PAULO EDUARDO BOEIRA CAPELLER

AVALIAÇÃO DE SISTEMA BASEADO EM *MAPREDUCE* PARA CARREGAMENTO DE MODELOS

CURITIBA

2013

PAULO EDUARDO BOEIRA CAPELLER

AVALIAÇÃO DE SISTEMA BASEADO EM *MAPREDUCE* PARA CARREGAMENTO DE MODELOS

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. Marcos Didonet Del Fabro

CURITIBA

2013

"Entre os melhores encontrarás o pior,

e entre os piores encontrarás o melhor"

AGRADECIMENTOS

Agradeço a Deus por mais esse passo em minha vida, um passo repleto de alegrias e novidades, dificuldades e desafios superados com a ajuda de meus familiares.

Agradeço ao meu orientador Marcos Didonet del Fabro, que muito me ensinou, não apenas para o mestrado. Agradeço também a minha primeira orientadora, Simone Nasser Matos, professora para toda vida. Agradecimentos ao Edson Ramiro, que muito me ajudou em tudo que precisei.

Não menos, agradeço a todos os meus amigos, que me ajudaram em tantas batalhas que a vida nos trás, agradeço dos Aztecas aos Vikings.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE SIGLAS

CDO - *Connected Data Objects*

DDM - Desenvolvimento Dirigido a Modelos

EMF - *Eclipse Modeling Framework*

GB - *Gigabyte*

IBM - *International Business Machines*

LHC - *Large Hadron Collider*

LINQ - *Language Integrated Query*

MB - *Megabyte*

MDE - *Model-driven engineering*

MOF - *Meta-Object Facility*

OMT - *Object-relational mapping*

OODB - *Object-Oriented Database*

OOSE - *Object-Oriented Software Engineering*

ORM - *Object-relational mapping*

P2P - *Peer-to-Peer*

PB - *Petabyte*

RG - Registro Geral

TB - *Terabyte*

UML - Linguagem de Modelagem Unificada

XMI - *XML Metadata Interchange*

XML - *Extensible Markup Language*

RESUMO

O sucesso do desenvolvimento de software baseado em modelos levou ao estudo e aplicação destas técnicas em situações bastante diversas, como migração de sistemas legados, que antes não eram previstas. Isto exigiu a adaptação das técnicas existentes, especialmente em relação as abordagens de armazenamento, porque os *frameworks* existentes não foram projetados para suportar modelos com tamanho superior a alguns *megabytes*, isto é, grandes modelos. Existem alguns trabalhos que já identificaram esse problema, e propõem técnicas mais eficientes para o armazenamento de dados, como grafos, orientado a documentos. Atualmente, o uso de *frameworks* de processamento de dados distribuídos é uma alternativa comum para análise de dados em grande escala. Porém, estas técnicas são ainda pouco exploradas no contexto de desenvolvimento baseado em modelos. Uma abordagem bastante comum para manipulação de dados, é o *MapReduce*, que usa técnica baseada em programação funcional para distribuição do processamento. Este trabalho propõem uma solução para implementação e avaliação do *MapReduce* para o carregamento distribuído de modelos, com o objetivo de melhorar o desempenho. Para distribuir um modelo, é necessário conhecer as dependências dos seus elementos, como classes, atributos e referências, assim pode-se efetuar uma distribuição com maior independência, não comprometendo o processamento. Diferentes modelos podem exigir diferentes divisões, a maneira escolhida para distribuir os elementos é crucial para o desempenho e essas divisões podem alterar consideravelmente o tempo do processo e a quantidade me memória exigida para o mesmo. Este trabalho apresenta uma implementação que integra técnicas de MDE e *MapReduce* para carregar modelos de maneira distribuída. Escolhemos 4 métodos existentes para carregar e distribuir os elementos de modelos de maneira diferente, escolhemos o método Média das Referências, Média dos Tipos, Tamanho do Modelo e Uma Classe por Divisão.

ABSTRACT

The success of software development based on models led to the study and the application of these techniques in very different situations, as migration from legacy systems that were not previously envisaged, this necessary the adaptation of existing techniques, especially in relation to storage approaches, because the existing frameworks were not designed to support models larger than some megabytes, that is, large models. There are some works that have identified this problem, and they propose techniques for efficient storage, as graphs-oriented documents. Currently, the use of frameworks for distributed data processing is a common alternative for the analysis of large-scale data. However, these techniques are still little explored in the context of model based development. A fairly common approach to data manipulation, is the MapReduce, which uses techniques based on functional programming to Distribution processing. This paper proposes a solution for applying MapReduce to the distributed loading of models, with the aim of improve the performance. To distribute a model, it is necessary to know the dependencies of its elements, such as classes, attributes and references, so one can make a distribution with greater independence, not compromising the processing. Different models may require different splits, the way chosen to distribute the elements is crucial for performance and these splits can change significantly the processing time and the amount of required memory for the same. This paper presents an implementation and evaluation that integrates techniques MDE and MapReduce to load models in a distributed manner. We chose four existing methods to load and distribute the model elements of different ways, chose the method Average of References, Average of Types, Size of the Model and a Class per Division.

CAPÍTULO 1

INTRODUÇÃO

No processo de desenvolvimento de um *software* é comum encontrar dificuldades para modelar, documentar, construir e evoluir o *software* [12]. O Desenvolvimentos Baseado em Modelos (MDD) fornece um conjunto de técnicas e ferramentas para criação e manipulação de modelos minimizando essas dificuldades [28] [9].

Um modelo representa uma parte de um sistema, aplicados para um problema específico. Os modelos possuem um formato que deve ser interpretado por computador [12]. Um dos formatos para representação de modelos mais utilizado é o *Ecore* [9], um exemplo do seu uso é encontrado no *framework* EMF (*Eclipse Modeling Framework*) [30].

Existem várias abordagens de manipulação de modelos, com maneiras de armazenamento e processamento variadas, além do EMF, existe o Morsa, usado para mapear objetos não relacionais [16], também o CDO que é um Object-relational mapping (ORM). Entretanto, essas abordagens possuem processamento centralizado, e de acordo com Ashish Thusoo [32], Laurent Goubet [10], não suportam grandes modelos, por falta por exemplo, de processamento e memória.

Uma solução adotada por alguns *frameworks* para o processamento de dados de grande escala é a utilização da distribuição dos dados, um dos mais utilizados é o *MapReduce* [5], este permite distribuir o processamento de dados, baseado em programação funcional, além de características como replicação de dados para *backup* durante a operação. Porém, estas técnicas são ainda pouco exploradas no contexto de desenvolvimento baseado em modelos.

Das soluções existentes, apenas EMF-Fragment [23] estuda a distribuição de seus

dados, mas utiliza *frameworks* de abstração, por exemplo o *Hive* [32], como camada intermediária para manipular dados relacionais e não relacionais. não possibilitando modificações e ajustes de baixo nível.

A solução proposta nesta dissertação para a manipulação de grandes modelos é distribui-los usando o *MapReduce*, sem nenhuma camada intermediária extra de abstração associada, para isso técnicas para a distribuição de modelos, levando em consideração seus elementos foram analisadas, aplicadas e testadas.

Modelos representam, no mínimo uma parte de um sistema, para isso, possuem características próprias, como um metamodelo associado, há também referências entre as classes gerando dependências entre as mesmas. Essas características foram estudadas para adaptar um modelo em um sistema estilo *MapReduce*.

Diferentes modelos necessitam de diferentes técnicas de divisões para um melhor desempenho, como por exemplo, um modelo das regiões brasileiras, que possua uma grande quantidade de referências entre as classes do tipo cidade e estado, sendo favorável sua divisão por um método que leve em consideração esses tipos. Este trabalho criou um *driver* para o *MapReduce* com a funcionalidade de distribuição de modelos XMI. Foram implementadas quatro técnicas de divisões levando em consideração alguns critérios inspirados em Sheidgen [25], Yang [35] e Zubow [24]. Média dos Tipos, os modelos são divididos considerando a média entre a quantidade de tipos existentes no modelo e o número de classes. Média das Referências, técnica onde os modelos são divididos considerando o número de referências dividido pelo número de classes. Tamanho do Modelo, uma técnica aplicada levando em consideração o tamanho de cada divisão do modelo. Uma Classe por Divisão cria cada divisão com apenas uma classe. Foram realizados vários experimentos com modelos extraídos de classes java, e também modelos criados para os testes, de tamanho entre 180 MB até 307 GB, para avaliar se há ganhos na distribuição com o *MapReduce*, e a partir de que tamanho, utilizando todas as técnicas implementadas.

Este trabalho está organizado em 4 capítulos. O Capítulo 2 introduz conceitos

da Engenharia Dirigida a Modelos (MDE), *frameworks* que possuem características necessárias para a manipulação de modelos de grande escala, sistemas distribuídos e o *MapReduce*. O Capítulo 3 apresenta o *driver* criado para a distribuição de modelos, exibe 4 métodos de criação de divisão desses modelos, e seus desempenhos. Finalmente o Capítulo 4 descreve a conclusão e propostas de trabalhos futuros dessa pesquisa.

CAPÍTULO 2

ENGENHARIA DIRIGIDA POR MODELOS

Engenharia dirigida por modelos, do inglês *Model-driven engineering* (MDE) tem como característica utilizar modelos como entidade de primeira classe. Essas entidades são utilizadas como referência para outras entidades da mesma linguagem [11]. Quanto maior o grau de redundância no código maior o lucro com o uso da MDE, pois uma das principais características da MDE é eliminar redundâncias, diminuindo assim custos no desenvolvimento [26].

Neste capítulo sera descrito, os conceitos da engenharia dirigida por modelos, as características dos modelos e metamodelagem, padrões de armazenamento de modelos e *frameworks* que trabalham com modelos.

2.1 Modelos

Os modelos possuem a finalidade de representar um sistema de maneira simplificada, sendo capaz também de responder possíveis dúvidas do mesmo, como por exemplo, quais as características das cidades de um determinado estado [3]. Os modelos seguem uma sintaxe, a qual deve ser sempre interpretada por computador [12].

A grande quantidade de informações e sistemas computacionais requerem modelos cada vez maiores, afim de proporcionar facilidade em seu compreendimento e reutilização. O uso de modelos revela vantagens para o projeto em relação ao custo, compreensão, disponibilidade, segurança e automação [12]. Por exemplo, um modelo pode representar um sistema de venda de produtos *on-line*, onde classes representam os produtos, clientes e suas buscas recentes. Outro exemplo de uso é a modelagem de uma aplicação *WEB* de um banco, onde seus clientes são representados por classes contendo informações, tias

como nome, um código único identificador e suas contas, outra classe pode possuir as informações das contas como número, agência, saldo entre outras informações. Também podem ser representados através de modelos, sistemas *workflow*, onde deve haver uma representação mais detalhada na entrada e saída de dados [13].

2.1.1 Metamodelagem

O modelo, que é uma tripla onde $M = (G, \omega, \mu)$, onde, G é um conjunto finito de nós e arestas do grafo, ω é um modelo associado ao multigrafo G, e o μ é a função que associa os elementos. A figura 2.1 possui os elementos utilizados como referências para a metamodelagem [6], como:

- **Modelo de Referência**: dados 2 modelos, M1 e M2, caso ω de M1 seja igual a M2, M2 é o modelo de referência de M1. Alguns modelos podem ser seus próprios modelos de referência.

- **Relação de conformidade**: é a relação entre um modelo e seu modelo de referência. Ela indica que um modelo esta coerente com seu modelo de referência. Um modelo de referência é um tipo de determinado modelo. Um exemplo pode ser o metamodelo UML sendo o tipo de um modelo terminal, e o *Meta-Object Facility* (MOF) sendo seu modelo de referência, como ilustrado na figura 2.2.

- **Metametamodelo**: modelo esse que também é seu próprio modelo de referência. Um metametamodelo é a base referencial para si mesmo e para um metamodelo, assim como o metamodelo é para um modelo [17].

- **Metamodelo**: é um modelo, seus tipos estão contidos em um metametamodelo, possui a função de referenciar o modelo terminal. Um metamodelo é o modelo de uma linguagem de modelagem [7].

- **Modelo Terminal**: modelo criado em conformidade ao metamodelo, pode ser um cliente ou um produto por exemplo, de um sistema de um supermercado.

A Figura 2.1 ilustra o metametamodelo situado na camada M3, representando os conceitos para a criação dos metamodelos, situados na camada M2 e para os modelos terminais da camada M1 [6].

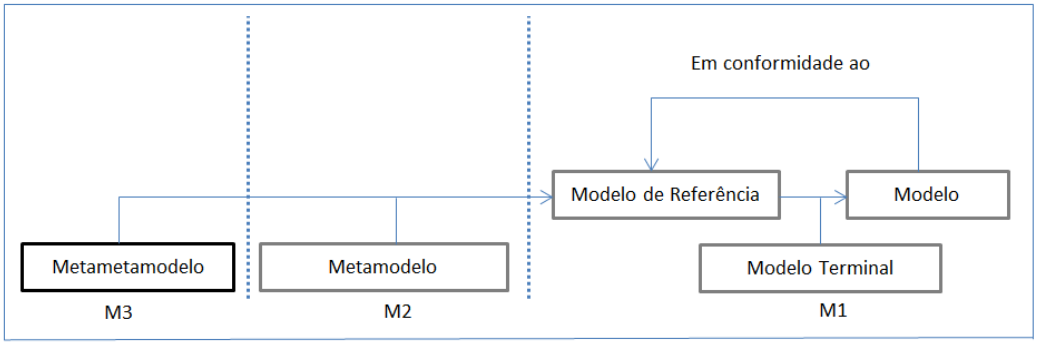

Figura 2.1: Relacionamentos da Modelagem [6]

Existem diferentes formatos para representar modelos, como o *Meta-Object Facility* (MOF), que além de servir de modelo para metamodelos é também seu próprio metametamodelo. Outro exemplo de metametamodelo é o *Ecore*, utilizado no *Eclipse Modeling Framework* (EMF). O EMF é um *framework* utilizado para geração de código e suporta a criação de ferramentas e de aplicações por modelos [30].

Um exemplo da utilização dos metametamodelos Ecore e MOF é ilustrado na figura 2.2. No lado direito o MOF, na camada M3, servindo de referência para o metamodelo UML situado na camada M2, e aos modelos terminais Produto e Cliente da camada M1, também nessa camada se encontram o Modelo de Referência e o Modelo.

Do lado esquerdo localiza-se o metametamodelo *Ecore*, que é seu próprio modelo de referência, e modelo de referência para a criação no exemplo, de uma Linguagem de Descrição de Componentes de Interfaces Ricas e também referência os modelos terminais Menu e Seção.

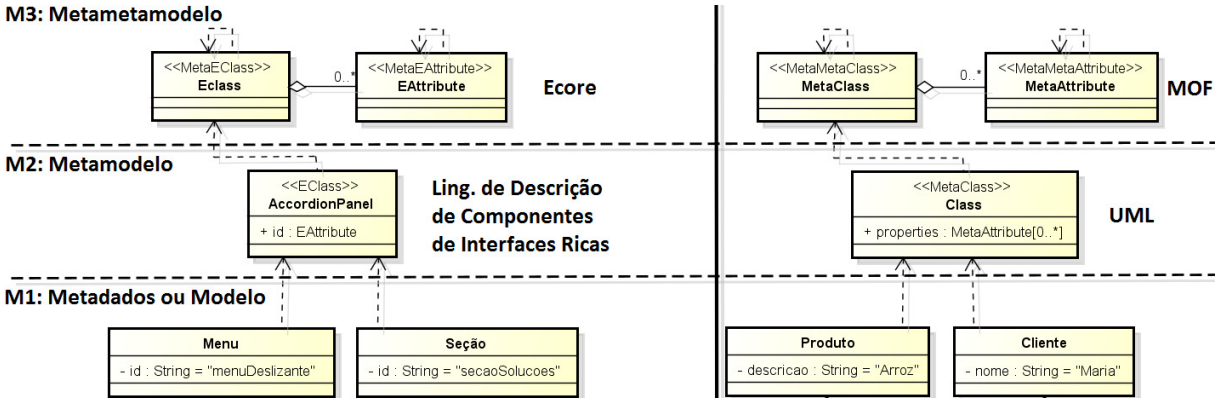

Figura 2.2: Exemplos de Metamodelagem.

2.1.2 Abordagens de Armazenamento de Modelos

Existem várias abordagens que possuem funcionalidades para manipular modelos, que possuem maneiras de armazenamento e processamento variadas. As mais utilizadas estão listadas a seguir.

2.1.2.1 Ecore

O formato para representação de modelos manipulado pelo EMF (*Eclipse Modeling Framework*) é o *Ecore* [9][30]. A figura 2.3 apresenta o subconjunto do *Ecore*, utilizado neste trabalho, estão contidos nele as *Eclasses* com a funcionalidade de representar as entidades do modelo e serem as classes do modelo, podem representar por exemplo, os imóveis de um sistema de imobiliárias. As *Ereferences* representam as relações entre as classes do modelo e os *Eattributes* que são valores simples de tipos básicos, podendo ser por exemplo os atributos nome e RG de um determinado usuário do sistema.

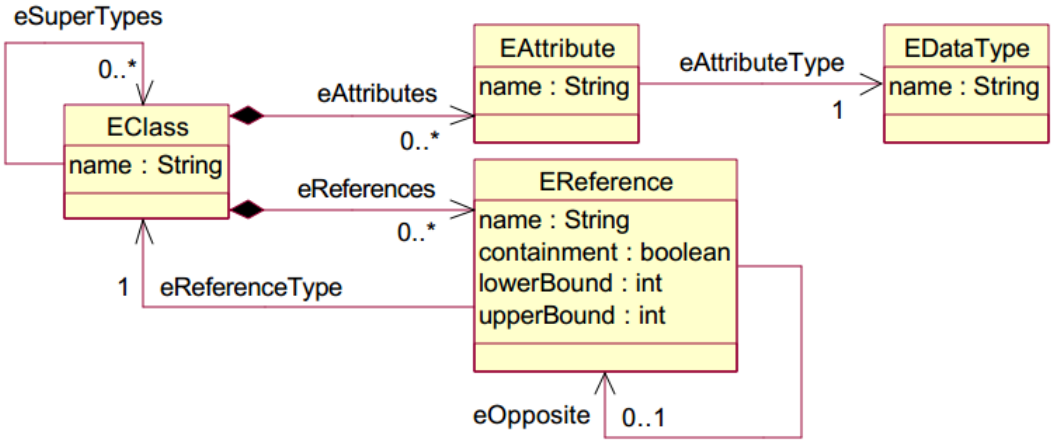

Figura 2.3: Metametamodelo Ecore. [15]

As *Eclasses* são compostas por *Eattributes*, os *Eattributes* por sua vez apontam para tipos simples como por exemplo, *EString* ou *EInteger*, e as *Eclasses* também podem se compostas por *Ereferences* para outras *Eclasses* [9]. Um exemplo de *Eattribute* de uma *EClass* é seu nome. No exemplo da figura 2.4, as *Eclasses* são as classes Movel1 e Movel2, os *Eattributes* são os atributos nomes e as referências da classe Movel2 são exemplos de *Ereferences*.

```
<?xml version="1.0" encoding="UTF-8"?>

<eClassifiers xsi:type="ecore:EClass" name="Movel">

    <eStructuralFeatures xsi:type="ecore:EAttribute" name="Nome"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>

</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Movel2">

    <eStructuralFeatures xsi:type="ecore:EAttribute" name="Nome"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>

    <eStructuralFeatures xsi:type="ecore:EReference" name="É" lowerBound="1" eType="#//Movel1"/>

    <eStructuralFeatures xsi:type="ecore:EReference" name="É" lowerBound="1" eType="#//Imovel2"/>

 </eClassifiers>
```

Figura 2.4: Exemplo de um Código XMI

O *Eclipse Modeling Framework* (EMF), *framework* desenvolvido pelo Eclipse org, é voltado para modelagem e facilidade de geração de código. A partir de uma especificação

do modelo descrito em XML Metadata Interchange (XMI) [20], linguagem essa derivada do XML sendo utilizada para especificação de modelos. O EMF fornece ferramentas e suporte *runtime* para produzir um conjunto de classes Java para o modelo. Para a representação de modelos no EMF se utiliza a linguagem de metamodelagem *Ecore* [9] [30][29].

Porém, o EMF possui processamento e armazenamento centralizado, limitando assim sua capacidade de transformar grandes modelos [4].

2.1.2.2 XML

Alguns padrões também foram criados para que houvesse interoperabilidade entre os sistemas, o XML é uma abordagem para armazenamento de dados, ele é um exemplo de formato criado para a padronização, um exemplo de seu uso é a interligações de banco de dados. O XML é composto apenas por nós, subnós e atributos. A figura 2.5 exibe um exemplo de arquivo XML, no qual se tem um nó denominado *Moveis*, esse nó possui dois subnós, um chamado *estante* e outro *cama*, ambos possuem um subnó denominado *cor*.

```
<?xml>
<Moveis>
        <estante codigo="0123">
                <cor>"marrom"</cor>
        </estante>
        <cama codigo="0321">
                <cor>"preto"</cor>
        </cama>
</Moveis>
```

Figura 2.5: Exemplo de um Código XML

Um outro tipo do XML é o XML *Metadata Interchange* (XMI), XMI é o padrão utilizado nos modelos *Ecore*. Em base desse modelo de padronização as distribuições são possíveis, levando em considerações os elementos do *Ecore*. Um exemplo de XMI é ilustrado na figura 2.4, onde estão representadas duas classes, *Movel* e *Movel2*. A classe *Movel* possui um atributo chamado *nome*, do tipo *EString*, a classe *Movel2* também um atributo chamado nome do tipo *EString* e outros 2 elementos do tipo *EReference*.

2.1.3 *CDO - Connected Data Objects*

CDO é um *Object-relational mapping* (ORM) de código livre. Possui a função de proporcionar de maneira simples e fácil a edição de modelos aos programadores, enquanto os modelos relacionais de banco de dados contam com uma fonte de dados JDBC no lado do servidor, o CDO proporciona por exemplo, o uso do *Hibernate* e *OODB-based* [19]. O CDO usa tabelas como sintaxe de seus dados, esses dados estão contidos em colunas quem compõem a tabela, a imagem 2.6 exibe um exemplo de tabela usada como sintaxe pelo CDO, nessa tabela estão contidas informações sobre *Moveis*, *Imoveis* e *Cores*.

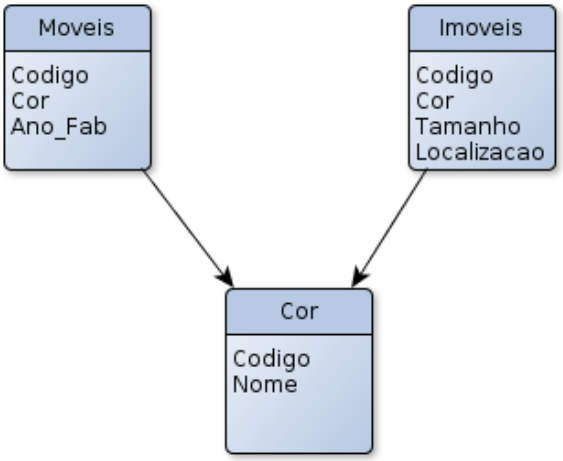

Figura 2.6: Exemplo da Sintaxe CDO

2.1.4 Morsa

Framework que utiliza o MongoDB como base de dados, o Morsa é um mapeador de objetos não relacionais, utiliza *JavaScript Object Notation* (JSON) como sintaxe para transferência de informações [31], esse *framework* possui a característica de ser um modelo de persistência destinado para a escalabilidade, uma de suas intenções é resolver o problema de estouro de memória do EMF ocasionado pelo largo tamanho dos modelos.

Outra de suas qualidades é a transparência na integração entre uma solução de persistência e o cliente [16].

A figura 2.7 exibe um exemplo de código JSON, sintaxe essa utilizada pelo Morsa. Nesse exemplo o elemento identificador do objeto é chamado “Aluno” e o atributo nome com valores simples como o João e o atributo disciplinas com valores multivalorados, por exemplo, banco de dados e LOO. Os atributos multivalorados são as principais diferenças, por exemplo, entre arquivos XML, esse tipo de atributo diminui o tamanho do arquivo, pois não há a necessidade de indicar o atributo para cada valor. Nesta figura esta contido 3 alunos com 2 disciplinas cada, as disciplinas são atributos multivalorados, por isso, estão entre colchetes.

```
{ "Aluno" : [
      { "nome": "João", "disciplinas": [ banco de dados, LOO ]  },
      { "nome": "Maria", "disciplinas": [ POO, LOO ] },
      { "nome": "Pedro", "disciplinas": [ banco de dados, estrutura de dados ] }
   ]
}
```

Figura 2.7: Exemplo de um Modelo JSON

2.2 Distribuição de dados

Com a quantidade de armazenamento crescendo de *megabytes* para *petabytes* em menos de 30 anos, os sistemas de armazenamento e processamento encontraram problemas para acompanhar esse crescimento, impossibilitando o trabalho com grande quantidade de dados [14]

2.2.1 *EMF Fragment*

Um *framework* que se preocupa com o problema de escalabilidade é o *EMF Fragment*, que tem a função de fragmentar grandes modelos de dados orientados a objetos, o *EMF Fragment* divide os modelos não em objetos individuais mas em um conjunto de elementos menores do modelo, do tipo chave/valor. Suas características de armazenamento e indexação são semelhantes as do EMF, já que seu uso é voltado para o EMF. Utiliza o CDO como seu ORM, e o Morsa para mapear os objetos não relacionais. A figura 2.8 exemplifica a fragmentação dos dados no *EMF Fragment*, nesse exemplo, a função A

possui uma referência para B, que esta na mesma divisão de C, foi criada uma cópia da fragmentação que contem a função B e C, resolvendo o detalhe da referência que existe de C para si mesma.

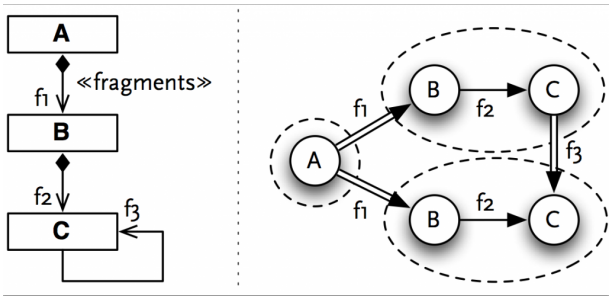

Figura 2.8: Exemplo da Fragmentação do *EMF Fragment* [23]

2.2.2 *MapReduce*

O *MapReduce* esta mais detalhado nesse trabalho, pois foi utilizado como *framework* de base. Ele é uma solução da Google para manipular 20 Pb por dia de dados, número esse que não passava de 100 Tb diariamente [5]. *MapReduce* é a solução mais popular atualmente para se manipular grandes quantidade de dados.

O *MapReduce* pode ser útil como no *Large Hadron Collider* (LHC), próximo a Genebra, o LHC é o maior acelerador de partículas do mundo, quando concluído irá produzir o equivalente a 15 Pb de dados por ano [2].

Outros modelos de sistemas distribuídos exigem que o programador defina funcionalidades como sincronização e compartilhamento de resultados, funções essas que não exigem serem definidas no *MapReduce*, permitindo ao programador se preocupar com outras funções do sistema.

O *MapReduce* é inspirado nos princípios da linguagem funcionalista, onde cada função é um valor, e pode ser passadas funções como parâmetros para outras funções, além de possuir funções especializadas para grandes quantidades de dados computacionais. Esses princípios necessitam de um tipo de entrada e saída. Para utilizar o *MapReduce*, o usuário deve configurar o *cluster*, indicando o *master* e os *slaves*, após esta etapa, o *cluster* deve

ser iniciado, inseri no HDFS a entrada, que pode ser, por exemplo, texto, XML, lista, entre outros e então executa as operações. O tipo da saída também deve ser indicado, e este terá uma chave e um valor "K,V" [22].

O *MapReduce* possui três fases: *Map*, *Reduce* e *Shuffle*. A fase *Map* tem a função de subdividir os dados de entrada. A fase *Reduce* agrupa chaves iguais geradas pela função *Map*. O *Shuffle* executa a ordenação das chaves e transfere os dados produzidos na saída do *Map*. Cada fase executada gera *tuplas* chave/valor, onde o valor é a soma dos valores das chaves semelhantes [33]. A arquitetura do *MapReduce* é baseada em um modelo *master/slave*, com um escalonador e mecanismos interligados como *split* dos dados de entrada [1]. A Figura 2.9 é um modelo do fluxo de dados do *MapReduce*.

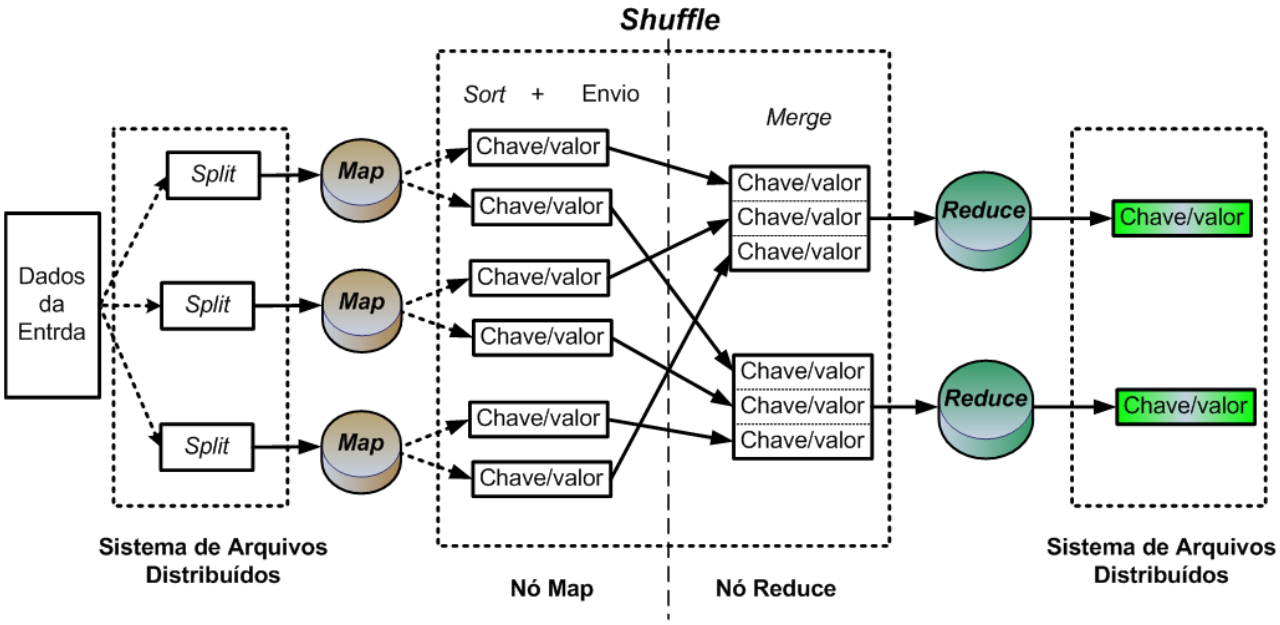

Figura 2.9: Modelo do Fluxo do *MapReduce* [33].

Há algumas implementações do *MapReduce*, como *Hadoop* utilizando a linguagem Java[27] [33], *DryadLINQ* para C# + *LINQ* [36], *Pydoop framework* para *Python* [21]. O *Hadoop* será mais detalhado, pois ele é a implementação escolhida para esse trabalho.

Uma das características do *Hadoop* é possuir um sistema de arquivos distribuídos [4], desse modo é possível dividir os dados em várias partes, aumentando assim a quantidade de dados possíveis a serem trabalhados e copiar essas partes em outros *clusters*, para que

seja possível uma recuperação caso ocorra falhas. O *Hadoop*, particiona cada tarefa em várias sub-tarefas de mesmo tamanho. O modo de controle do *Hadoop* é chamado de *jobTracker* (*master*), ele é responsável em distribuir cada subtarefa em máquinas *slaves*, intituladas por *taskTracker*.

Para que os dados sejam divididos, o *Hadoop* necessita de algoritmos de distribuição que são a base para a computação distribuída [1].

Um exemplo simples para sua utilização é um contador de palavras, na fase *Map* as palavras são distribuídas por linhas, onde cada linha é enviada para uma máquina que pode, as quais podem conter uma ou mais linhas, essas linhas podem também serem clonadas em outras máquinas para garantir maior segurança e confiabilidade. Uma máquina pode receber mais de uma linha. Na fase *Shuffle* as palavras semelhantes são agrupadas e redistribuídas para então serem contabilizadas na fase *Reduce*, onde cada máquina faz a operação correspondente, nesse caso cada máquina somaria a quantidade de palavras iguais gerando em suas saídas tuplas, essas são reduzidas novamente até todos os resultados estarem em apenas uma tupla no *host* principal. A figura 2.10 é um exemplo de uma operação *WordCount*, onde dados, no caso texto são inseridos no diretório *Input* dentro HDFS, divididos pelos *Splits* padrões do *MapReduce*, e mais uma vez ocorre a divisão na fase *Map*, a fase *Shuffle* os organiza para então os *Reduces* concluírem a contagem das palavras e armazená-los no diretório *Output* localizado no HDFS.

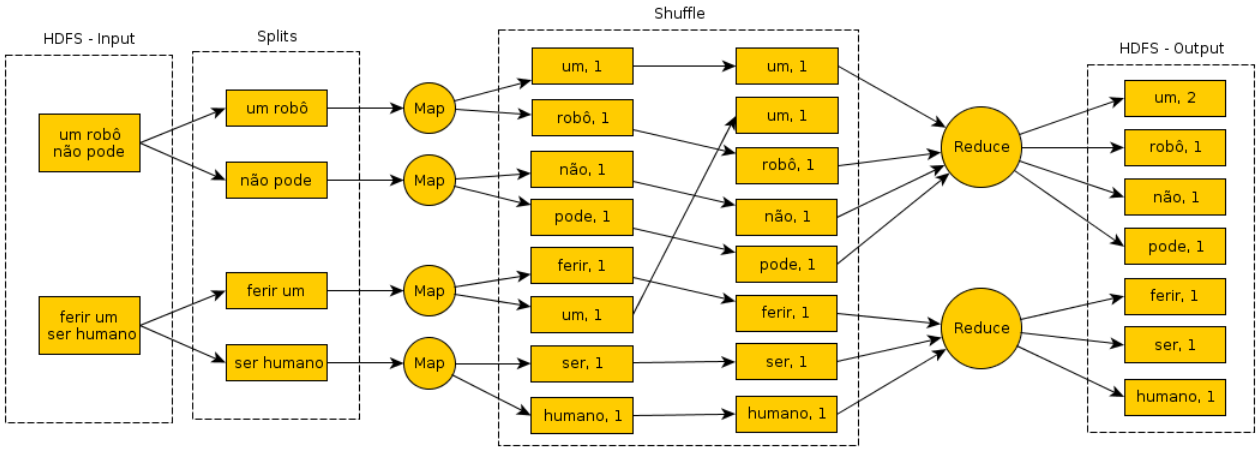

Figura 2.10: Fluxo do *WordCount*

2.2.3 Formatos de Entrada

Existem algoritmos de distribuição com vários formatos para o *MapReduce*, o formato XML esta descrito com mais detalhes, pois será usado no nosso trabalho, alguns dos formatos estão listados a seguir:

- **Texto**: formato de entrada e saída focando em arquivos de texto sem formatação específica. Nesse formato a leitura do arquivo é executada linha a linha [34].
- **Banco de Dados**: o *MapReduce* também possui entradas para banco de dados, como o HBase, um banco de dados distribuído da *Apache Fundation*, esse banco é voltado para armazenamento de grande quantidade de dados [18].
- **XML**: um exemplo de trabalho com arquivos XML é o *Apache Mahout*, desenvolvido pela *Apache Fundation*, o projeto tem como objetivo possibilitar o trabalho com grandes arquivos XML. Porém, o XML é composto por *TAGs*, a distribuição dos arquivos XML não se preocupa com o acoplamento, já que sua divisão é por *TAG* e essas não são dependentes umas das outras [8], a distribuição do *Mahout* concentra-se em encontrar o início da *tag* e o final da mesma. Caso haja:
  - Uma *TAG*: O *Mahout* irá encontrar seu início e fim, contemplando tudo o que há entre eles.
  - Uma *TAG* acoplada a outra: A *TAG* presente dentro da *TAG* principal não é identificada e apenas irá contemplar a *TAG* inicial.
  - Uma *TAG* seguida de outra: Duas divisões irão ocorrer, considerando separadamente cada *TAG*.

2.3 Sumário

O EMF-Fragment assim como o EMF utiliza o Ecore como seu formato de representação de modelos, formato esse que possui *EClass*, *EAttribute*, e *EReference*

para representar as possíveis classes, atributos e referências de um modelo. Tanto o EMF-Fragment quanto o EMF, fornecem ferramentas para produzir um conjunto de classes Java a partir de um modelo XMI. O EMF-Fragment manipula modelos de maneira distribuída, para isso, ele utiliza o CDO como seu mapeador de objetos relacionais, e o Morsa para mapear objetos não relacionais, possibilitando assim, manipular grandes modelos, caso esse que não ocorre com o uso do EMF, que trabalha de maneira centralizada, não permitindo o escalonamento de memória ou processamento, características necessárias para a manipulação de grandes modelos. O CDO é ser um ORM capaz de proporcionar de maneira simples aos programadores a edição de modelos além de proporcionar, por exemplo, o uso do Hibernate e OODB-based como ilustrado na figura 2.6. O Morsa, que utiliza o MongoDB como base de dados, possui o JSON como sintaxe para transferência de informações, o formato JSON possui em sua composição objetos, esses objetos possuem elementos que podem conter valores monovalorados e multivalorados, esses elementos podem ser simples ou serem outros objetos. O Morsa também facilita a transparência na integração entre a persistência e o cliente. Porém, esses *frameworks* não são capazes de manipular modelos de grande porte, sem depender de outros *frameworks*, impossibilitando modificações de baixo nível.

O *MapReduce* é capaz de manipular variados formatos de entrada, com características e tamanhos diferentes, como texto, banco e XML. O XML foi o formato de entrada que mais se aproximou do foco desse trabalho que utiliza modelos XMI. A grande diferença a ser tratada é a dependência que há entre os elementos XMI, característica essa não encontrada em arquivos XML. Essa dependência é o ponto chave para a distribuição de modelos utilizando o *MapReduce*, alterando não apenas a velocidade mas também o tamanho do modelo que poderá ser manipulado.

CAPÍTULO 3

AVALIAÇÃO DE SISTEMA BASEADO EM *MAPREDUCE* PARA CARREGAMENTO DE MODELOS

Neste capítulo será apresentada, uma abordagem que integra o *Mapreduce* com MDD, a implementação de 4 técnicas de distribuição usando o *framework Mapreduce* e uma avaliação das implementações com modelos de tamanho entre 180 Mb e 307 Gb.

3.1 *Driver* para divisão de modelos de grande porte

Esse trabalho implementou um *driver* para divisão de modelos de grande porte, para manipulação dos mesmos de maneira distribuída. Esse *driver* possibilita dividir um modelo considerando as características encontradas e inspiradas em Sheidgen [25], capazes de executar operações com maior velocidade na maioria dos modelos encontrados em Yang [35] e Zubow [24]. A figura 3.1 ilustra a adaptação que foi criada para o Hadoop, possibilitando que modelos sejam formatos de entrada. Após a criação dos *Splits*, o *driver* é acionado, criando os *maps* de acordo com o tipo de divisão escolhida. Os elementos, *Splits*, DriverDivisãoModelos e Resultado, estão contidos em caixas retangulares para sinalizar que estão no sistema de arquivos distribuído do Hadoop (Hadoop Distributed File System - HDFS).

A figura 3.1 exibe a implementação do DriverDivisãoModelos no *MapReduce*, semelhante a figura 2.8 onde esta ilustrado o *MapReduce*, com a inserção do DriverDivisãoModelos na transição entre a criação dos *Splits* e dos *Maps*. No exemplo dessa figura, o modelo que esta no *Input* do HDFS passa pelas seguintes fases:

- O modelo é primeiramente dividido pelos *Splits* do Hadoop, onde por padrão os

arquivos são divididos em arquivos menores contendo 64Mb.

- Após, os arquivos são novamente divididos pelo *Driver*, levando em consideração o método escolhido para a divisão.
- Então, a fase *Map* é efetuada, em seguida ocorrem as operações na fase *Shuffle*, por fim, as funções *Reduce*.
- Concluindo as operações, os dados são salvos novamente no HDFS.

As divisões executadas nos modelos, são feitas baseadas em critérios extras, que serão descritos na próxima sessão, os *Splits* dividem os dados apenas por linhas, modo esse inadequado para dividir os modelos, o DriverDivisãoModelos adiciona essa adaptação levando em consideração o metametamodelo do modelo à ser trabalhado.

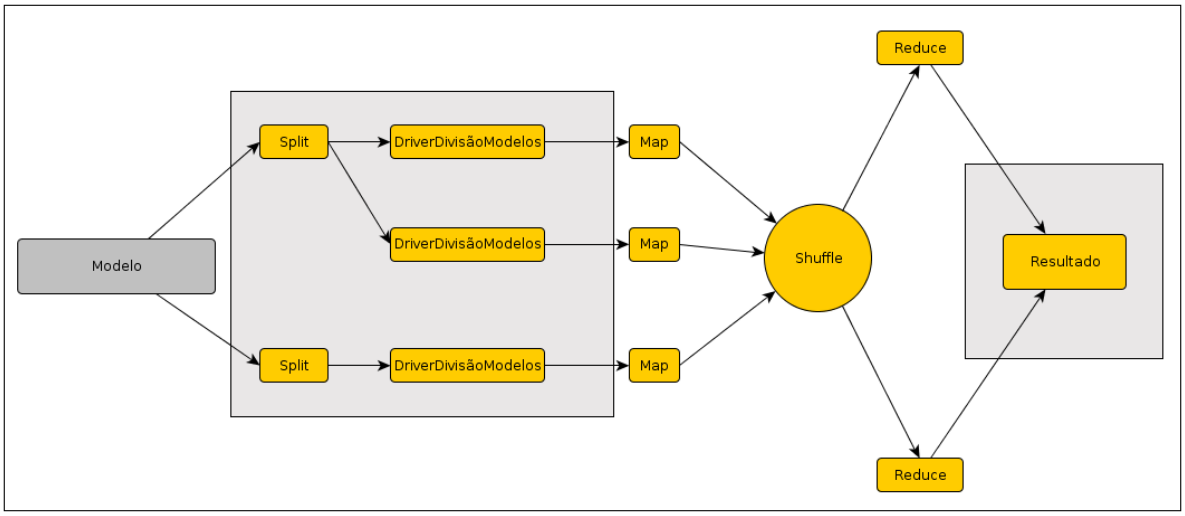

Figura 3.1: Localização do Driver para Divisão de Modelos no MapReduce

3.2 Distribuição de Modelos

Foi escolhido o *XMLInputFormat* como implementação de base para este trabalho. Porém, na manipulação de modelos, um fator que deve ser levado em consideração é a dependência dos seus elementos, como as referências entre as classes, caso que não ocorre nas *TAGs* dos arquivos XML, pois são independentes uma das outras.

Para realizar trabalhos com modelos de grande porte, alguns métodos de divisões foram escolhidas e desenvolvidas, onde a principal característica é a criação das divisões de acordo com uma constante, valor esse, que varia de acordo com o método escolhido para a divisão, também foi necessário alterações na função *Map* da operação que será executada, para que essa função encontre corretamente as *TAGs* iniciais e finais dos arquivos do tipo Ecore. Foram desenvolvidas várias formas de divisões, pois, diferentes modelos necessitam de diferentes métodos de divisões para um melhor desempenho [25].

Para isso leva-se em consideração as dependências entre os seus elementos, onde dado um modelo M, uma divisão é um modelo MS onde $MS \subset M$. O cálculo dessas divisões é executado em um pré processamento, utiliza-se uma pequena parte do modelo contendo por exemplo, 5 Mb, para o cálculo e em cima do resultado é estipulado um valor para todo o modelo, essa operação é executada de modo centralizado para melhor desempenho, justificado pelo tamanho pequeno da parte do modelo.

3.2.1 Média das Referências

Uma dos métodos de divisões implementados ocorre levando em consideração os elementos da camada M3, onde a quantidade de classes e referências são utilizadas. Nessa técnica é calculada a média de referências por classes:

$(A = \frac{NR}{NC})$, onde:

- A é a média calculada, esse valor é usado para a criação das divisões, ele defini quantas classes contemplarão cada divisão do modelo.
- NR é o número de referências encontradas no modelo.
- NC é a quantidade de classes que o modelo em questão possui.

Caso optado esse método para a divisão, de um modelo com 10 classes, onde elas possuem 20 referências entre si, cada divisão será composta por 2 classes.

3.2.2 Média dos Tipos

Há casos onde modelos, podem obter resultados diferentes com uma divisão mais específica, como em um modelo que possui várias cidades e estados, uma operação de transformação, por exemplo, pode obter melhor resultado se as classes de mesmo tipo estiverem nas mesmas divisões. Considerando o caso citado, as cidades devem ficar em divisões contendo apenas cidades e todos os estados em outras divisões. Sendo a média:

$M = (\frac{NC}{NT})$,onde

- NC é o número de classes
- NT é o número de tipos identificados,

Para essa identificação, padronizou-se o início do nome da classe como sendo o tipo, por exemplo, a classe CidadeCuritiba e CidadePontaGrossa. A quantidade de caractere que identifica o tipo da classe pode ser alterado, no exemplo das cidades, 6 caracteres é o máximo necessário. Com a utilização da média chega-se em número aproximado para a melhor divisão, onde a maior quantidade de classes com dependências estarão no mesmo *Map*. Essa divisão leva em consideração os elementos da camada M1 da arquitetura de modelagem.

Escolhendo esse método, um modelo com 20 classes representando cidades, e 20 classes representando estados, será dividido em 2 frações, cada uma contendo 20 classes.

3.2.3 Tamanho do Modelo

Outro método para a criação das divisões é levando em consideração o tamanho dos modelos. Esse valor é obtido através da divisão do modelo pelo tamanho sugerido:

$N = (\frac{TM}{TS})$, onde:

- N é o número de divisões que serão criadas para um determinado modelo.

- TM é a tamanho do modelo em operação.
- TS é o tamanho sugerido para a melhor divisão possível.

Por exemplo, um modelo de 1024 Mb, caso sejam escolhidas divisões contendo 64 Mb, serão criadas 16 divisões onde, caso cada classe possua 1 Mb, todas as 16 divisão irão contemplar 64 classes.

3.2.4 Uma Classe por Divisão

Outra técnica para a distribuição de modelos, contempla apenas uma classe em cada divisão, criando assim um maior número de divisões, porém, com um menor tamanho.

$N = (\frac{NC}{1})$, onde:

- NC é o número de classes
- 1 é o número de classes em cada *Map*

Seu uso, por exemplo, em um modelo de 3000 classes, resultará em 3000 divisões, contendo cada divisão, 1 classe.

3.2.5 Algoritmo de implementação das técnicas de divisão

O algoritmo 1 contém o pseudocódigo do *driver*, onde no início são indicadas as *tags* iniciais e finais, após, é indicado o número de classes que contemplarão cada divisão. Percorre-se o arquivo até encontrar a *tag* inicial; adiciona ela em um *Buffer* de saída; continua percorrendo o modelo até encontrar a *tag* de saída; adiciona todo o texto percorrido e a *tag* de saída no *buffer*; incrementa o contador de classes no *buffer* e verifica se ele atingiu o número de classes indicadas por divisão. Caso tenha atingido, envia os dados do *buffer* para a fase *Map*; limpa o *buffer* e o contador de classes; e continua percorrendo o modelo até que chegue ao seu fim.

```
var StartTag, EndTag, OutputBuffer : String;
var P : int;
var ExistsNextToken : boolean;
 StartTag <- "StartTagXMI";
 EndTag <- "EndTagXMI";

   /* P recebe a constante contendo a quantidade de classes, que
 irão compor cada divisão */

 P <- ClassDivisionNumber;
 While (ExistsNextToken) do
       While (P > 0 or !ExistsNextToken) do

              /* Ler o arquivo até encontrar o StartTag */

              ReadUntilFind (StartTag);
              OutputBuffer <- StartTag;
              ReadUntilFind (EndTag);

              /*Inseri todo o texto lido até encontrar a EndTag, no
              OutputBuffer */

              OutputBuffer <- All Text Read Until EndTag;
              P --;
       End While;
 End While;
End.
```

Algoritmo 1 - (Driver XMIInputFormat)

A figura 3.2 exibe o fluxograma do *driver XMIInputFormat*.

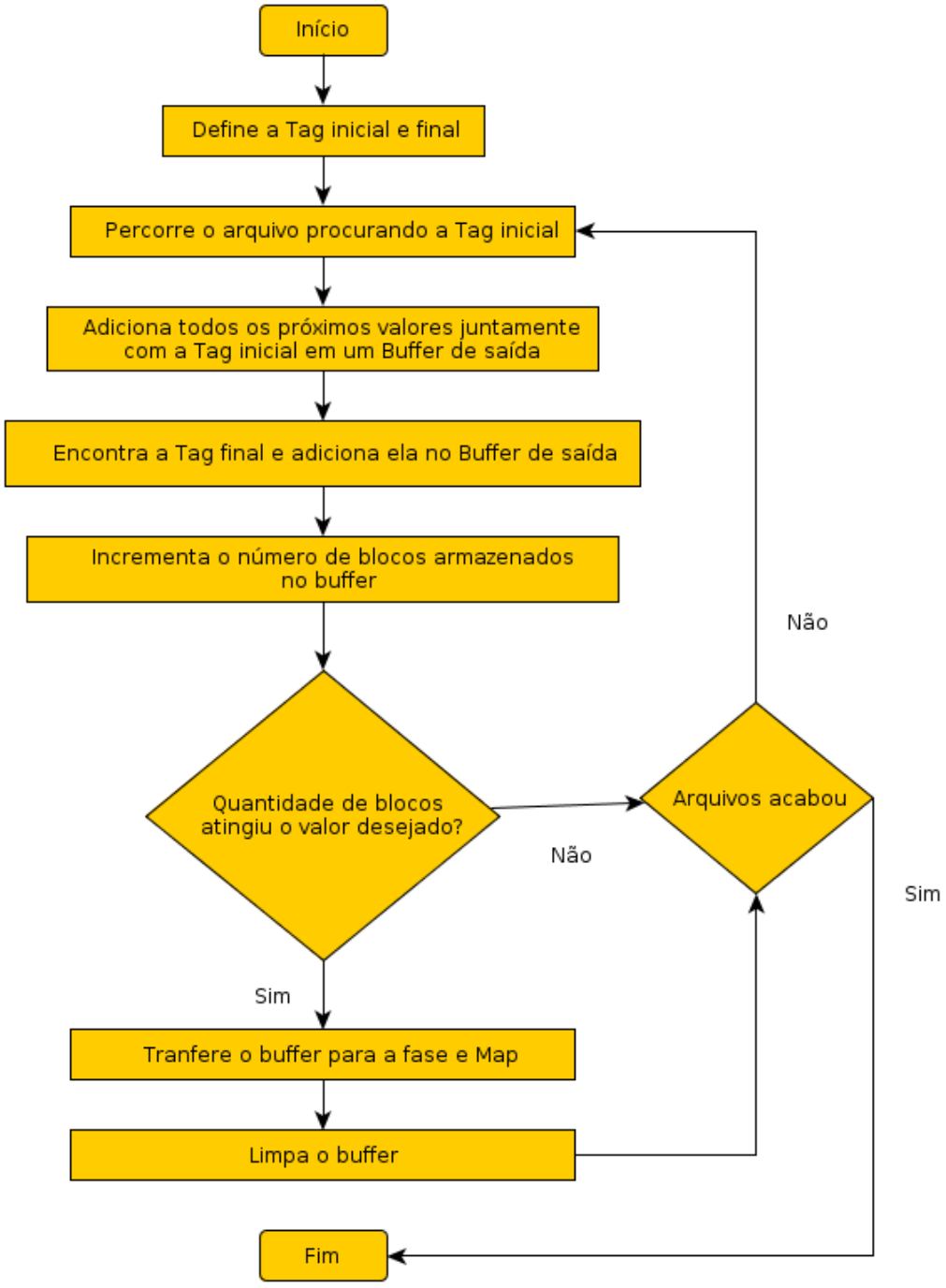

Figura 3.2: Fluxograma do Driver XMIInputFormat

3.3 Avaliação

Com a criação do *driver* para divisão de modelos (*XMIInputFormat*), foi possível a execução de vários testes, gerando resultados diferentes de acordo com a configuração escolhida no *driver* e do *cluster*. Todos os resultados foram obtidos por meio de testes em máquinas que possuem as seguintes configurações:

- Processador - Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz
- Arquitetura - x86 64
- Sistema Operacional - 3.2.0-40-generic #64-Ubuntu SMP GNU/Linux
- Memória Ram - 4Gb

Foram utilizados modelos XMI reais extraídos de um programa java teste e modelos XMI gerados especificamente para os testes, ambos do tipo Ecore, e duplicados várias vezes para possuírem tamanhos variados. O *Hadoop* foi configurado para replicar 2 vezes os dados, para que caso, ocorra erro em um nodo, haja os dados em outro nodo para recuperação. Nesse caso, novamente é replicado esses dados em outro nodo. Quanto maior o número de replicações, maior a segurança e tempo necessário para as operações. Outra configuração do *Hadoop* foi a criação de seus *splits* com 64Mb.

Como não foram encontrados modelos grande o suficiente, utilizou-se um modelo extraído de classes Java, esse modelo foi duplicado várias vezes, para que fosse obtido variações de tamanhos, porém com modelos semelhantes, foram também criados modelos para que tivéssemos os exatos tamanhos escolhidos para os testes.

A imagem 3.3 ilustra uma pequena parte de um modelo utilizado para os testes, nesse exemplo, caso optado utilizar o método Média das Referências, as classes seriam divididas em 2 grupos com 2 classes em cada, *Movel1* e *Movel2* em uma divisão, *Imovel1* e *Imovel2* em outra.

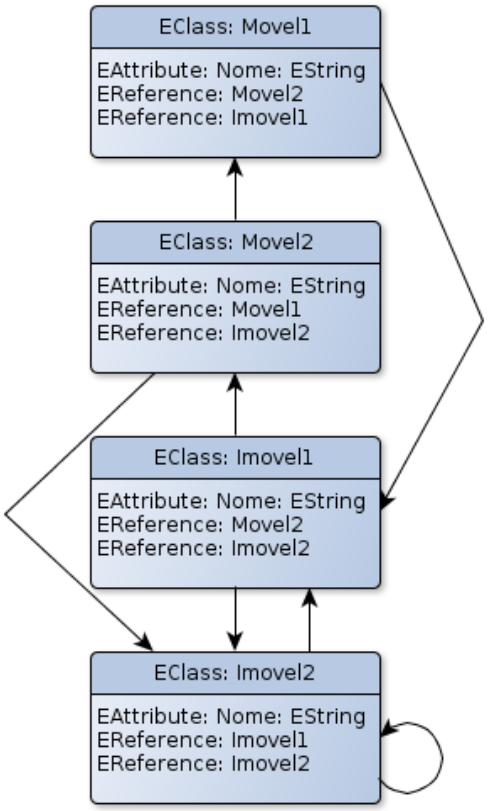

Figura 3.3: Exemplo da parte de um modelo utilizado no teste

Foram executados testes centralizados e distribuídos com uma, três e cinco máquinas, números esses escolhidos pela infraestrutura encontrada. Foram utilizados modelos variando de 180Mb a 307Gb. Foram utilizadas os 4 métodos de distribuição citados nesse trabalho:

- Média de Referências - Resultou uma média de 6 classes em cada divisão.
- Média de Tipos - Resultou uma média de 30 classes por divisão.
- Tamanho do Modelo - Executamos testes com 8Mb, 16Mb, 32Mb e 64Mb.
- Uma Classe por Divisão - Executamos testes onde cada divisão possuiu apenas uma classe.

Os testes distribuídos foram uma contagem das classes dos modelos, foram executados mais de 1000 testes, variando proporcionalmente a quantidade de testes com o tamanho

do modelo utilizado.

A figura 3.4 ilustra 4 testes executados de maneira centralizada, assim houve uma base de valores e limites, para comparar com os resultados do modo distribuído. As barras que estão a esquerda são a contagem dos elementos de um modelo com 180Mb e outro com 720Mb, obtendo os resultados de 8 segundos para o menor e 40 para o maior. Também foi executada uma operação de transformação nesses modelos, obtendo tempos de 93 e 435 segundos para os modelos, como indicados pelas barras que estão a direita. Os testes de transformação para modelos maiores não foram completados devido a falta de infraestrutura dos equipamentos.

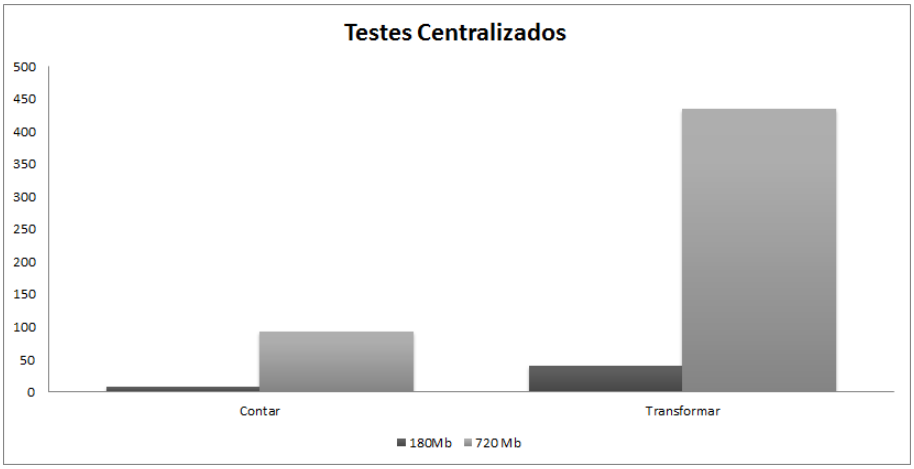

Figura 3.4: Testes centralizados

A figura 3.5 contém um teste centralizado que obteve um tempo de 8 segundos e exibe 4 testes distribuídos executados com 5 máquinas em um modelo de 180Mb, obtendo tempo de 20 segundos para os testes com 1 e 6 classes por divisão, 21 segundos para as divisões com 30 classes e 26 segundos para 30 classes por divisão. Testes com 1 classe por divisão resultaram em 57052 divisões, com 6 classes 9509 divisões, 30 classes resultou em 1902 divisões e os testes contendo 8Mb de classes resultaram em 11 divisões. Esses tempos mostram a desvantagem no trabalho distribuído com modelos de tamanho inferior a 700 Mb.

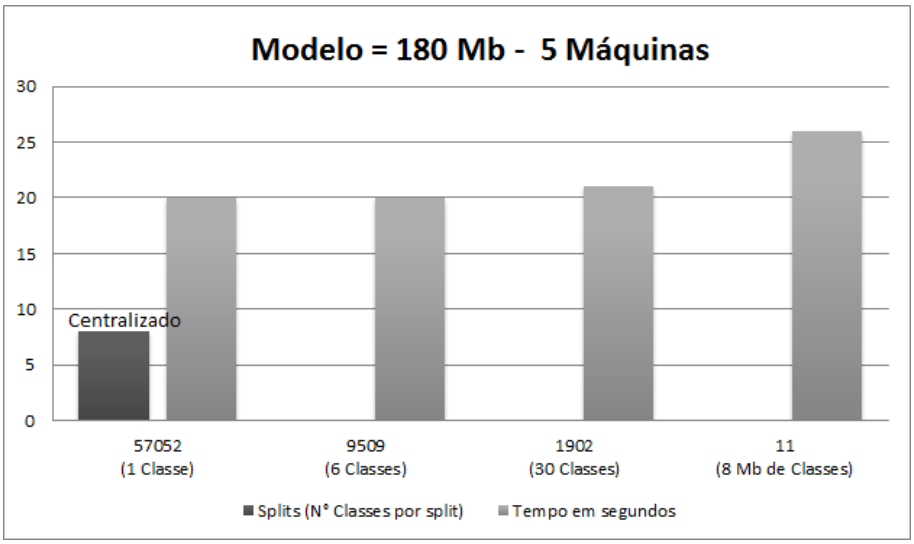

Figura 3.5: Modelo de 180Mb en 5 máquinas

A figura 3.6 exibe um teste centralizado com tempo de execução de 40 segundos. Essa figura contém também 4 testes distribuídos com tempos de 28 segundos para testes com uma classe por divisão, 29 para testes com 6 e 30 classes em cada divisão e 30 segundos para testes contendo 8Mb de classes. O teste com 1 classe resultou em 228208 divisões, com 6 classes obteve 38035 divisões, 30 classes resultou em 7607 divisões e o teste com 8Mb de classe em cada divisão possui 23 divisões. Nesses testes a distribuição resultou em um tempo favorável se comparada ao modo centralizado, que além de obter um tempo maior, resultou erros de estouro de memória em algumas execuções.

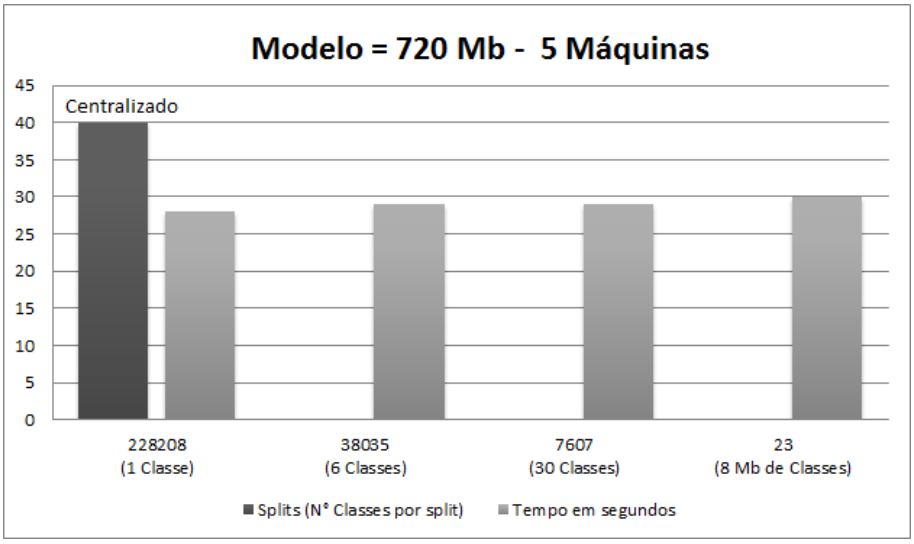

Figura 3.6: Modelo de 720Mb processado em 5 máquinas

Estão ilustrados na figura 3.7 testes efetuados em um modelo Ecore de 1Gb de tamanho, com divisões contendo de uma classe até um montante de classes suficientes para produzir divisões de 64Mb. Esses testes foram executados em uma máquina simulando um *cluster* e em *clusters* contendo 3 e 5 máquinas. Os resultados dessa comparação demonstraram uma variação pequena em comparação aos *clusters* de 3 e 5 máquinas, com tempo menor no *cluster* contendo maior número de máquinas, já os valores dos testes em apenas uma máquina simulando um *cluster* foram desproporcionais ao outros 2, explicado por exemplo, pelo fato dessa única máquina efetuar além dos papeis de *slave* o papel do *master*, cujo nos outros dois *clusters* o *master* é executado em uma outra máquina. O teste com divisões de 64Mb resultou em alguns erros, que o próprio Hadoop conseguiu corrigir e concluir, porém com um tempo muito superior se comparado com os demais. Testes com divisões maiores não puderam ser concluídos devido a alta ocorrência de erros em ambos os testes.

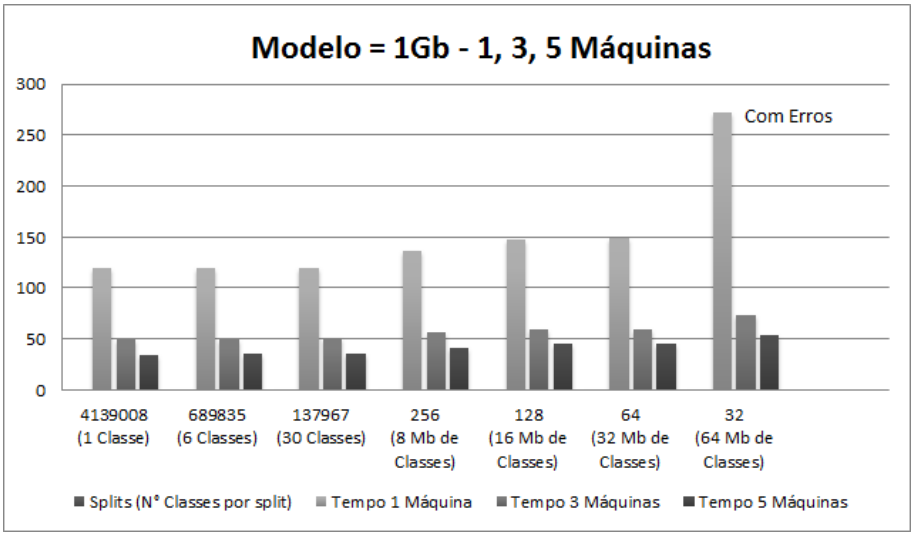

Figura 3.7: Modelo de 1Gb processado em 1, 3 e 5 máquinas

Os testes exibidos na figura 3.8 foram executados com um modelo Ecore de 10Gb. Foram executadas simulações contendo 1 classe por divisão, gerando 41390080 divisões, 6 classes produzindo 6898346 divisões, 30 classes dividiu o modelo em 1379669 divisões e 8 Mb, 16Mb, 32Mb e 64 Mb gerando um montante consequentemente de 2560, 1280, 640 e 320 divisões.

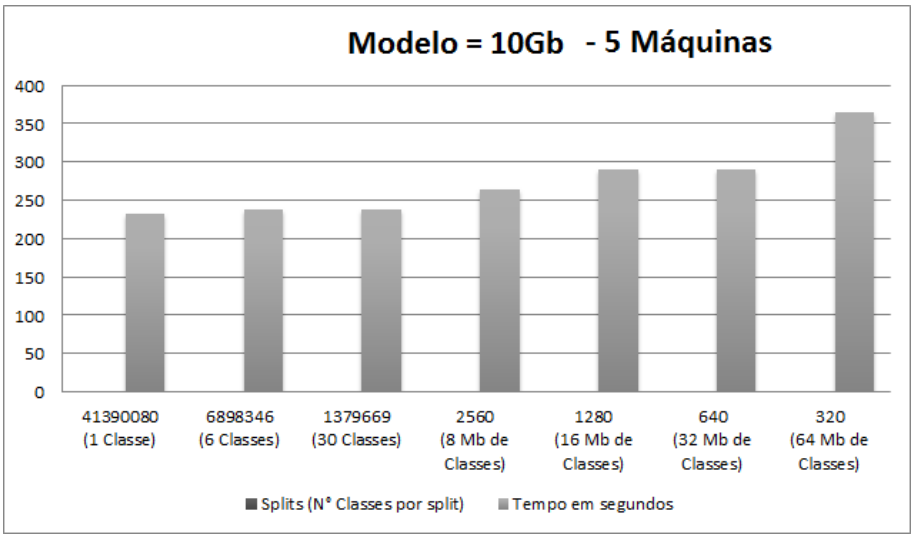

Figura 3.8: Modelo de 10Gb processado em 5 máquinas

Os resultados exibidos na figura 3.9 são semelhantes ao da figura 3.8, porém, foram executados com um modelo de 30Gb, gerando divisões 3 vezes maiores do que aos do modelo de 10Gb, os valores dos tempos também foram proporcionais. Os testes executados com divisões contendo classes suficientes para gerar 64Mb em cada divisão gerou um valor de tempo maior relativamente comparando com os demais testes.

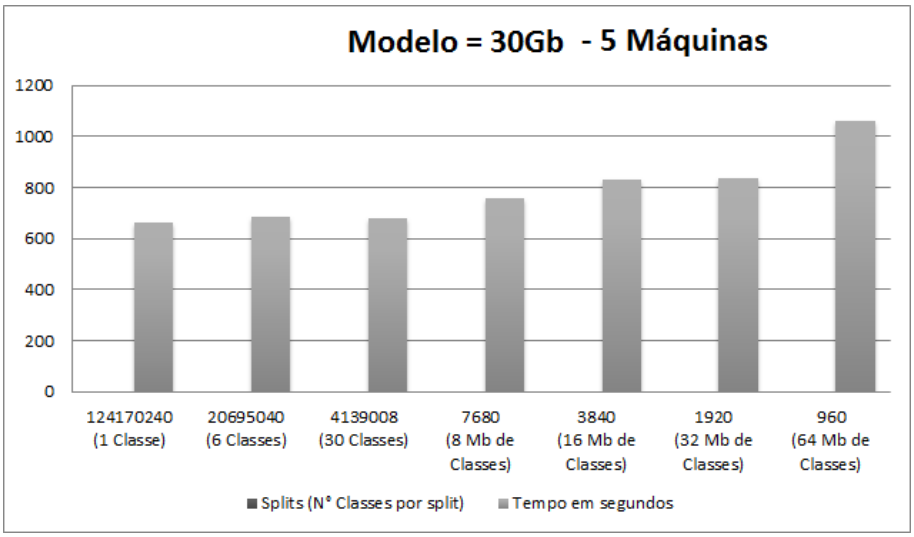

Figura 3.9: Modelo de 30Gb processado em 5 máquinas

A figura 3.10 exibe 5 testes, ambos foram executados com divisões contendo apenas uma classe, nesta técnica é gerada a maior quantidade de divisões possíveis para o modelo. O modelo de 1Gb gerou 4139008 divisões, o modelo de 10 Gb gerou 10 vezes mais, o de

30 gerou 30 vezes, o de 100Gb gerou 413900800 divisões e o modelo com 307Gb gerou 1270675456 divisões. A coluna em cinza claro exibe os tempos dos testes nos modelos de 1Gb, 10Gb, 30Gb, 100Gb e 307Gb, cujo tamanho foi o maior que as máquinas conseguiram gerar. Os resultados mostram que entre 10 Gb e 100Gb, quanto maior for o modelo menor é a porcentagem de tempo utilizado, porém no teste com o modelo de 307Gb essa porcentagem volta a subir, não permitindo assim, afirmar que, quanto maior o modelo, menor será o tempo de suas operações proporcionalmente.

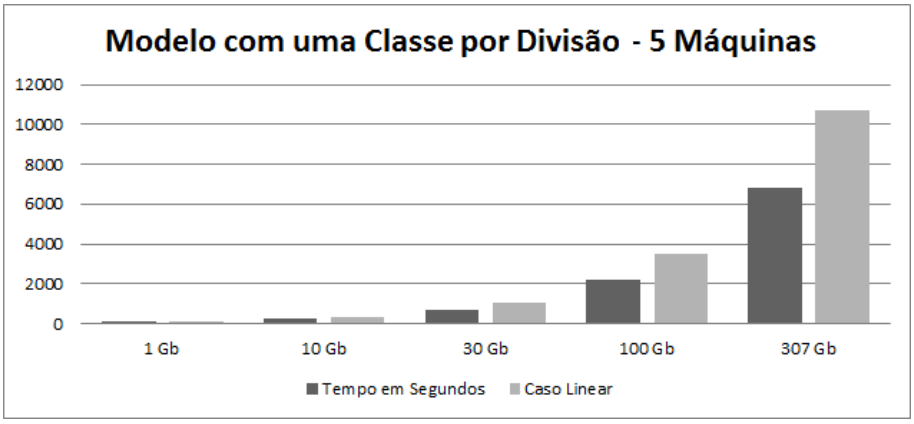

Figura 3.10: Modelos processado por divisões contendo apenas uma classe

3.3.1 Comparação dos Resultados

Há uma deficiência no trabalho centralizado com grandes modelos, sua grande exigência de memória e tempo de processamento torna muito custosas suas operações. A figura 3.4 exibe que há diferença grande do tempo necessário para contar (93 seg) e transformar (435 seg) um modelo de 720 Mb, modelos maiores ocasionaram estouro de memória.

Por outro lado, o trabalho com modelos pequenos apresentou ser mais custoso de maneira distribuída, testes centralizados exibidos na figura 3.5 mostraram um tempo, na contagem das classes de um modelo de 180Mb, inferior aos testes distribuídos utilizando 5 máquinas, a menor diferença apresentou a necessidade de tempo maior que o dobro para o modo distribuído.

Os resultados obtidos nos testes com um modelo de 720Mb foram satisfatórios para o modo distribuído. De maneira centralizada foi necessário 40 segundos para a contagem das classes, tempo esse que chegou a ser de apenas 28 segundos para o melhor caso de maneira distribuída, porém, utilizando 5 máquinas.

A figura 3.7 exibiu um comparativo de testes executados com 1, 3 e 5 máquinas, concluiu-se que há uma grande diferença nos trabalhos com 1 e 3 máquinas, caso que ocorre com menor magnitude em relação a testes executados com 3 e 5 máquinas, valores esses que poderiam ser diferentes devido a não utilização da rede nos testes, porém o fato de que utilizando apenas uma máquina, tanto os papeis dos *slaves* quanto os do *master* são executas pela mesma, resultou em uma grande diferença.

Os resultados mostraram que as divisões contendo apenas uma classe são mais eficientes, podendo ser mais que 35% veloz em relação aos demais métodos de divisão dos modelos. Os testes executados utilizando os métodos, Média de Classes e Média de Tipos, resultaram valores parecidos, decorrentes de que o teste executado foi um *Class-Count*, os teste utilizando o método, Tamanho do Modelo, resultaram vários problemas de estouro de memória.

Com esses dados conclui-se que com a utilização do *driver* XMIInput, para a distribuição dos modelos é possível, tendo ganhos significativos por exemplo, no tamanho máximo do modelo a ser executado em comparação, a maneira centralizada. A quantidade de computadores inseridos no *cluster* que efetuará os testes resulta em uma variação de tempo significativa, o tempo gasto nos trabalhos com modelos variando seu tamanho entre 10 e 300 Gigabyte é proporcional ao tamanho do modelo, sendo em média 36% menor em uma relação proporcional ao modelo de 1Gb. As operações executadas utilizando o método de apenas uma classe por divisão, obteve resultado melhores, porém, isso pode ter sido ocasionado por não haver dependência nas referências das classes em um teste *ClassCount*.

CAPÍTULO 4

CONCLUSÃO

Neste trabalho foi apresentado uma avaliação e implementação de *MapReduce* para carregamento de modelos. A partir dele é possível a divisão e manipulação de modelos de grande porte de modo distribuído utilizando o *Hadoop* e seu sistema *HDFS*.

Esse *driver* contempla 4 métodos de divisão para modelos Java de grande porte, esses métodos são baseados no tamanho do modelo, tipos e quantidade de seus elementos. Porém, ele não usa diretamente a API do EMF, fazendo necessário, adaptações para que as ferramentas atuais possam utilizá-lo.

Foram encontradas dificuldades, para obter modelos apropriados para os testes, o maior modelos Java encontrado possui apenas 180 Mb, para gerar modelos maiores, foi duplicado este modelo, também foram criados modelos Java variando de 1 à 307 Gb.

Os testes mostraram, que o *XMIInputFormat* suporta modelos. Em operações simples, como a contagem dos elementos de um modelo, o método de divisão única, onde cada divisão contempla apenas uma classe, foi mais eficiente nas operações *ClassCount*, pelo fato de não haver dependência entre as classes.

Com os resultados obtidos, conclui-se que o *driver* é capaz de carregar modelos de grande porte, foram obtidos resultados satisfatórios com modelos de até 307 Gb, valor esse mostrando ser possivelmente superado com o aumento de máquinas.

4.1 Trabalhos Futuros

Neste trabalho foi detectada uma deficiência na criação das divisões, no momento em que o próprio *Hadoop* às cria, já que com isso pode ocorrer uma divisão que corrompa

uma futura divisão do *driver*. Um trabalho futuro para operações que necessitam da integridade total dos dados, é o tratamento desse erro no *XMIInputFormat*.

BIBLIOGRAFIA

[1] J. C. S. Anjos. *MapReduce Unavailability-Aware para Computação Intensiva em um Ambiente Desktop Grid*. Tese de Doutorado, Universidade Federal do Rio Grande do Sul, 2010.

[2] J. Becla e D. L. Wang. Lessons learned from managing a petabyte. *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, 2005.

[3] J. Bezivin, S. Hammoudi, D. Lopes, e F. Jouault. Applying MDA Approach for Web Service Platform . páginas 58–70. 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004), 2004.

[4] G. Coulouris, J. Dollimore, e T. Kindberg. *Sistemas Distribuídos - Conceitos e Projeto*. Artmed, 2006.

[5] 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.

[6] M. D Del Fabro. *Gestion de metadonnees utilisant tissage et transformation de modeles*. Tese de Doutorado, Universidade de Nantes, 2007.

[7] J. M. Favre. Towards a Basic Theory to Model Driven Engineering. Workshop in Software Model Engineering (WISME 2004), 2004.

[8] Apache Software Foundation. Apache Mahout, 2012.

[9] Eclipse Foundation. Eclipse Modeling Framework Project (EMF), 2012.

[10] Laurent Goubet. EMF Compare scalability, 2012.

[11] P. Henderson. *Functional Programming - Application and Implementation*. 1980.

[12] A. G. Kleppe, J. B. Warmer, e W. Bast. *MDA Explained: The Model Driven Architecture : Practice and Promise*. Addison Wesley, 2003.

[13] 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.

[14] J. Lin e C. Dyer. Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, abril de 2010.

[15] ED. Merks. The eclipse modeling framework. Relatório técnico, 2004.

[16] Universidade de Murcia. Modelum. Morsa, 2013.

[17] OMG. MDA Guide Version 1.0.1. Relatório técnico, 2003.

[18] Apache org. Apache hbase, 2013.

[19] Eclipse ORG. CDO, 2013.

[20] OMG org. XML Metadata Interchange, 2012.

[21] Pydoop. Pydoop version 0.6.0, 2012.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, e C. Kozyrakis. Software deployment, past, present and future. *Evaluating MapReduce for Multi-core and Multiprocessor Systems*, páginas 13–24. IEEE Computer Society, 2007.

[23] M. Scheidgen. EMF-Fragment, 2013.

[24] M. Scheidgen e A. Zubow. Map/Reduce on EMF Models.

[25] M. Scheidgen e Anatolij Zubow. Automated and Transparent Model Fragmentation for Persisting Large Models.

[26] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February de 2006.

[27] 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.

[28] W. C. Silva. *Gerência de Interfaces para Sistemas de Informação: uma abordagem baseada em modelos*. Mestre, Universidade Federal de Goiás, 2010.

[29] H. C. S Sousa. *Construção Automatizada de Casos de Teste Usando Engenahria Dirigida por Modelos*. Dissertação, Universidade Federal do Maranhão, 2009.

[30] D. Steinberg, F. Budinsky, M. Paternostro, e E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.

[31] Rico Suter. Mongodb an introduction and performance analysis. Relatório técnico, 2012.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, e R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. *Data Engineering (ICDE)*, páginas 996–1005. IEEE Computer Society, 2010.

[33] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[34] Yahoo. Advanced mapreduce features, 2013.

[35] S. Yang, X. Yan, B. Zong, e A. Khan. Towards Effective Partition Management for Large Graphs.

[36] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erligsson, P. K. Gunda, e J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *OSDI'08: Eighth Symposium on Operating System Design and Implementation*. USENIX, 2008.