,14	216,65	24,53 %

Tabela 4.3: Experimento 2 - Grupos do Algoritmo K-Means no agrupamento de Logs do Hadoop com 23 grupos

4.2 Hadoop Padrão x Regras de Ouro x Chameleon

Essa seção tem a finalidade de mostrar os resultados dos experimentos realizados que demonstram o ganho de desempenho do *Chameleon* em relação ao *Hadoop* e às regras de ouro utilizadas pelo *AutoConf*. Como o *AutoConf* utiliza valores de parâmetros determinados por regras encontradas no estado da arte [36], nomeou-se os testes realizados com o *AutoConf* de Regras de Ouro. A seção é dividida em cinco partes, a seção 4.2.1 mostra a metodologia e o ambiente utilizado nos experimentos. A seção 4.2.2 discute o uso dos recursos como cpu e disco que foram utilizados nos experimentos, também discute o valor de k utilizado. A seção 4.2.3 discute sobre a geração dos valores dos parâmetros de configuração utilizados nos experimentos. A seção 4.2.4 mostra os resultados dos experimentos realizados e a discussão entre as três arquiteturas. E a seção 4.2.5 define os trabalhos futuros.

4.2.1 Metodologia e Ambiente dos experimentos

O objetivo dos experimentos é a geração de resultados de tempo de execução das vinte e duas consultas do TPC-H. Para determinar os valores de parâmetros para os grupos de comportamento do *Chameleon* utilizou-se o *Starfish*. No final dos experimentos, deve-se obter as seguintes informações:

- Tempo total e individual de execução das vinte e duas consultas do TPC-H das arquiteturas do *Hadoop* Padrão, *Hadoop*+Regras de Ouro e *Chameleon*;
- Número de consultas que cada uma das arquiteturas venceram.

O *Chameleon* foi desenvolvido para otimizar o *Hadoop* 1.2.1 e *Hive* 0.11, já o *Starfish* otimiza o *Hadoop* 0.20.x e *Hive* 0.7.x. Para solucionar essa diferença de versão entre as arquiteturas, foram preparados dois *clusters* de máquinas virtuais com as mesmas configurações físicas e virtuais; porém, em um deles foi configurado um ambiente para o *Starfish* e no outro um ambiente para o *Chameleon*. Nomeou-se os ambientes pelo nome

do software: *Chameleon* e *Starfish*. Todos os experimentos foram executados no ambiente do *Chameleon*, e o ambiente do *Starfish* foi utilizado apenas para geração dos valores de parâmetros relacionados aos grupos de comportamento.

Os ambientes possuíam as seguintes características:

- Máquina Física: Processador Core I7 930 2.8GHz com 4 núcleos e 8 threads. 9 GB de memória RAM DDR3 1600MHz. Três discos rígidos de 7200rpm em RAID 0;
- Dois *clusters* de máquinas virtuais, cada um com quatro máquinas sendo um *master* e três *slaves*. Cada máquina utilizando 2 núcleos de processamento, 2 GB de memória RAM e 30GB de disco.

Para execução dos experimentos no ambiente do *Chameleon*, fez-se necessário a execução prévia do TPC-H para obter grupos de comportamento consistentes. Foram executadas seis vezes as vinte e duas consultas do TPC-H para *Hive* com entrada de 1GB e 10GB sequencialmente. A tabela 4.4 mostra como os grupos de comportamento ficaram dispostos após a classificação pelo K-Means de 1970 logs resultantes da execução prévia.

Percebe-se pela Tabela 4.4 que a predominância continua nas tarefas com menor consumo de *cpu*.

4.2.2 Escolha das dimensões (recursos) e do valor de k

Para os experimentos, utilizou-se a análise dos seguintes recursos: memória, disco e cpu. Concluiu-se que para se utilizar vários recursos na mesma classificação é necessário que todos os valores estejam na mesma escala, caso contrário, uma das dimensões do algoritmo irá influenciar no resultado da classificação, gerando resultados não satisfatórios. No caso dos três recursos analisados têm-se as seguintes escalas:

- Memória: *Snapshot* em bytes;
- Disco: Leitura e Escrita em bytes;

Grupo	**CPU em milisegundos**	**Tamanho do Grupo (logs)**	**Porcentagem do Grupo**
1	1707960.03	0	0 %
2	1752088.63	0	0 %
3	1712581.82	0	0 %
4	1801939.29	1	0,0508 %
5	1677675.25	0	0 %
6	1648002.99	2	0,1015 %
7	1573726.43	0	0 %
8	1456164.70	3	0,1523 %
9	1423961.70	0	0 %
10	1305171.82	5	0,2538 %
11	1250740.56	0	0 %
12	1205619.45	2	0,1015 %
13	1024232.01	1	0,0508 %
14	1002581.58	7	0,3553 %
15	874972.82	0	0 %
16	862693.81	23	1,1675 %
17	816546.49	2	0,1015 %
18	816538.25	0	0 %
19	733438.98	2	0,1015 %
20	730841.42	14	0,7106 %
21	683629.27	21	1,0660 %
22	714580.05	9	0,4568 %
23	730434.42	3	0,1523 %
24	576725.84	39	1,9797 %
25	487231.49	51	2,5888 %
26	241430.37	45	2,2843 %
27	129824.50	146	7,4112 %
28	58475.52	248	12,5889 %
29	7749.98	1346	68,3249 %

Tabela 4.4: Disposição dos grupos de comportamento na execução prévia dos experimentos

- Cpu: Tempo de cpu em milisegundos.

Memória e disco estão na mesma escala; porém, identificou-se que os valores de uso de memória são de um *snapshot* (soma dos valores totais de memória utilizada nas máquinas do *cluster*), o que faz com que o valor mínimo não seja zero já que a memória utilizada pelo sistema operacional está incluída no valor. Decidiu-se executar os experimentos voltados para apenas um recurso (apenas uma dimensão). Optou-se por executar os experimentos com o recurso *cpu*.

Quanto ao valor de k, optou-se executar os experimentos com 29 grupos de comportamento. Quanto maior o número de grupos maior a granularidade obtida, e como o *AutoConf* possui 29 assinaturas de código diferentes (29 grupos de intenção), decidiu-se determinar o mesmo número de grupos de comportamento ao *Chameleon*.

4.2.3 Geração dos valores de parâmetros

Utilizou-se o *Starfish* para gerar os valores de parâmetros no *Chameleon*. Cada grupo de comportamento deve possuir uma configuração associada, portanto, utilizou-se a seguinte técnica para gerar os valores:

1. Executar diferentes tarefas *MapReduce* com entradas de diferentes tamanhos no *Chameleon*;
2. Anotar os grupos de comportamento em que as tarefas são agrupadas;
3. Executar as mesmas tarefas com a mesma entrada no ambiente do *Starfish*;
4. Anotar os valores de parâmetros sugeridos pelo *Starfish* para cada tarefa;
5. Associar os valores de parâmetros nos grupos de comportamento.

A tabela 4.5 mostra um exemplo de valores sugeridos pelo *Starfish* ao executar a tarefa *WordCount* com uma entrada de 100 MB.

Parâmetro	**Valor Sugerido**
io.sort.mb	55
mapred.reduce.tasks	4
io.sort.factor	53
mapred.job.shuffle.input.buffer.percent	0.2924902042035873
min.num.spills.for.combine	3
io.sort.spill.percent	0.5057433758063934
io.sort.record.percent	0.011872191689591442
starfish.use.combiner	true
mapred.job.shuffle.merge.percent	0.6864921606424759
mapred.inmem.merge.threshold	21
mapred.output.compress	true
mapred.compress.map.output	true
mapred.job.reduce.input.buffer.percent	0.6921695681661945

Tabela 4.5: Exemplo de valores de parâmetros sugeridos pelo Starfish na execução do *WordCount* com 100 MB de entrada

Os valores da Tabela 4.5 foram atribuídos ao grupo de comportamento 27 da Tabela 4.4.

4.2.4 Resultados

Cada teste foi executado três vezes e o resultado final é a média das execuções. O gráfico da Figura 4.1 mostra o tempo de execução total das 22 consultas do TPC-H executada em sequência. O gráfico da Figura 4.2 mostra o tempo de execução de cada consulta. E o gráfico da Figura 4.3 mostra a porcentagem de consultas em que cada arquitetura teve o menor tempo de execução.

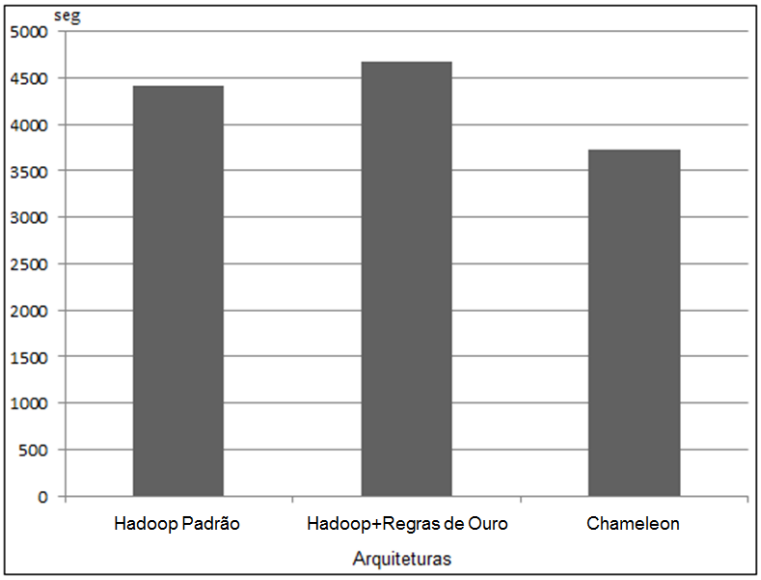


Figura 4.1: Tempo total de execução (segundos) das 22 consultas do TPC-H

Pode-se observar na Figura 4.1 o melhor desempenho do *Chameleon* em relação ao *Hadoop* Padrão ao executar todas as consultas do TPC-H. Porém, as Regras de Ouro não foram satisfatórias em relação ao *Hadoop* Padrão, a hipótese é que o *Hive* também possui otimizações internas, que demonstraram-se mais satisfatórias que as Regras de Ouro.

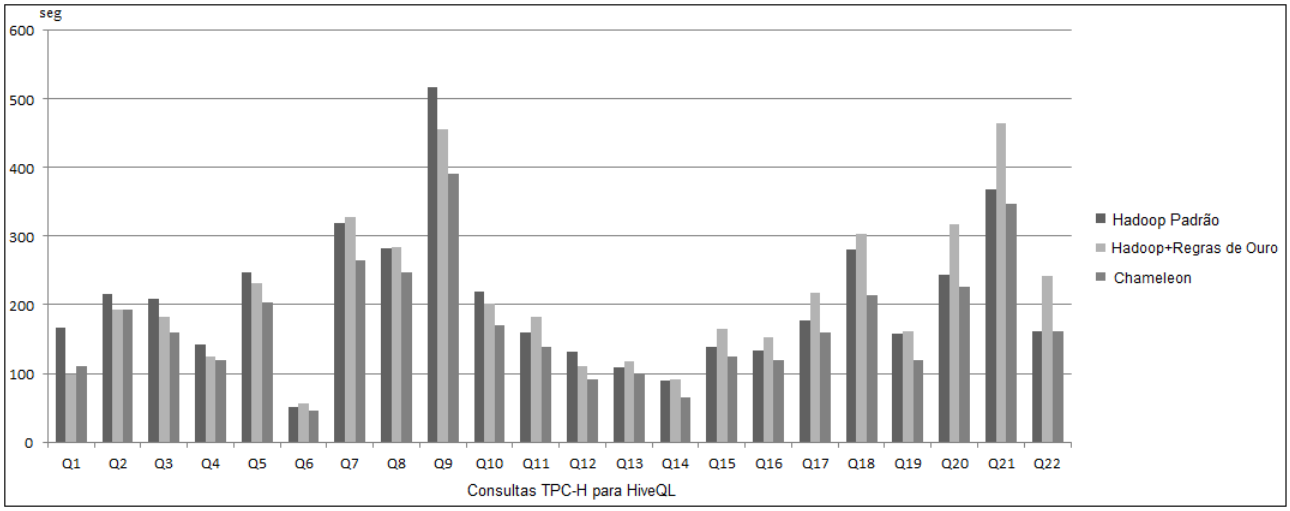


Figura 4.2: Tempo de execução (segundos) das 22 consultas do TPC-H individualmente

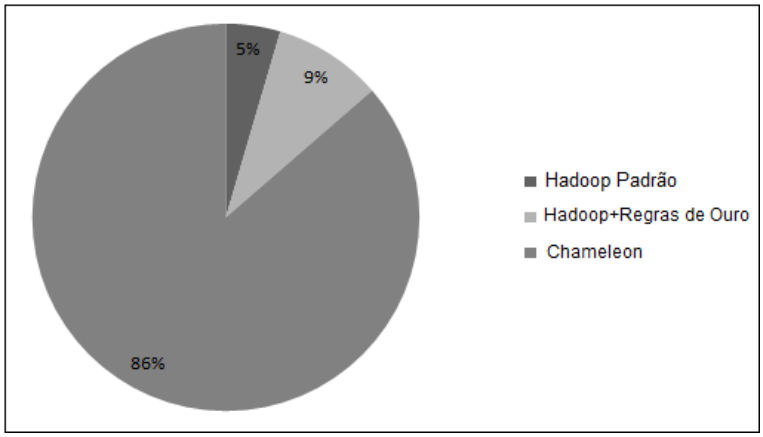


Figura 4.3: Porcentagem de consultas que cada arquitetura se demonstrou mais rápida

Pode-se observar nas Figuras 4.2 e 4.3 que o *Chameleon* teve um desempenho melhor em 86% das consultas em relação ao *Hadoop* Padrão. Pode-se concluir que para o recurso *cpu* o *Chameleon* demonstra-se satisfatório ao otimizar consultas no *Data Warehouse Hive*.

4.2.5 Trabalhos Futuros

Este trabalho deixa como trabalhos futuros os seguintes itens:

- Analisar o comportamento da classificação com outros recursos, como memória, disco e rede. Analisar também o possível uso de recursos em conjunto. É viável a classificação mesclando recursos?
- Implementar e analisar o comportamento de outros algoritmos de classificação, como redes neurais. O K-Means obteve resultados satisfatórios, mas qual o desempenho se utilizado outros algoritmos?
- O *Chameleon* utiliza valores de parâmetros para os grupos de comportamento manualmente. Nos experimentos foram utilizados valores sugeridos pelo *Starfish*. A implementação de um módulo que gere os valores automaticamente está em andamento; porém, o estudo de outras arquiteturas para geração dos valores de parâmetros é uma forma de otimizar ainda mais o *Chameleon*. Como gerar valores de parâmetros mais eficientes que o *Starfish*?

- Banco de dados em memória é um eficiente método de melhorar o desempenho. Não diferindo em datawarehouse, a implementação da abordagem do *Chameleon* em outra plataforma *MapReduce* que utiliza a memória como armazenamento faria com que o desempenho aumentasse ainda mais. Quais as mudanças e desafios para implementar o *Chameleon* no *Spark*?

CAPÍTULO 5

CONCLUSÃO

Pode-se concluir através dos experimentos realizados com o *K-Means*, que o uso de classificação não supervisionada para encontrar o comportamento das tarefas do *Hadoop* em relação ao consumo de recursos computacionais, é viável. Relacionar tarefas do *Hadoop* a uma configuração de parâmetros de desempenho também se mostra satisfatório ao utilizar o algoritmo *K-Means*. A experiência na implementação do *Chameleon* demonstrou-se eficiente em ajustar os grupos de assinaturas de código do *AutoConf* substituindo pelos grupos de comportamento encontrados por análise de log. As soluções para os desafios apresentados podem ser visualizadas na Tabela 5.1.

Desafio	**Solução**
Como adaptar a otimização das tarefas antes de sua execução?	Criar um histórico de tarefas agrupadas pelo consumo de recursos
Como otimizar novas tarefas antes de sua execução se a classificação é realizada em tarefas concluídas?	Acessar o histórico e relacionar tarefas similares em consumo de recursos, utilizando os grupos de inteção e de comportamento
Como analisar logs com dados não rotulados?	Utilizar um algoritmo de classificação não supervisionado, como o algoritmo K-means
Como agrupar tarefas que possam receber a mesma otimização?	Utilizar um algoritmo de classificação para agrupar os logs das tarefas

Tabela 5.1: Soluções encontradas para os desafios apresentados

Como visto na seção 4.2.5, ainda há estudos e melhorias para otimização do *Chameleon*, portanto, conclui-se que os trabalhos devem continuar. O *Chameleon* obteve resultados satisfatórios em 86% das consultas, sendo que em algumas delas o desempenho foi maior, como na consulta 9 que obteve mais de 20% de diminuição do tempo em relação ao *Hadoop* Padrão. O *Chameleon* poderá diminuir ainda mais o tempo de resposta das consultas seguindo os trabalhos futuros, por exemplo, realizando testes com novos algo-

ritmos, classificando através de outros recursos e implementando a abordagem em outras arquiteturas *MapReduce*.

BIBLIOGRAFIA

[1] A. Abouzeid, K. B. Pawlikowski, D. J. Abadi, A. Silberschatz, e E. Paulson. Efficient processing of data warehousing queries in a split execution environment. *Very Large Data Base Endowment Inc. (VLDB)*. Very Large Data Base Endowment Inc. (VLDB '11), 2011, 2011.

[2] Sonali Aggarwal, Shashank Phadke, e Milind A. Bhandarkar. Characterization of Hadoop Jobs Using Unsupervised Learning. *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings*, páginas 748–753. IEEE, 2010.

[3] Apache. Apache Hadoop Documentation, 2013.

[4] UC Berkeley. Spark and Shark - High-Speed In-Memory Analytics over Hadoop and Hive Data, 2012.

[5] D. Borthakur, K. Muthukkaruppan, K. Ranganathan, S. Rash, J. S. Sarma, N. Spiegelberg, D. Molkov, R. Schmidt, J. Gray, H. Kuang, A. Menon, e A. Aiyer. Apache hadoop goes realtime at facebook. *Special Interest Group on Management of Data (SIGMOD)*. Association for Computing Machinery's SIGMOD, 2011.

[6] Cloudera. Impala: Real-time Query for Hadoop, 2014.

[7] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, e C. Welton. Mad skills: New analysis practices for big data. *Very Large Data Base Endowment Inc. (VLDB)*. Very Large Data Base Endowment Inc. (VLDB '09), 2009, August 24-28, Lyon, France, 2009.

[8] J. Dean e S. Ghemawat. MapReduce: Simplified data processing on large clusters. *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, páginas 137–150, 2004.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, e W. Vogels. Dynamo: Amazon's highly available key-value store. *Symposium on Operating Systems Principles (SOSP)*. Symposium on Operating Systems Principles (SOSP '07), 2007, October 14-17, Stevenson, Washington, USA, 2007.

[10] R. Escriva, B. Wong, e E. G. Sirer. Hyperdex: A distributed, searchable key-value store. *Association for Computing Machinery's Special Interest Group on Data Communications (SIGCOMM)*. Association for Computing Machinery's Special Interest Group on Data Communications (SIGCOMM '12), 2012, August 13-17, Helsinki, Finland, 2012.

[11] Edson Ramiro Filho. HiveQL Self-Tuning, 2013.

[12] The Apache Software Fundation. Rumen: a tool to extract job characterization data from job tracker logs, 2013.

[13] Márcio Leandro Gonçalves, Márcio Luiz de Andrade Netto, Jurandir Zullo Jr., e José Alfredo Ferreira Costa. Unsupervised classification of remote sensors images using self-organizing neural networks and hierarchical clustering methods. Relatório técnico, PUC Minas, UNICAMP, UFRN, 2008.

[14] H. Herodotou, H. Lim, Gang Luo, N. Borisov, Liang Dong, F. B. Cetin, e Babu S. Starfish: A self-tuning system for big data analytics. *Conference on Innovative Data Systems Research (CIDR)*. 5a Biennal Conference on Innovative Data Systems Research (CIDR '11), 2011.

[15] Anil K. Jain. Data clustering: 50 years beyound k-means. *International Conference on Pattern Recognition (ICPR)*. 19a International Conference on Pattern Recognition (ICPR '08), Tampa, FL, December 8, 2008, 2008.

[16] A. Labrinidis e H. V. Jagadish. Challenges and opportunities with big data. *Very Large Data Base Endowment Inc. (VLDB)*. Very Large Data Base Endowment Inc. (VLDB '12), 2012, August 27-31, Istanbul, Turkey, 2012.

[17] H. Lim, H. Herodotou, e S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *Very Large Data Base Endowment Inc. (VLDB)*. Very Large Data Base Endowment Inc. (VLDB '12), 2012, August 27-31, Istanbul, Turkey, 2012.

[18] Wei Lu, Yanyan Shen, Su Chen, e Beng Chin Ooi. Efficient Processing of k Nearest Neighbor Joins using MapReduce. *Very Large Data Base Endowment Inc. (VLDB)*. Very Large Data Base Endowment Inc. (VLDB '12), 2012, August 27-31, Istanbul, Turkey, 2012.

[19] Edson R. Lucas Filho, Eduardo C. Almeida, e Yves Le Traon. Intra-query Adaptivity for MapReduce Query Processing Systems. *International Database Engineering e Applications Symposium (IDEAS)*. 18th International Database Engineering e Applications Symposium (IDEAS '14), 2014.

[20] Edson R. Lucas Filho, Ivan L. Picoli, Eduardo C. Almeida, e Yves Le Traon. Chameleon: The Performance Tuning Tool for MapReduce Query Processing Systems. *Simpósio Brasileiro de Banco de Dados (SBBD)*. XXIX Simpósio Brasileiro de Banco de Dados (SBBD '14), 2014.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, e A. Tomkins. Pig latin: A not-so-foreign language for data processing. *Association for Computing Machinery's SIGMOD*. Special Interest Group on Management of Data (SIGMOD '08), 2008, June 9-12, Vancouver, BC, Canada, 2008.

[22] Spiros Papadimitriou e Jimeng Sum. Disco: Distributed co-clustering with MapReduce. ICDM, 2008.

[23] Ivan L. Picoli e Eduardo C. de Almeida. Uma Abordagem de Classificação não Supervisionada de Carga de Trabalho MapReduce Utilizando Análise de Log. *Simpósio Brasileiro de Banco de Dados (SBBD)*. XXIX Simpósio Brasileiro de Banco de Dados (SBBD '14), 2014.

[24] Ivan L. Picoli, Leandro B. de Almeida, e Eduardo C. de Almeida. Otimização de Desempenho em Processamento de Consultas MapReduce. *Simpósio Brasileiro de*

Banco de Dados (SBBD). XXIX Simpósio Brasileiro de Banco de Dados (SBBD '14), 2014.

[25] Kenai Project. BTrace: *A Dymamic Instrumentation Tool for Java*. Oracle One 2008.

[26] Gang Qian, Shamik Sural, Yuelong Gu, e Sakti Pramanik. Similarity between euclidian and cosine angle distance for nearest neighbor queries. *ACM Symposium On Applied Computing*. 19th Annual ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14-17, 2004, 2004.

[27] Solange Oliveira Rezende. Mineração de dados. *Congresso da Sociedade Brasileira de Computação (SBC)*. XXV Congresso da Sociedade Brasileira de Computação (2005), UNISINOS, São Leopoldo, RS, 2005.

[28] T. D. Sanger. capítulo Optimal unsupervised learning in a single-layer linear feedforward neural network.

[29] K. Shvachko, H. Kuang, S. Radia, e R. Chansler. The Hadoop Distributed File System. *Mass Storage Systems and Technologies (MSST)*, páginas 1–10. IEEE Computer Society, 2010.

[30] Craig Silverstein, Monika Henzinger, Hannes Marais, e Michael Moricz. Analysis of a Very Large Web Search Engine Query Log. *ACM Special Interest Group on Information Retrieval (ACM SIGIR)*. 33th ACM SIGIR, New York, NY, USA, 1999.

[31] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, e Priya Narasimhan. Mochi: visual log-analysis based tools for debugging hadoop. *Workshop in Hot Topics in Cloud Computing*. Workshop in Hot Topics in Cloud Computing, 2009, June 15, Sandiego CA, EUA, 2009.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, Hao Liu, P. Wyckoff, e R. Murthy. Hive - a warehousing solution over a mapreduce framework. *Very Large Data Base Endowment Inc. (VLDB)*, 2009.

[33] TPC. TPC-H, 2013.

[34] Cisco Visual Networking Index (VNI). Cisco visual networking index: Forecast and methodology, 2012 a 2017. Relatório técnico, Cisco, 2012.

[35] Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, e Zhenjie Zhang. Efficient and Effective KNN Sequence Search with Approximate n-grams. *Very Large Data Base Endowment Inc. (VLDB)*. Very Large Data Base Endowment Inc. (VLDB '13), 2013, September 1-5, Hangzhou, China, 2013.

[36] Chen Yanpei, Sara Alspaugh, e Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Very Large Data Base Endowment Inc. (VLDB)*. Very Large Data Base Endowment Inc. (VLDB '12), 2012, August 27-31, Istanbul, Turkey, 2012.