cap:3

Capítulo 3

Otimização de Desempenho em Processamento de Consultas MapReduce

Ivan Luiz Picoli, Leandro Batista de Almeida, Eduardo Cunha de Almeida

Abstract

Performance tuning in MapReduce query processing systems is a current hot topic in database research. Solutions like Starfish and AutoConf provide mechanisms to tune MapReduce queries by changing the values of the configuration parameters for the entire query plan. In this course, it will be presented some performance tuning solutions for query processors based on MapReduce focused on a solution using Hadoop, Hive and AutoConf. Furthermore, it will be presented how the unsupervised learning can be used as a powerful tool for the parameter tuning.

Resumo

Otimização de desempenho em processamento de consulta MapReduce é um tópico de pesquisa bastante investigado atualmente. Soluções como o Starfish e AutoConf fornecem mecanismos para ajustar a execução de consultas MapReduce através da alteração de parâmetros de configuração. Neste minicurso serão apresentadas soluções de tuning para processadores de consulta baseados em MapReduce com foco principal numa solução baseada em Hadoop, Hive e AutoConf. Além disso, iremos discutir como um sistema de aprendizado de máquina não supervisionado pode ser utilizado como uma poderosa ferramenta na tarefa de configuração de parâmetros do sistema.

3.1 Introdução

Na última década, houve um crescimento exponencial na quantidade de dados a serem armazenados na rede mundial. A Internet se tornou o meio de comunicação mais utilizado do planeta e devido à facilidade que essa tecnologia proporciona cada vez mais pessoas se juntam às redes sociais e utilizam sistemas em nuvem para armazenar arquivos e assistir vídeos em alta resolução. Além disso, ligações utilizando voz sobre IP também crescem a cada ano, auxiliando no aumento do tráfego de dados. Segundo a empresa Cisco, em 2015 o tráfego de dados mundial anual atingirá 1.0 ZB, o que equivale a 87,125 PB por mês [Cisco 2012]. Aliado a isso, sistemas de *business intelligence* começam a utilizar uma massa de dados cada vez maior, para gerar relatórios e estratégias de negócio.

Fatores como: crescente massa de dados, a exaustão de recursos dos sistemas centralizados e seu custo mais alto para aumento de capacidade exige escalabilidade dos sistemas, obrigando a utilização de sistemas distribuídos. A escalabilidade gera grandes esforços de hardware e software para o armazenamento, processamento e transferência de informações, e processos que levavam algumas horas passam a levar dias para concluírem. Da mesma forma, podemos analisar os esforços no desenvolvimento de novos sistemas de banco de dados e *data warehouse* distribuídos, de acordo com a necessidade de processamento.

Com o crescimento da demanda de processamento analítico, outro esforço no desenvolvimento de novos *data warehouse*s baseados em sistemas escaláveis é atender as propriedades MAD (*Magnetic, Agile* e *Deep*) [Cohen 2009] criadas em 2009 que determinam uma arquitetura para padronização e realização de *business inteligence* em *data warehouses*. Essas características exigem que o sistema de *data warehouse* seja capaz de processar em tempo hábil toda a demanda requisitada. No cenário de hoje, reduzir o tempo em processamento de consultas pode gerar benefícios como economia no uso de recursos de hardware [Babu e Herodotou 2011]. O presente minicurso aborda os esforços atuais e formas de otimização do desempenho de *data warehouses* baseados em MapReduce. Também apresentaremos uma solução de otimização de consultas baseada em uma abordagem de aprendizado não supervisionada de máquinas.

3.2. BigData

O conceito de Big Data é amplo, mas pode ser resumidamente definido como o processamento (eficiente e escalável) analítico de grandes volumes de dados complexos, produzidos por possivelmente várias aplicações [Labrinidis e Jagadish 2012]. É uma estratégia que se adequa a cenários onde se faz necessário analisar dados semiestruturados e não estruturados, de uma variedade de fontes, além dos dados estruturados convencionalmente tratados. Também se mostra interessante quando o conjunto completo de informações de uma determinada fonte deve ser analisada e processada, isso torna o processamento lento, permitindo análises interativas e exploratórias.

Esses conceitos se apoiam nos chamados "V"s de BigData: Variedade, Volume e Velocidade, além de Veracidade e Valor [Troester 2012]. De maneira simplista, a Variedade mostra que os dados podem vir de fontes das mais diversas, com estrutura ou não, em formatos também diversificados. O Volume acrescenta a complexidade de se tratar espaços de dados da ordem de Pb ou Zb, e a Velocidade exige que esse

processamento ocorra ainda em tempo suficientemente pequeno para que as análises possam ser de utilidade. O Valor e Veracidade se referenciam a confiabilidade, correlação e validação das informações. Analisado pela ótica dos "V"s, o panorama de BigData se mostra complexo e com tarefas que fogem ao que geralmente é encontrado em processamento de dados.

Sendo assim, novas técnicas de processamento distribuído vêm sendo pesquisadas, desenvolvidas e aprimoradas. Uma das tecnologias existentes é o MapReduce [Dean e Ghemawat 2004]. Desenvolvido para simplificar o processamento distribuído, o MapReduce torna os problemas de escalabilidade transparentes, possibilitando que um programa do usuário seja executado de forma distribuída sem a preocupação com os diversos fatores que dificultam esse tipo de programação. Exemplos de dificuldades são as falhas de rede e a comunicação entre os nodos de armazenamento espalhados pela rede.

3.3. MapReduce

O MapReduce, desenvolvido em 2004 pela Google Inc. [Dean e Ghemawat 2004] foi projetado para simplificar as tarefas de processamento distribuído que necessitavam de escalabilidade linear. Esse tipo de tarefa e plataformas baseadas em nuvem podem ser reduzidas às aplicações MapReduce, juntamente com o ecossistema de aplicações envolvidas a partir dessa tecnologia emergente [Labrinidis e Jagadish 2012] .

As principais características do MapReduce 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. Gerenciar uma grande quantidade de dados requer processamento distribuído, e o uso de sistemas de armazenamento e busca utilizando pares de chave/valor tornou-se comum para esse tipo de tarefa, pois oferecem escalabilidade linear.

O Hadoop [Apache 2014] é uma implementação de código aberto do MapReduce, hoje gerenciado e repassado a comunidade de desenvolvedores através da Fundação Apache. Existem outras implementações baseadas em MapReduce, como por exemplo o Disco [Papadimitriou e Sum 2008], Spark [UC Berkeley 2012], Hadapt [Abouzeid 2011] e Impala [Cloudera 2014].

3.3.1. Arquitetura do MapReduce

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 utilizaremos um programa que realiza a contagem da frequência de palavras em um arquivo texto. A Figura 3.1 ilustra um programa MapReduce.

A função *map* recebe por parâmetro uma chave (nome do documento) e um valor (conteúdo do documento) e 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*.

```
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 3.1. Primitivas Map e Reduce contando a frequência de palavras em um texto [Dean e Ghemawat 2004]

3.3.2. Hadoop

O Hadoop é um framework de código aberto desenvolvido na linguagem Java, C e Bash. O framework disponibiliza bibliotecas em Java (.jar) para desenvolvimento de aplicações MapReduce, onde o programador poderá criar suas próprias funções Map e Reduce.

3.3.2.1. Entrada e Saída de Dados

O Hadoop possui o HDFS (Hadoop Distributed File System) [Shvachko 2010] como sistema de arquivos, sendo responsável pela persistência e consistência dos dados distribuídos. Ele 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.

3.3.2.2. Arquitetura do Hadoop

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

- **DataNode**: cliente do HDFS que gerencia o armazenamento local de dados e controla pedidos de leitura e escrita em disco;

- **NameNode**: servidor mestre do HDFS que gerencia a distribuição dos arquivos na rede e controla o acesso aos arquivos;

- **JobTracker**: servidor mestre do motor de processamento do Hadoop; é responsável por coordenar a distribuição das funções Map e Reduce para cada nodo.

- **TaskTracker**: cliente do motor de processamento do Hadoop; é responsável por executar as funções Map e Reduce recebidas do JobTracker sobre os dados locais do próprio nodo;

O motor de processamento MapReduce é formado pelo servidor JobTracker e vários clientes TaskTracer. Usualmente, mas não necessariamente, executa-se um DataNode e um TaskTracker por nodo, enquanto que executa-se apenas um JobTracker e um NameNode em toda a cluster/rede.

A Figura 3.2 mostra a arquitetura do Hadoop e seus módulos.

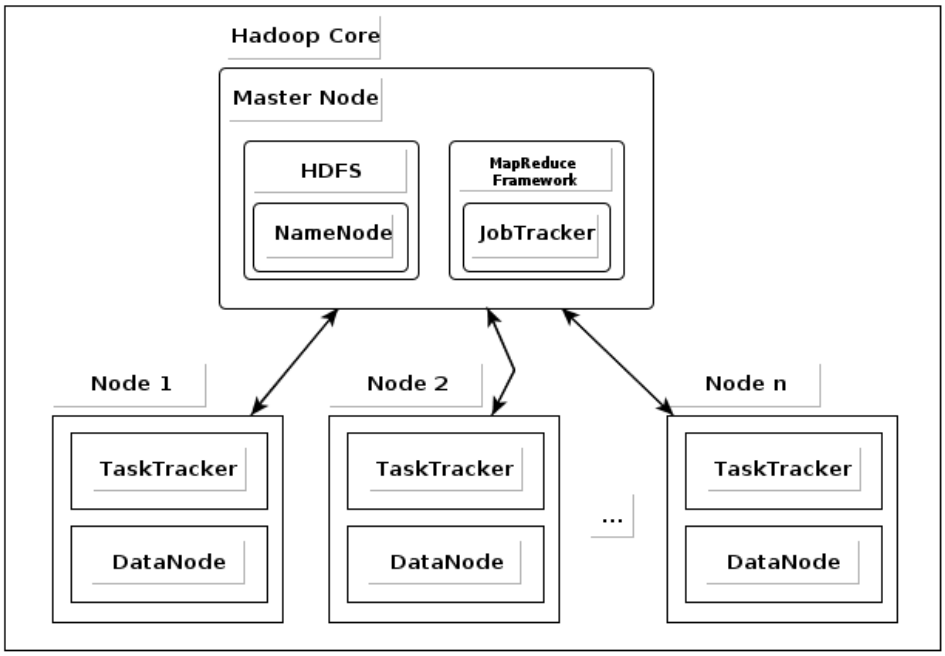

Figura 3.2. Arquitetura do Hadoop e seus módulos

Algumas tarefas importantes executadas pelo servidor mestre (master node) são as 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;

- Armazena o estado de tarefas Map e Reduce, como: *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*".

A Figura 3.3 mostra o fluxo de informações citadas nesta seção, 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 servidor mestre, responsável por atribuir tarefas de Map e Reduce às demais denominadas *workers*. Os *workers* 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 armazenados em buffer na memória e periodicamente salvos em disco em regiões de memória que são enviadas ao servidor mestre.

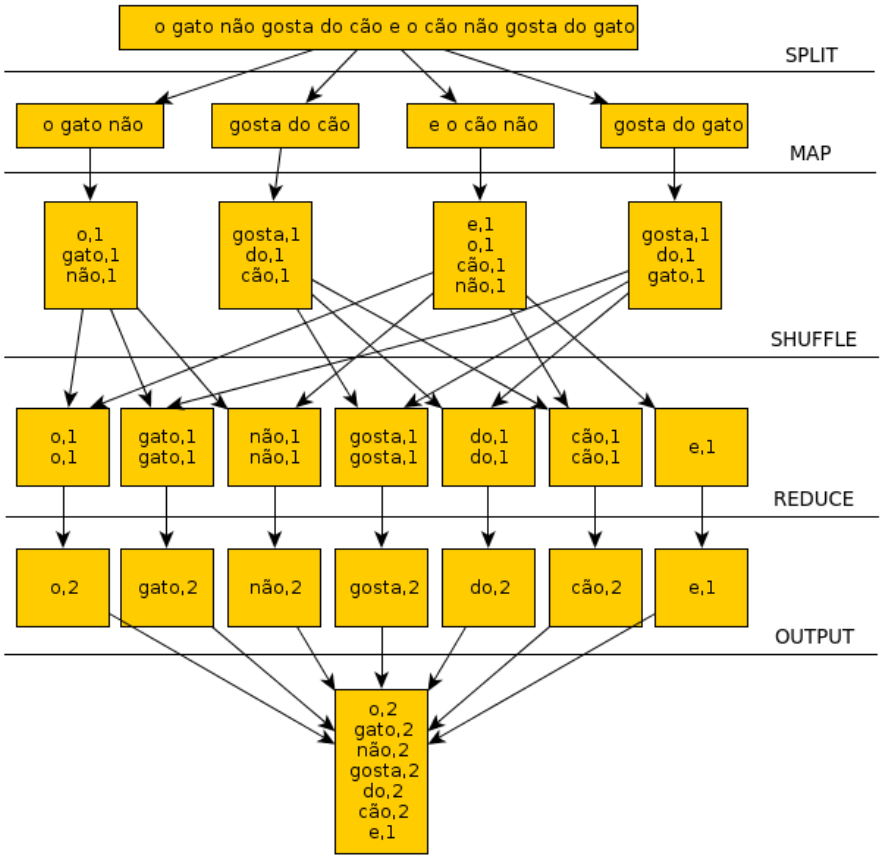

Figura 3.3. Fluxo de informações durante um programa MapReduce para contar palavras

O fluxo de informações pode ser dividido nas fases a seguir:

- 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 do Job.

3.4. Frameworks construídos sobre sistemas baseados em MapReduce

O uso do MapReduce e do Hadoop em ambientes de produção cresceu na última década devido a grande massa de dados denominada BigData. Pode-se destacar o uso dessas tecnologias principalmente nas grandes empresas de tecnologia da informação. Para cada situação do cotidiano da empresa novos programas MapReduce eram gerados, porém, com o tempo o número de programas cresceu e tornou-se inviável a reescrita de programas todos os dias conforme a necessidade da empresa. Tendo em vista o desafio de gerar programas MapReduce mais eficazes e rapidamente, a empresa Facebook desenvolveu o Data Warehouse Apache Hive [Thusoo 2009] que hoje é gerenciado pela Fundação Apache e possui código aberto.

O Hive é um exemplo de framework desenvolvido sobre o Hadoop. Ele possui uma linguagem declarativa chamada HiveQL para geração de consultas. O principal objetivo desse framework é a geração automática de programas MapReduce através da análise da consulta HiveQL para manipulação do *data warehouse*. A linguagem HiveQL não possibilita alteração e deleção de registros, mas possibilita a criação de tabelas e importação de dados de arquivos. Possibilita também criar partições nos dados para melhorar o desempenho do sistema. Quando uma partição é criada, a tabela é dividida em subtabelas internamente, tornando possível selecionar os dados apenas de uma partição se assim desejar. Isso auxilia na redução de leitura em disco. A Figura 3.4 mostra um script HiveQL gerado pela empresa Facebook.

```
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 3.4. Exemplo de script HiveQL [Thusoo 2009]

Podemos observar na Figura 3.4 a manipulação de duas tabelas e a execução de um *join*. 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.

Existem diversos frameworks desenvolvidos sobre sistemas baseados em MapReduce, outro exemplo de framework sobre o Hadoop é o Pig [Olston 2008] e sua linguagem declarativa PigLatin. Outro exemplo de framework é o Shark sobre o Spark [UC Berkeley 2012]. Além de sistemas de *data warehouse*, existem frameworks para otimização das tarefas executadas. Existem várias formas de otimizar a execução de uma tarefa, uma delas é ajustar os valores dos parâmetros de configuração do sistema MapReduce. O Hadoop, por exemplo, possui mais de 200 parâmetros de configuração. Um exemplo de framework construído sobre o Hadoop para otimização é o Starfish [Herodotou 2011], sendo responsável por sugerir uma melhor configuração de parâmetros ao administrador do sistema para cada tarefa MapReduce executada.

3.5. Otimização de Consultas em MapReduce

Nesta seção apresentamos de forma geral como funciona a otimização de desempenho do processamento de consultas MapReduce. O desempenho de aplicações MapReduce 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 auto ajustável, o qual ajusta seus parâmetros de acordo com a carga de trabalho submetida é uma solução para a diminuição do tempo de processamento dos dados.

O ajuste de parâmetros para otimização pode ser feito de duas formas, pelo administrador do cluster ou automaticamente por uma ferramenta de auto ajuste. A segunda alternativa é mais complexa, pois é necessário que as tarefas sejam classificadas e os parâmetros sejam ajustados antes que a tarefa seja executada. 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.

O desempenho das aplicações MapReduce 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. Para que um software de auto ajuste de parâmetros seja capaz de identificar os valores corretos, vê-se necessário a implementação de regras que classificam as tarefas de MapReduce. Alguns softwares de otimização preveem o possível uso de recurso computacional antes mesmo da execução propriamente dita. Porém, essa classificação não mostra o real uso de recursos que a tarefa irá utilizar e sim uma projeção através de regras impostas, este é o caso do Starfish e AutoConf. A sessão 3.5.1 mostra uma visão sobre como classificar uma tarefa MapReduce através de um perfil e como é possível utilizar arquivos de log para classificá-las. A sessão 3.5.2 apresenta o AutoConf, uma ferramenta que auxilia um *data warehouse* baseado em MapReduce a otimizar suas consultas a partir da análise dos operadores da consulta. A sessão 3.5.3 mostra uma abordagem que utiliza a análise de log em conjunto

com uma ferramenta de otimização baseada em regras, com o objetivo de obter uma otimização mais eficaz.

3.5.1. Geração de Perfis e Análise de Log

A geração de perfis consiste em coletar informações sobre a execução de uma tarefa e criar o que chamamos de perfil, contendo todas as informações coletadas. Essa técnica é importante, pois através de buscas e análises nas informações coletadas é possível ajustar os parâmetros de sistemas MapReduce.

A análise de log irá auxiliar na geração de um perfil mais preciso para as tarefas, pois os arquivos de log contêm informações de tarefas já executadas e qual foi o comportamento das mesmas, em termos de uso de recursos computacionais. Sendo assim, tarefas similares podem receber ajustes de parâmetros de acordo com o comportamento já conhecido.

Sistemas como o Mochi [Tan 2009] e o Rumen [Apache 2013] 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. O uso de logs para descoberta de padrões de comportamento é uma opção que auxilia na otimização. O Hadoop, por exemplo, possui um módulo que gerencia o histórico de execução de suas tarefas chamado JobHistory, onde 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 se torna possível o uso de um mecanismo de classificação para encontrar um comportamento comum de uso de recursos computacionais.

No caso do Hadoop, as informações mais relevantes a serem extraídas se encontram armazenadas nos contadores gerados durante a execução da tarefa. Como por exemplo, a quantidade bytes lidos do HDFS. A Figura 3.5 mostra alguns exemplos de parâmetros de configuração utilizados por uma tarefa Hadoop, e a Figura 3.6 mostra o histórico de um Map contendo um contador. Ambas as figuras foram extraídas de arquivos de log do Hadoop.

Na Figura 3.5 podemos observar nas linhas 15, 16 e 17 parâmetros contendo valores inteiros, porém, a partir da linha 18 observamos um parâmetro contendo uma consulta, esse parâmetro é o nome da tarefa.

Na Figura 3.6, podemos observar na linha 9 o identificador do Job e seu estado. Na linha 10 observamos uma subtarefa de Map, seu identificador e horário de início em milissegundos. Nas linhas 11 e 12 observamos as fases de execução do Map e seu histórico, contendo os contadores e as informações a serem extraídas para classificação.

```
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 3.5. Exemplos de parâmetros de configuração armazenados em arquivo de log do Hadoop

```
 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 3.6. Histórico de execução de um Map extraído de arquivo de log do Hadoop

3.5.2. AutoConf

AutoConf [Lucas Filho 2013] é uma ferramenta desenvolvida para otimização de consultas do sistema de Data Warehouse Apache Hive. O Hive analisa a consulta e a divide em estágios, onde cada estágio é um Job no Hadoop. O AutoConf realiza a análise dos operadores utilizados em cada estágio da consulta e extrai uma assinatura de código para esses estágios, 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 da tarefa ser executada. Ou seja, as tarefas MapReduce ou 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. Os grupos de tarefas classificadas a partir das assinaturas de código recebem o nome de grupos de intenção. A Figura 3.7 mostra a arquitetura do AutoConf e sua integração com o ecossistema do Hadoop.

O processo de otimização do AutoConf na Figura 3.7 é descrito a seguir.

- 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 (Jobs) 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.

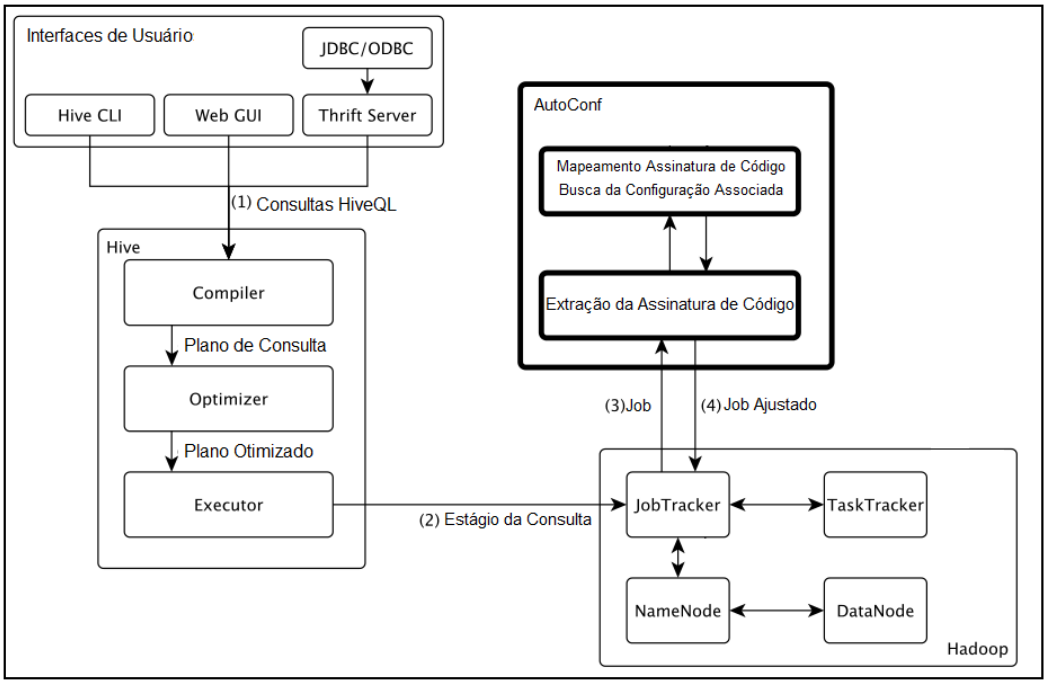

Figura 3.7. AutoConf em meio ao ecossistema do Hive e Hadoop [Lucas Filho 2013]

3.5.3. Uma Abordagem de Otimização de Consultas por Análise de Log

O AutoConf, como vimos, classifica os estágios da consulta (tarefas MapReduce) antes da execução, analisando o script HiveQL e extraindo os operadores. Essa forma de otimização prevê o uso de recursos computacionais para classificar as tarefas. A junção da análise da consulta, no caso HiveQL, e a análise do histórico de logs é uma forma de classificar mais corretamente as tarefas oriundas da consulta. Essa hipótese existe devido aos seguintes tópicos.

- Os arquivos de log dos sistemas MapReduce 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 uso de recursos computacionais, predições são menos representativas do que os dados armazenados em arquivos de log;

- Os clusters e perfis criados antes da execução da tarefa são pouco representativos em termos do comportamento real das tarefas, pois utilizam regras;

A melhoria citada 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. As informações serão extraídas dos arquivos de log, sendo possível encontrar padrões com mais precisão e voltados para os recursos computacionais definidos pelo administrador do sistema.

Para analisar os arquivos de log, utilizaremos algoritmos de classificação não supervisionados, responsáveis por classificar os logs em grupos, onde os logs integrantes de cada grupo possuirão comportamento similar. Sendo assim, tarefas oriundas de novas consultas poderão ser comparadas aos clusters gerados pelo algoritmo. O AutoConf classifica as tarefas de acordo com a assinatura de código recebida, então denominaremos os grupos de distintas assinaturas de código como **"grupos de intenção"**. O aprendizado não supervisionado classifica os logs das tarefas através do comportamento encontrado pelo algoritmo, então, denominaremos esses grupos como **"grupos de comportamento"**. É possível utilizar diversos algoritmos de aprendizagem não supervisionada durante a otimização, um deles é o K-Means [Jain 2008].

O estudo sugere o uso de logs de tarefas MapReduce, sendo assim, é possível utilizar qualquer sistema baseado em MapReduce que gere algum tipo de arquivo de log.

3.6. Aprendizado de Máquina

A classificação é uma técnica de mineração de dados [Rezende 2005]. Sua função é classificar os dados de forma a determinar grupos de dados com características comuns. Abordaremos o algoritmo de aprendizagem não supervisionada K-Means para geração dos grupos de comportamento.

3.6.1. 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 centroides dos clusters. Por exemplo, se desejamos seis grupos, o registro será classificado para o grupo onde a distância do centroide é mais próxima. Desta maneira, o algoritmo automaticamente fornecerá uma classificação sem nenhum conhecimento preexistente dos dados. Nesta sessão veremos como são calculados os valores dos centroides 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 [Qian, 2004] para calcular o quão distante uma ocorrência está da outra. Após o cálculo das distâncias o algoritmo calcula centroides para cada uma das classes. Considere os pontos *P* e *Q* em (1).

$$P = \{p_1, p_2, ..., p_n\} \text{ e } Q = \{q_1, q_2, ..., q_n\} \quad (1)$$

A equação abaixo (2) representa o cálculo da distância euclidiana entre os pontos *P* e *Q* em um espaço n-dimensional.

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

Uso com logs: No caso de logs, suponhamos que estamos trabalhando com apenas duas dimensões (total de bytes lidos e total de bytes escritos). Essas são as dimensões contidas no vetor *P* e *Q* da equação descrita. Suponhamos que *P* seja o ponto do centroide de uma das classes no espaço bidimensional e *Q* seja o ponto para um determinado log no espaço bidimensional. Os valores das dimensões do centroide são iguais às médias das dimensões de todos os logs classificados na classe de tal centroide, 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, descritos pelo Algoritmo 3.1.

No Algoritmo 3.1, os vetores *C* e *LC* armazenam o resultado final da classificação. *C* contém as classes que representam o uso dos recursos computacionais, onde os recursos são representados pelas dimensões *dim*_x. *LC* contém a referência para os logs e a qual classe eles pertencem após a classificação.

A Figura 3.8 contém um exemplo de execução do K-Means com duas dimensões (bytes lidos e escritos). Podemos observar na Figura 3.8 a mudança da posição dos centroides no decorrer dos 6 estágios e a redistribuição dos registros (logs) nas classes de acordo com suas distâncias euclidianas.

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**

Algoritmo 3.1. Algoritmo K-Means em um estudo de caso ao analisar logs

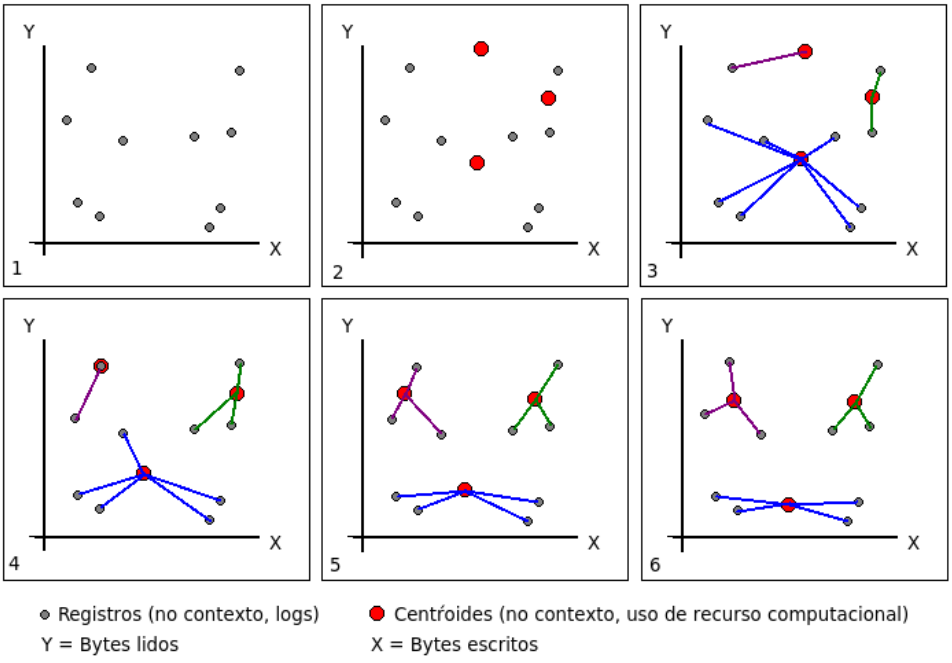

Figura 3.8. Exemplo de execução do K-Means com duas dimensões

3.7. Chameleon: Otimização de Consultas por Análise de Log

Chameleon é a uma ferramenta de auto ajuste de desempenho que utiliza o algoritmo K-Means sobre os logs de execução do Hadoop. O Chameleon complementa o AutoConf auxiliando na otimização de consultas do Hive, através da atribuição de um grupo de comportamento às tarefas da consulta. A Figura 3.9 mostra a arquitetura do Chameleon e onde se enquadra no ecossistema do Hadoop.

Como extensão da Figura 3.7, a Figura 3.9 adiciona a Análise de Log ao ecossistema, onde dois módulos distintos trabalham separadamente. O Hadoop Log Parser é a implementação dos dados de entrada, nesse caso especificamente para o Hadoop, mas também seria 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.

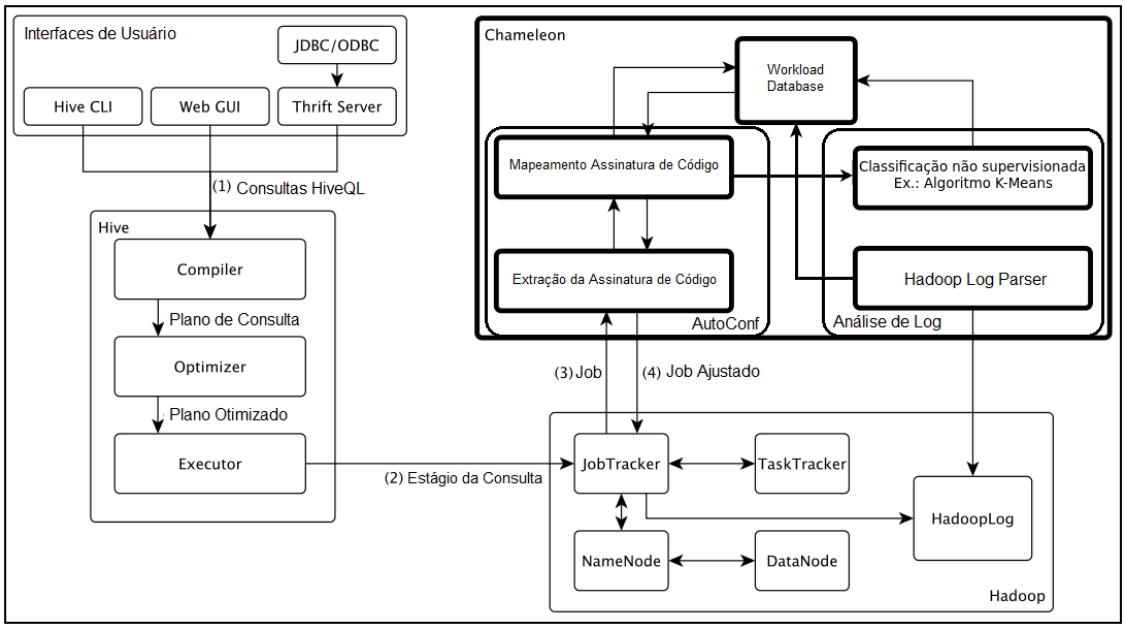

Figura 3.9. Chameleon e o ecossistema do Hadoop

3.7.1. Workload Database

O Workload Database 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 3.10 mostra o modelo entidade-relacionamento da base de dados.

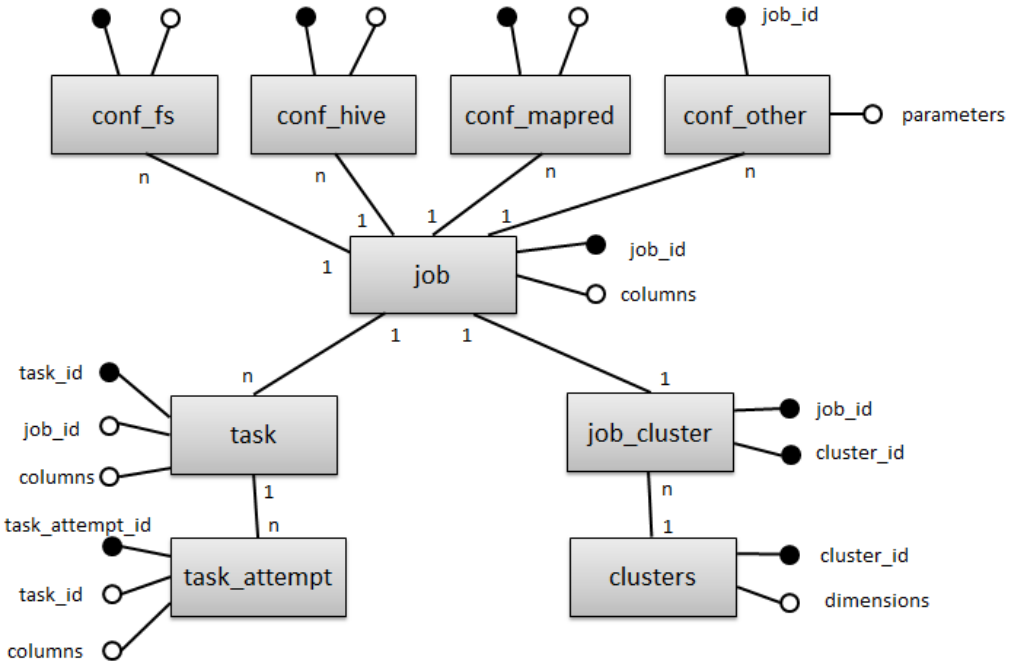

Figura 3.10. Modelo Entidade-Relacionamento da base de dados do Workload DW

Na Figura 3.10 a tabela *job* é a tabela principal, que armazena as informações únicas de cada log das tarefas executadas, e as tabelas iniciadas com *conf_* armazenam as configurações de parâmetros de diferentes tipos que foram utilizadas na execução das tarefas. As tabelas *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 clusters de comportamento são armazenados na tabela *clusters* e por fim, cada log é classificado a um cluster de comportamento através da tabela *job_cluster*.

3.7.2. Hadoop Log Parser

Hadoop Log Parser é a 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 Database com os logs. Esse módulo pode ser executado no modo *loop*, proporcionando a verificação de novos logs conforme o Hadoop processa novos Jobs. Algumas características fazem com que o LogParser torne-se um módulo 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 Database.

3.7.3. Classificação

A Figura 3.9 demonstra o módulo responsável por gerar os chamados grupos de comportamento, que determinam quais os recursos computacionais usados pelo Hadoop em determinadas tarefas. No caso do Chameleon são os recursos usados pelas consultas do Hive. O módulo de classificação possui diversas configurações que podem ser alteradas pelo administrador. A Figura 3.11 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 3.11. Arquivo com algumas configurações do módulo de classificação

- **refresh-time:** tempo de espera da Classificação para realizar uma classificação dos logs.
- **number-of-classes:** valor de *k* do algoritmo de classificação ou a quantidade de clusters 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. Uma dimensão é cada tipo de informação selecionada dos logs para processamento do algoritmo. Foram encontrados 63 tipos de informações possíveis de se extrair dos logs do Hadoop, e o módulo de classificação possui um arquivo de configuração que determina quais delas serão utilizadas. Então, o administrador pode configurar 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.

3.7.4. Viewer

Para visualização dos clusters gerados pela classificação foi desenvolvido o módulo Viewer, que se trata de uma interface web onde é possível a visualização dos clusters de comportamento gerados pelo K-Means e o histórico do uso de recursos. A Figura 3.12 mostra um exemplo de clusters de comportamento gerados pelo Chameleon. A Figura 3.13 mostra um exemplo de dimensões utilizadas com o uso de recurso computacional encontrado pelo algoritmo e a Figura 3.14 mostra o histórico do uso de um recurso no decorrer do tempo. Todas as figuras foram extraídas do Viewer.

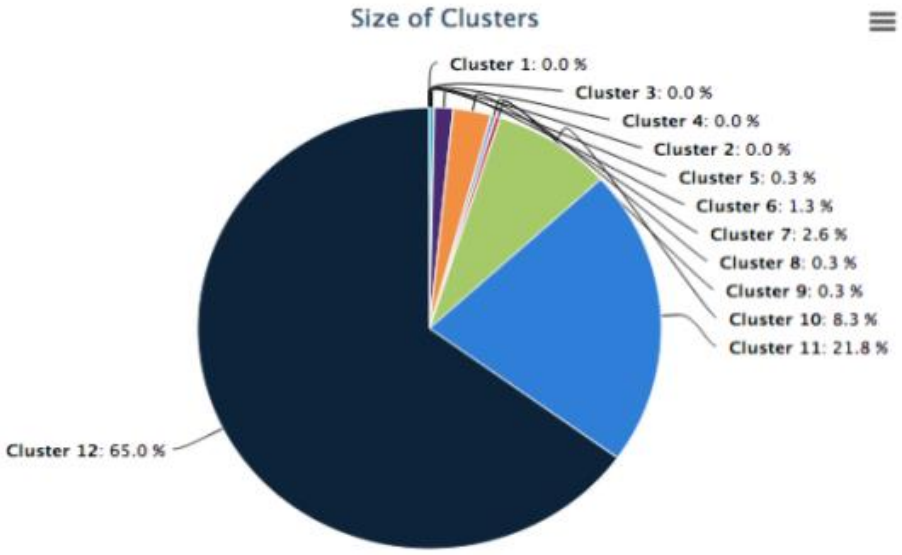

Figura 3.12. Clusters de comportamento gerados pelo Chameleon

Podemos observar na Figura 3.12 a predominância de um cluster, onde a 65% dos logs possuem comportamento similar.

Na Figura 3.13 podemos observar seis dimensões utilizadas no agrupamento de logs pelo algoritmo K-Means. Cada dimensão possui um valor diferente em cada cluster de comportamento, esse valor é a média de todos os logs, ou seja, o valor do centroide da dimensão.

Settings | Tuning | **Current Workload** | Workload Shifting | Report

Current Clusters

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 3.13. Clusters de comportamento com as dimensões utilizadas

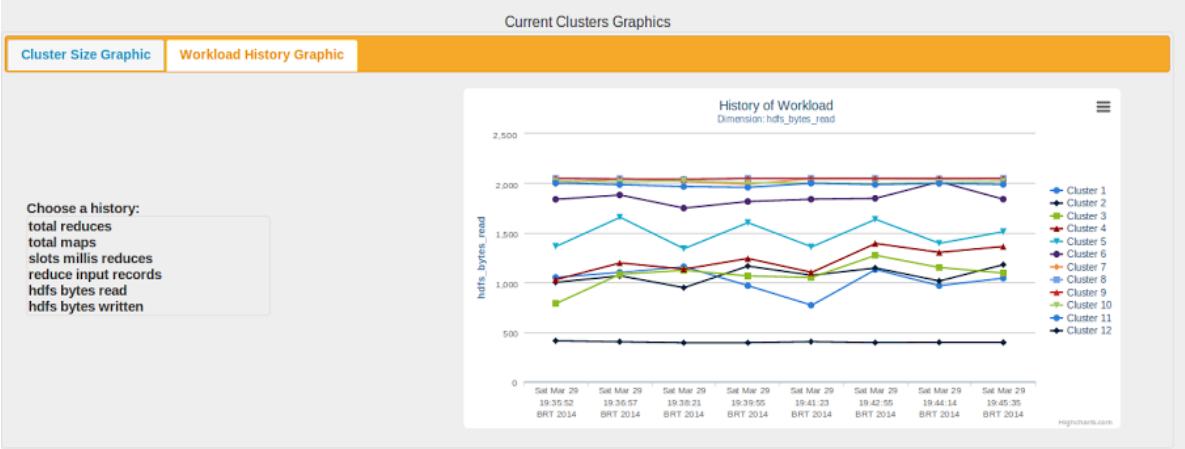

Figura 3.14. Histórico de bytes lidos durante 10 minutos de execução do Chameleon

Na Figura 3.14 podemos observar o histórico dos valores dos clusters de comportamento por dimensão, no caso, *hdfs_byte_read* durante um período de execução de dez minutos. Neste período, novas consultas chegaram e foram executadas pelo Chameleon, cada tarefa gerou um novo arquivo de log, esses arquivos foram adicionados à nova classificação do K-Means, o que resultou na mudança dos valores dos clusters e na quantidade de logs em cada cluster. O gráfico de linhas da Figura 3.14 demonstra essa mudança.

3.7.5. Integração do K-Means e AutoConf

Tendo como base o funcionamento dos módulos e o objetivo do Chameleon na otimização das consultas do Hive, viu-se necessário a criação de uma integração entre o AutoConf e a Classificação. 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 da Classificação, o qual é responsável por determinar qual será a configuração que deverá ser aplicada a partir dos grupos de comportamento. O método pode ser visualizado no Algoritmo 3.2, que mostra o processo de ajuste utilizando os grupos de comportamento.

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

Algoritmo 3.2. Algoritmo para aplicação da configuração a partir dos clusters de comportamento

No Algoritmo 3.2, 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. Como essas tarefas de histórico já foram completados elas possuem logs que estarão armazenados no Workload DB, então a Classificação 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 podemos visualizar a geração da lista no laço entre as linhas 4 e 7 do Algoritmo 3.2.

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 3.2). Em seguida, o AutoConf aplica a configuração relacionada ao grupo de comportamento encontrado.

3.7.6. Visão Geral sobre o Autoajuste do Chameleon

O autoajuste possui duas fases, na primeira é a extração dos operados da consulta do Hive e a partir da assinatura de código é gerado o grupo de intenção. Na segunda fase, através da classificação pelo K-Means são gerados os grupos de comportamento que determinam os recursos reais utilizados pelas tarefas. Através da comparação entre o grupo de intenção da tarefa a ser ajustada e os grupos de comportamento é determinada um configuração de parâmetros à tarefa. É importante saber que cada grupo de comportamento possui uma configuração de parâmetros predeterminada, que pode ser modificada pelo administrador do sistema. Então, quando uma nova tarefa chega, os parâmetros são ajustados de acordo com o arquivo de configuração atrelado ao grupo de comportamento da tarefa.

3.8. Conclusão

Neste minicurso apresentamos uma visão geral do processamento de consultas MapReduce e exemplificamos este processamento através de uma solução que engloba os sistemas Hive e AutoConf. Além de processamento de consultas sobre MapReduce, esta solução tem como objetivo encontrar os melhores parâmetros de configuração de desempenho. A modificação dos valores de parâmetros em diferentes tipos de tarefas influencia diretamente no uso de recursos, economizando processamento, disco e rede. Esse fato transforma essa abordagem de otimização em uma área ampla de pesquisas, abrindo oportunidades para diversos novos estudos. Aliando os parâmetros com análise dos operadores da consulta, arquivos de log e algoritmos de aprendizagem de máquina, ampliamos ainda mais as possibilidades de inovações na área.

Novos sistemas baseados em MapReduce são criados de forma a aperfeiçoar as tecnologias já existentes. Portanto, abordagens de otimização que são capazes de abranger as inovações da área de BigData e MapReduce conforme sua evolução, de forma ampla para vários sistemas, serão de extremo valor no avanço das pesquisas na área. Dessa forma, esperamos incentivar pesquisadores a unir seus esforços na pesquisa sobre otimização de consultas MapReduce.

Referências

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

Apache. Apache Hadoop Documentation, 2014. Disponível em < http://hadoop.apache.org/>.

Apache. The Apache Software Fundation. Rumen: a tool to extract job characterization data from job tracker logs, 2013. Disponível em < http://hadoop.apache.org/docs/r1.2.1/rumen.html>.

Babu, Shivnath e Herodotou, Herodotos. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. Very Large Databases (VLDB), 2011.

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

Cloudera. Impala: Real-time Query for Hadoop, 2014. Disponível em <http:// www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html>

Cohen, Jeffrey e Dolan, Brian e Dunlap, Mark e Hellerstein, Joseph M. e Welton, Caleb. MAD Skills: New Analysis Practices for Big Data. Very Large Databases (VLDB). 2009.

Dean, J. e Ghemawat, S. MapReduce: Simplied data processing on large clusters. 6th Symposium on Operating System Design and Implementation (OSDI), 2004.

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

Jain, Anil K. Data clustering: 50 years beyound k-means. International Conference on Pattern Recognition (ICPR). 19a International Conference on Pattern Recognition (ICPR), 2008.

Labrinidis, A. e Jagadish, H. V. Challenges and opportunities with big data. Very Large Data Base Endowment Inc. (VLDB), 2012.

Lucas Filho, Edson Ramiro. HiveQL self-tuning – Curitiba, 2013. 44f. : il. color. ; 30 cm. Dissertation (master) – UFPR, Pos-graduate Program in Informatics, 2013.

Olston, C. e Reed, B. e Srivastava, U. e Kumar, R. e Tomkins, A. Pig latin: A not-so-foreign language for data processing. Special Interest Group on Management of Data (SIGMOD), 2008.

Papadimitriou, Spiros e Sum, Jimeng. Disco: Distributed co-clustering with MapReduce. IEEE International Conference on Data Mining (ICDM), 2008.

Qian, Gang e Sural, Shamik e Gu, Yuelong e Pramanik, Sakti. Similarity between Euclidian and cosine angle distance for nearest neighbor queries. Symposium on Applied Computing (SAC), 2004.

Rezende, Solange Oliveira. Mineração de dados. XXV Congresso da Sociedade Brasileira de Computação (SBC), UNISINOS, 2005.

Shvachko, K. e Kuang, H. e Radia, S. e Chansler R. The Hadoop Distributed File System. IEEE Computer Society. Mass Storage Systems and Technologies (MSST), 2010.

Troester, Mark. Big Data Meets Big Data Analytics: Three Key Technologies for Extracting Real-Time Business Value from the Big Data That Threatens to Overwhelm Traditional Computing Architectures. SAS Institute Inc. White Paper. 2012.

Tan, Jiaqi e Pan, Xinghao e Kavulya, Soila e Gandhi, Rajeev e Narasimhan, Priya. Mochi: visual log-analysis based tools for debugging hadoop. Workshop in Hot Topics in Cloud Computing (USENIX), 2009.

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

UC Berkeley. Spark and Shark - High-Speed In-Memory Analytics over Hadoop and Hive Data, 2012. Disponível em <https://spark.apache.org/>.

Sobre os Autores

Eduardo Cunha de Almeida é Professor Adjunto III na Universidade Federal do Paraná (UFPR). Seus principais interesses de pesquisa são em sistemas de bancos de dados e data warehousing, com atenção especial na automatização e testes de sistemas de processamento distribuído de consultas. Recebeu seu doutorado em Ciência da Computação, com grandes honras (félicitations du jury), da Universidade de Nantes, França, em 2009. Sua tese de doutorado foi em testes de tabelas hash distribuídas (DHT). De 1998 a 2005, trabalhou como engenheiro de tecnologia de *data warehouse* no Banco HSBC, na GVT Telecom e na Fundação UFPR. De 2010 a 2012, serviu como Vice-Coordenador do Programa de Pós-Graduação em Ciência da Computação da UFPR.

Leandro Batista de Almeida é professor de Ciência da Computação na Universidade Tecnológica Federal do Paraná (UTFPR) desde 1994, onde ministra disciplinas sobre sistemas de bancos de dados, computação móvel e linguagens de programação. Recebeu seu grau de mestre em 2000, em telemática, e seus principais interesses de pesquisa são em análise de redes sociais e BigData.

Ivan Luiz Picoli nasceu em 1990, é atualmente estudante de mestrado na Universidade Federal do Paraná (UFPR) com uma bolsa de tempo integral da Capes. Seus principais interesses em pesquisa são sistemas de bancos de dados baseados em MapReduce e data warehousing. Sua dissertação de mestrado trata de otimização de consultas em sistemas baseados em MapReduce usando aprendizado não-supervisionado e análises de logs. Ele obteve sua graduação em Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná (UTFPR) em 2012. De 2012 a 2013, trabalhou como desenvolvedor Java no Instituto Nacional de Colonização e Reforma Agrária (INCRA) em Brasília.