ALEX AMORIM DUTRA

Orientador: Álvaro Rodrigues Pereira Jr.
Co-orientador: Felipe Santiago Martins Coimbra de Melo

ADICIONANDO ESCALABILIDADE AO *FRAMEWORK DE RECOMENDAÇÃO IDEALIZE*

Ouro Preto
Novembro de 2011

Universidade Federal de Ouro Preto
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

ADICIONANDO ESCALABILIDADE AO *FRAMEWORK DE RECOMENDAÇÃO IDEALIZE*

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

ALEX AMORIM DUTRA

Ouro Preto
Novembro de 2011

UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO

ADICIONANDO ESCALABILIDADE AO *FRAMEWORK DE RECOMENDAÇÃO IDEALIZE*

ALEX AMORIM DUTRA

Monografia defendida e aprovada pela banca examinadora constituída por:

Dr. Álvaro Rodrigues Pereira Jr. – Orientador
Universidade Federal de Ouro Preto

Msc. Felipe Santiago Martins Coimbra de Melo – Co-orientador
Universidade Federal de Ouro Preto

Dr. Joubert de Castro Lima
Universidade Federal de Ouro Preto

Dr. Luiz Henrique de Campos Merschmann
Universidade Federal de Ouro Preto

Ouro Preto, Novembro de 2011

Resumo

Palavras-chave: Escalabilidade. Sistemas de Recomendação. *Idealize Recommendation Framework* (IRF). *Hadoop*. *HBase*.

Desde a antiguidade o homem utiliza meios para realizar recomendações às outras pessoas com as quais se relaciona. Como por exemplo na *Web*, sistemas de recomendação têm a cada dia deixado de ser uma novidade e se tornado uma necessidade para os usuários, devido ao grande volume de dados disponíveis. Estes dados tendem a crescer cada vez mais, o que poderá ocasionar uma perda de tempo considerável pelo usuário ao realizar buscas manualmente para encontrar conteúdos relevantes. Sistemas de recomendação têm a finalidade de levar conteúdo relevante a seus utilizadores de forma personalizada. Para isto são utilizados métodos de aprendizagem de máquina, tais como agrupamento de usuários, cálculo de similaridade entre itens, entre outros. Para atuarem de maneira eficaz, os algoritmos de recomendação precisam manipular grandes volumes de dados, o que torna necessário tanto o armazenamento quanto o processamento destes dados de maneira distribuída. Ainda, é desejado que a distribuição tanto do armazenamento quanto do processamento sejam escaláveis, ou seja, é desejado que mais capacidade de armazenamento e processamento possam ser acrescentados à medida que forem necessários. Neste trabalho descrevo a respeito de escalabilidade sobre o *Idealize Recommendation Framework* (*IRF*). Para tornar o *IRF* escalável dois *frameworks open-source* foram utilizados, o *Hadoop* e *HBase*. *Frameworks* estes presentes em grandes sistemas que utilizam computação distribuída e escalável.

Abstract

Keywords: Scalability. Recommender Systems. *Idealize Recommendation Framework* (*IRF*). Hadoop. HBase.

Since ancient times men have used several means to make recommendations to other people with whom they relate. In Web, recommendation systems have ceased to be a novelty to become a necessity for users due to the large volume of data available. These data tend to grow more, which may cause a considerable loss of time by users to manually perform searches to find relevant content. Recommender systems are designed to bring relevant content to their users in a personalized way. For so much, they used methods from machine learning, such as clustering of users, calculation of similarity among items, among other. To work effectively, the recommendation algorithms must handle large volumes of data, which requires both storage and processing of these data to be performed in a distributed manner. Still, it is desired that the distribution of both storage and processing be scalable, that means, can be added as needed. In this work describe about scalability on *Idealize Recommendation Framework* (*IRF*). To make the *IRF* scalable have used two open-source frameworks, *Hadoop* and *HBase*, because they are present in large distributed and scalable computing systems.

Dedico este trabalho:

A Deus por ter me oferecido a oportunidade de viver e poder realizar mais um sonho.

Ao meu pai Pedro, pelos ensinamentos e exemplo de vida.

A minha querida mãe Zélia, por todo incentivo e carinho oferecido durante toda a minha trajetória de vida.

Aos meus irmãos que estão sempre me incentivando com palavras, troca de experiências, pelos momentos de alegria e diversão, o que impulsiona e torna estimulante cada dia da vida.

Agradecimentos

A Deus pelo fato de me proporcionar a vida e por eu poder conviver com as pessoas que estiveram a minha volta durante este tempo.

Aos meus pais, por sempre acreditarem em mim e terem me apoiado, oferecendo além de carinho, ensinamentos de vida e condições para conclusão do curso e deste trabalho.

Aos meus irmãos pelas palavras de incentivo, por todo o companheirismo que recebo e também por estarem presentes em vários momentos da minha vida.

Aos professores, e especialmente ao meu orientador Álvaro Rodrigues pelo apoio, credibilidade e ajuda nas dúvidas que por vezes me surgiam.

Ao meu co-orientador Felipe Melo, pela paciência e idéias que não lhe faltavam na hora que eu precisava.

Aos meus familiares e em especial minha avó "Dona Maria" que é um exemplo para minha vida.

Aproveito, assim, para agradecer aos integrantes do laboratório Idealize por estarem sempre dispostos a ajudar e passar conhecimentos.

Aos meus amigos e colegas pelo companheirismo e pelos momentos agradáveis de convívio e ao "Thiagão" (in memoriam), pois sempre estava disposto para os momentos de estudos e descontração.

A Universidade Federal de Ouro Preto, onde encontrei um ambiente acolhedor e uma infra-estrutura que possibilitou a realização deste trabalho.

E por fim agradeço a todos aqueles que sei que contribuíram de forma direta e indiretamente para a realização deste trabalho.

A todos estes os meus sinceros agradecimentos.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Com o crescimento da produção de dados, principalmente na *Web* [10], temos ao alcance informações relevantes em diversas áreas. Algumas vezes quando estamos realizando buscas sobre um determinado assunto, produto ou qualquer outro item, acabamos não encontrando o que desejamos. Não encontrar o que seja realmente relevante deve-se à grande quantidade de dados existentes e a dificuldade de realização de buscas manuais sobre estes dados. Sistemas de recomendação têm a finalidade de levar ao usuário o que realmente é relevante para ele.

O *Idealize Recommendation Framework* (*IRF*) foi desenvolvido para suportar diversas estratégias de recomendação, sendo recomendações realizadas por filtragem colaborativa, baseada em conteúdo, dados de uso e híbrida [13, 15, 4, 1]. As aplicações de recomendação desenvolvidas sobre o *IRF* até o momento possuem as seguintes abordagens: baseada em conteúdo [4, 6], filtragem colaborativa [13, 15], dados de uso [5] e híbrida [1]. Em resumo, a recomendação baseada em conteúdo é realizada com base na descrição dos itens mais similares ao item sendo acessado, ou baseada em itens que possuem características similares as definidas no perfil do usuário [1]. Recomendações por filtragem colaborativa têm sua origem na mineração de dados [3] e constituem o processo de filtragem ou avaliação dos itens através de múltiplos usuários [1, 15, 19], muitas vezes formando grupos de usuários que possuem características similares. Recomendações baseadas em dados de uso levam em consideração as ações realizadas por seus usuários, por exemplo, a sequência de links clicados por um usuário quando navega em um site de compras. A recomendação híbrida possibilita que as limitações de cada técnica sejam supridas por características das demais [1].

De acordo com Chris Anderson, editor chefe da revista *Wired*, *"We are leaving the age of information and entering the age of recommendation"* ("Estamos deixando a era

da informação e entrando na era da recomendação") [2]. Isto significa que não é suficiente recuperar informações. Estas informações devem ser recuperadas de forma personalizada. Esta personalização exige processamentos computacionalmente caros que dependem de grandes quantidades de dados para apresentarem uma boa acurácia. Para que este processamento possa acontecer, deve-se destacar a importância da distribuição e escalabilidade nestes sistemas, uma vez que estes fatores estão relacionados ao volume de dados a ser processado.

Escalabilidade está relacionada à distribuição dos componentes e serviços do sistema de forma a aumentar o desempenho e ser capaz de processar grandes volumes de dados. No caso de sistemas de recomendação, pretende-se diminuir o tempo de processamento das recomendações à medida em que aumenta-se o número de máquinas, e além disto ser capaz de processar as recomendações a partir de grandes volumes de dados.

Grandes empresas como *Facebook*[1], *Yahoo*[2], *Google*[3], *Twitter*[4] e *Amazon*[5] armazenam volumes de dados da ordem de petabytes[6], de onde podem ser extraídas informações relevantes e personalizadas. Sabe-se que este volume de dados está em constante crescimento, o que obriga as empresas a adotarem estratégias de distribuição tanto do processamento quanto do armazenamento destes dados. Sistemas de recomendação criados para processar estes dados devem, na mesma linha, possibilitar o armazenamento e processamento distribuído, enquanto ao mesmo tempo devem possibilitar a adoção de diferentes métodos de recomendação, o que justifica o desenvolvimento de um *framework* distribuído e escalável para sistemas de recomendação.

Analisando as características citadas no parágrafo anterior, os objetivos propostos para este trabalho foram:

- A criação de uma aplicação de recomendação baseada em conteúdo [4] utilizando a arquitetura do *IRF* pseudo-distribuída[7].
- A criação de classes comuns que facilitem a implementação de aplicações de recomendação distribuídas sobre o *IRF*.

[1] www.facebook.com
[2] www.yahoo.com
[3] www.google.com
[4] www.twitter.com
[5] www.amazon.com
[6] http://escalabilidade.com/2010/05/18/
[7] A arquitetura do IRF pseudo-distribuída é assim denominada por possuir somente três máquinas que operam para o processamento das recomendações.

- A implementação de uma aplicação de recomendação distribuída baseada em filtragem coltaborativa [15]. Desta forma, ao se criar novas aplicações distribuídas, poderá ser seguido o exemplo da aplicação implementada.

Neste trabalho, o Capítulo 2 denominado "Conceitos Básicos", possui conceitos gerais e básicos que são necessários para o entendimento dos capítulos seguintes. No Capítulo 3, "*Idealize Recommendation Framework*", descrevo a implementação dos componentes do *IRF*, baseando em geral nas informações contidas no artigo [13]. O Capítulo 4, "*IRF* escalável e distribuído", possui a descrição dos componentes e classes implementadas para a parte distribuída e escalável do *IRF*. Ainda no Capítulo 4, na seção 4.3.1 denominada "Descrição dos componentes distribuídos implementados", apresento detalhes sobre a implementação de uma aplicação de recomendação totalmente distribuída baseada em filtragem colaborativa. No Capítulo 5, "Experimentos", estão descritos os resultados dos experimentos realizados e comparações realizadas com estes resultados. E por fim no Capítulo 6, são apresentadas as conclusões baseando no desenvolvimento deste trabalho e nos resultados obtidos.

Capítulo 2

Conceitos Básicos

2.1 Introduzindo *Hadoop* e *MapReduce*

Para adicionar um módulo com capacidade de processamento de recomendações de forma distribuída ao *Idealize Recommendation Framework*, foram utilizados os *frameworks Apache Hadoop*[1] e *Apache HBase*[2].

O *Hadoop* possui um paradigma de programação chamado *MapReduce*. O *MapReduce* oferece um mecanismo de estruturação de cálculos de forma a tonar possível a execução por muitas máquinas. O modelo de programação dentro do paradigma *MapReduce* é o seguinte:

1 - A entrada é feita na forma de muitas chaves e valores (*K1, V1*), em geral a partir de arquivos de entrada contidos em um *Hadoop Distributed File System (HDFS)*. O *HDFS* é o sistema de armazenamento primário usado por aplicativos *Hadoop*. O *HDFS* cria várias réplicas de blocos de dados e distribui estes blocos em nós de computação de um *cluster* para que seja confiável devido à replicação, e possa realizar cálculos de forma rápida a partir da localidade espacial[3].

2 - Um *Map* é uma função aplicada a cada par (*K1, V1*), que resulta em pares de chave-valor (*K2, V2*). Os valores de *V2* são combinados para todas as chaves de *K1* que possuem o mesmo valor[4]. Desta forma a saída do *Map* e entrada para o *Reduce* possui a forma *K2, Iterator<Valores>*.

3 - A função *Reduce* opera sobre a entrada recebida a partir do *Map* e faz os devidos cálculos sobre estes dados gerando novos valores chave-valor (*K3, V3*). A partir destes

[1] http://hadoop.apache.org/
[2] http://hbase.apache.org/
[3] http://hadoop.apache.org/hdfs/
[4] http://hadoop.apache.org/common/docs/current/mapred_tutorial.html

cálculos os novos valores (*K3, V3*) são armazenados novamente no *HDFS*. Os valores oriundos tanto do *Map* quanto do *Reduce* são do tipo *Writable* (tipo definido pelo próprio *Hadoop*), pois estes objetos devem ser passíveis de serem armazenados em disco.

A Figura 2.1 apresenta a sequência de funcionamento do *MapReduce*.

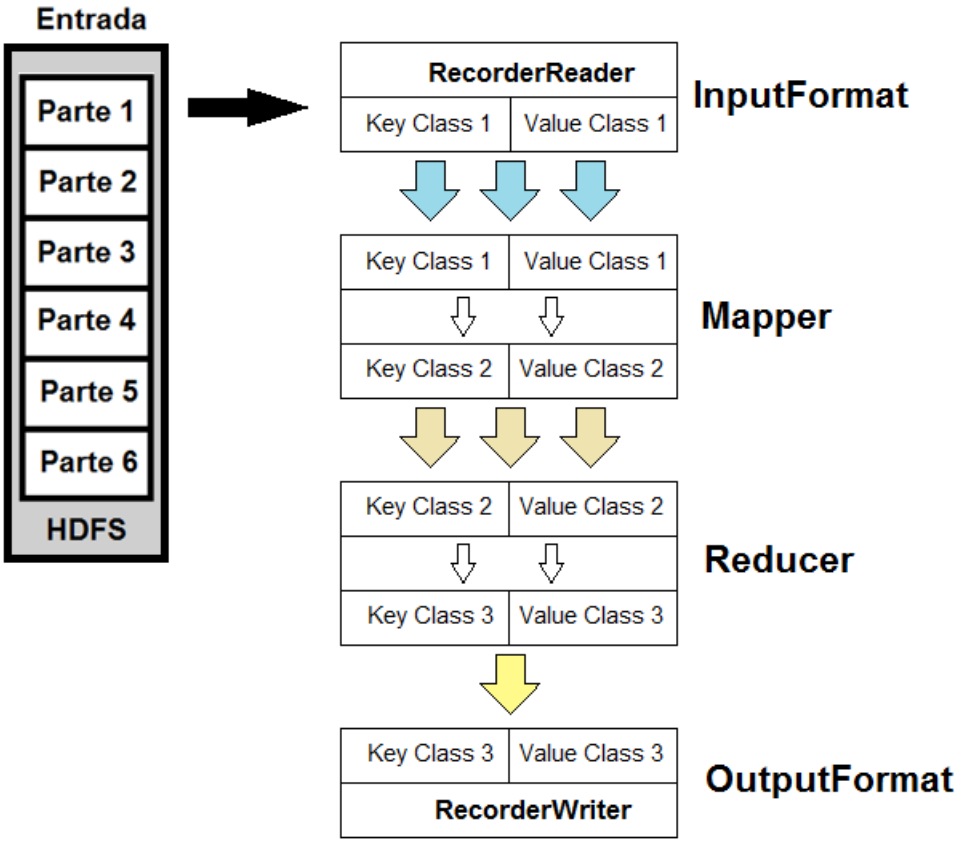

Figura 2.1: Funcionamento do *MapReduce*

2.2 Estrutura de execução do *HBase*

HBase é um banco de dados utilizado pelo *Hadoop* e é uma maneira rápida e confiável para manipular grandes volumes de dados indexados[5]. É utilizado no *IRF* devido à necessidade de acessos aleatórios de leitura e escrita em tempo real em grandes bases de dados. O objetivo do *HBase* é suportar tabelas muito grandes com milhões de linhas

[5] http://hbase.apache.org/

sobre *hardwares* de baixo custo[6]. O *Hbase* trabalha de forma distribuída e possui um modelo de armazenamento orientado a colunas modelado de acordo com o *Bigtable*[8].

Uma tabela do *Hbase* possui a seguinte estrutura: *<key>:<column family>:<qualifier><value>*. *Key* é um valor único dos registros de uma tabela, *column family* é o nome dado a uma coluna, *qualifier* é um valor único para um determinado *column family* e *value* é o valor do respectivo *qualifier*. O *Hadoop* provê operações sobre *HBase*. O *MapReduce* quando utiliza o *HBase* é implementado por duas novas classes *TableMap* e *TableReduce*, que possuem o seguinte modelo de operação:

1 - A entrada é constituída na forma de muitas chaves e valores (*K1, V1*). Cada uma destas chaves e valores é análoga a um registro de um banco de dados relacional. Os valores de entrada são os valores que estão em tabelas, as tabelas do *Hbase* são denominadas *HTables*.

2 - Um *TableMap* é uma função aplicada a cada par (*K1, V1*), que resulta em pares de chave-valor (*K2, V2*). Os valores de *V2* são combinados para todas as chaves de *K2* que possuem o mesmo valor. Desta forma a saída do *TableMap* e entrada para o *TableReduce* possui a seguinte forma: *K2, Iterator<Valores>*. Os valores de *K1* são as chaves com valores únicos no *HBase*. Em analogia a um banco de dados relacional, podemos considerar que os valores de *K1* são chaves primárias de uma tabela e os valores de *V1* são os valores das colunas.

3 - A classe *TableReduce* opera sobre a entrada e faz os devidos cálculos sobre estes dados, entrada esta recebida a partir da saída do *TableMap*. A partir dos cálculos realizados pelo *TableReduce*, os valores são armazenados em uma *HTable*. Os valores (*V2*) que são gerados pelo *TableMap* são valores do tipo *Writable*. No *TableReduce* temos a restrição de que os objetos de saída sejam do tipo *Put*, que é o tipo de um objeto de gravação em uma *HTable*.

[6] http://hbase.apache.org/

Capítulo 3

Idealize Recommendation Framework (IRF)

As informações presentes neste capítulo referentes ao *Idealize Recommendation Framework* estão baseadas no artigo [13]. O *IRF*[1] foi implementado utilizando a linguagem de programação Java. Ao realizar pesquisas relacionadas à área de recomendação foram encontrados apenas dois *frameworks*, *Apache Taste*[2] e *MyMedia*[3]. O *IRF* foi construído a partir do *Framework Apache Mahout*[4] que encapsula o *Apache Taste*.

O *Apache Taste* oferece componentes para criação de modelos de dados e alguns algoritmos de recomendação que utilizam a abordagem de filtragem colaborativa.

Além do *framework Mahout*, foram utilizados dois outros *frameworks* para implementar a arquitetura distribuída sobre o *IRF*, *Apache Hadoop*[5] e *Apache HBase*[6].

Um *framework* provê uma solução para uma família de problemas semelhantes [12], usando um conjunto de classes abstratas e interfaces que mostram como decompor a família destes problemas, e como os objetos dessas classes colaboram para cumprir suas responsabilidades. O conjunto de classes de um *framework* deve ser flexível e extensível para permitir a construção de aplicações diferentes mais rapidamente dentro do mesmo domínio, sendo necessário implementar apenas as particularidades de cada aplicação.

Em um *framework*, as classes extensíveis são chamadas de *hot spots* [12, 13]. O importante é que exista um modelo a ser seguido para a criação de novas aplicações de

[1] http://sourceforge.net/projects/irf/
[2] http://taste.sourceforge.net
[3] http://mymediaproject.codeplex.com
[4] http://lucene.apache.org/mahout
[5] http://hadoop.apache.org/
[6] http://hbase.apache.org/

recomendação e definir a interface de comunicação entre os *hot spots* desse modelo. As classes que definem a comunicação entre os *hot spots* não são extensíveis e são chamadas de *frozen spots* [13], pois constituem as decisões de *design* já tomadas dentro do domínio ao qual o *framework* se aplica.

A Figura 3.1 apresenta a estrutura em alto nível dos *frameworks* que compõem o *IRF*.

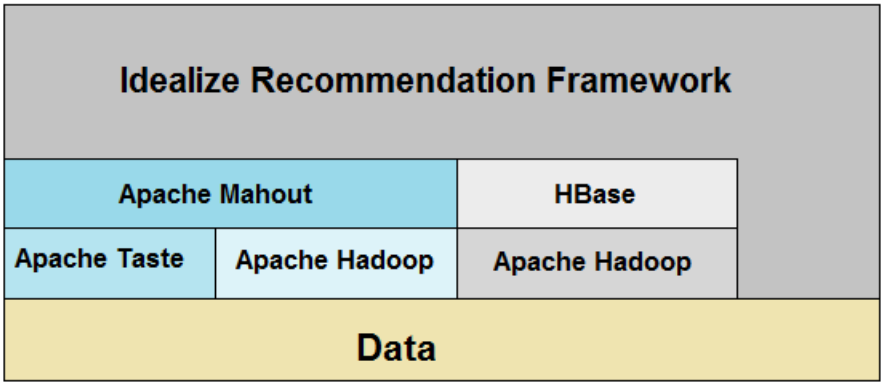

Figura 3.1: Arquitetura alto nível dos *frameworks* utilizados

As características que diferem o *IRF* dos outros dois *frameworks* de recomendação aqui citados são:

- *Componentes que proporcionam a criação de máquinas de recomendação que utilizem diversas abordagens de recomendação.* Isto se deve ao fato de o *IRF* possuir componentes que abordam todo o ambiente de produção, como serializadores de entrada, serializadores de saída, modelo de dados, entre outros componentes que são implementados de acordo com a necessidade da abordagem de recomendação em questão. O *Apache Taste* é focado em recomendações utilizando somente filtragem colaborativa.

- *Criação de setores dedicados à realização de tarefas específicas no ambiente de recomendação, trabalhando de forma independente.* Os outros dois *frameworks* de recomendação não possuem esta distribuição de setores. Os setores criados para o *IRF* são: setor de *Batch*, setor de *Cache* e setor de *Input*, descritos na Seção 3.1, denominada "Descrição dos setores do modelo de produção".

- *Processamento e armazenamento distribuído*, quando comparado ao *MyMedia*, pelo fato do *MyMedia* não possuir esta característica. Utilização do *framework HBase* para armazenamento de dados de forma distribuída, quando comparado ao *Apache Mahout*, pelo fato do *Mahout* utilizar diretamente o HDFS[7] do *Apache Hadoop*.

3.1 Descrição dos setores do modelo de produção

As aplicações do *IRF* são divididas em três setores que podem ser vistos como três máquinas distintas, onde cada uma é responsável por uma funcionalidade da aplicação de recomendação. A finalidade dos três setores é dissociar os componentes que possuem operações custosas computacionalmente e permitir que as recomendações sejam fornecidas de forma instantânea.

Os setores do *IRF* utilizados no modelo de produção são setor de *Batch* (utilizado para processamento em lote), setor de *Cache* (fornece a recomendação para o mundo exterior e armazena recomendações pré-calculadas em memória principal) e setor de *Input* (responsável por realizar a manipulação da base de dados) [13]. A Figura 3.3 ilustra a comunicação entre cada um dos setores e a comunicação do setor de *Batch* com o *cluster*[8]. Além destes setores podemos ver também na Figura 3.3 a comunicação com a base de dados. A base de dados representada na Figura 3.3 pode ser um banco de dados, arquivos de texto, um índice do *Apache Lucene*[9] ou qualquer outra forma de armazenamento de dados.

Ressaltando a diferença entre uma arquitetura pseudo-distribuída e uma arquitetura distribuída no *IRF*, na arquitetura pseudo-distribuída são utilizadas somente três máquinas e possivelmente uma quarta máquina para armazenamento de dados. Cada uma destas três máquinas é uma instância do seu respectivo setor, portanto, as recomendações são processadas na máquina de *Batch* e não processadas no *cluster* como acontece com o processamento de recomendações na arquitetura totalmente distribuída.

Pode-se observar na Figura 3.2 como segue o fluxo de uma requisição de recomendação.

[7] Hadoop Distributed File System

[8] Uma quantidade de computadores que comunicam entre si a fim de solucionar determinado problema.

[9] http://lucene.apache.org/

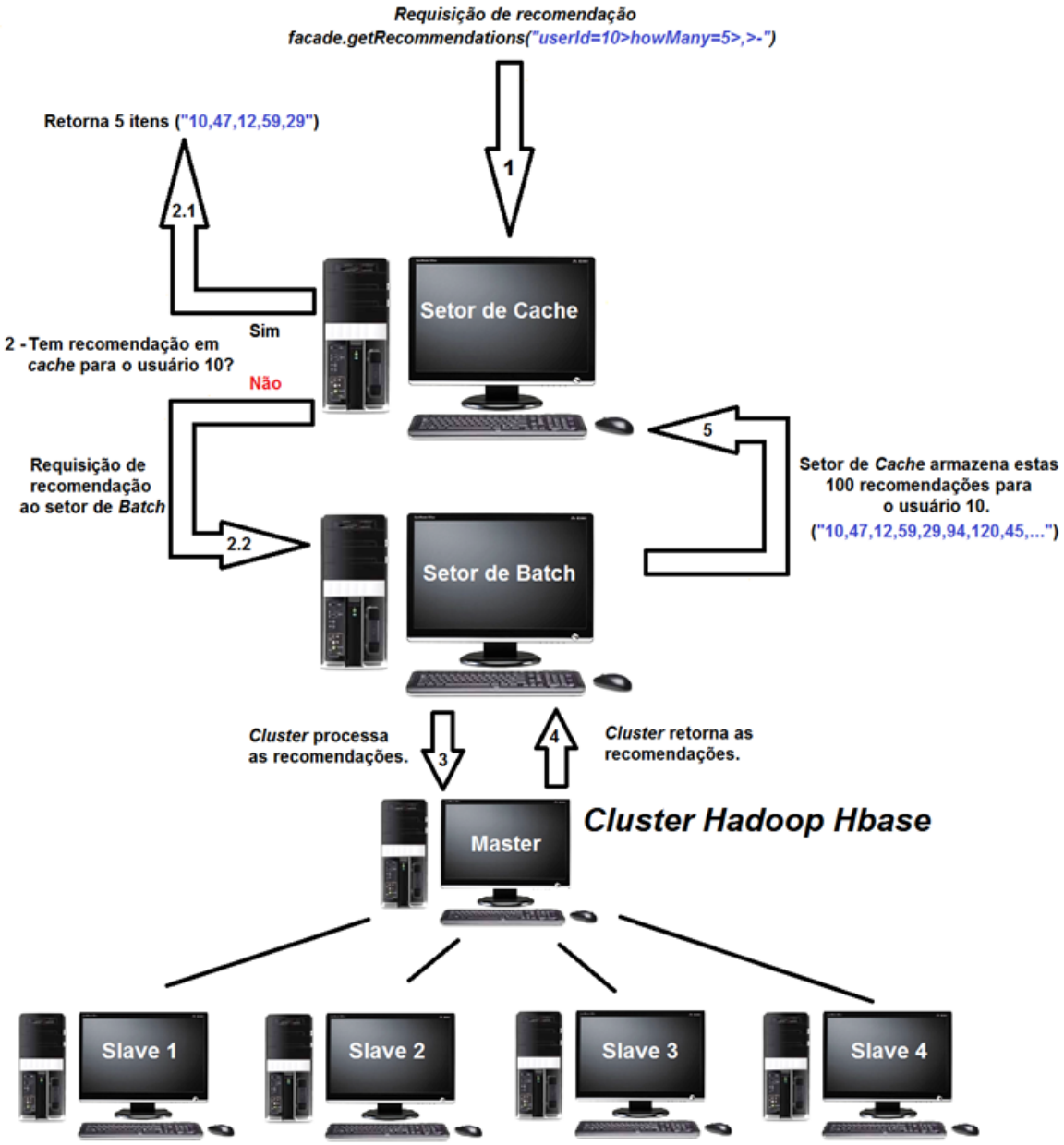

Figura 3.2: Fluxo de requisição de recomendação

3.1.1 Setor de *Cache*

O setor de *Cache* implementa a interface do sistema com o usuário. Este setor permite aos usuários a requisição de recomendações e o envio de informações de *feedback* ao sistema. É destinado a armazenar recomendações pré-calculadas de tal forma a fornecer respostas instantâneas aos pedidos de recomendações que chegam à sua fachada [13]. Neste setor são implementadas as heurísticas de gestão de *cache* de dados de forma a decidir quais informações devem ser mantidas em memória principal, e quais devem ser armazenadas em memória secundária. O desenvolvedor de aplicações sobre o *IRF* que deverá definir a política de *Cache* para sua aplicação.

3.1.2 Setor de *Batch*

Quando não se utiliza a arquitetura distribuída, este setor é responsável pelo processamento das recomendações e dos *feedbacks* recebidos dos usuários. As requisições de recomendações e operações de *feedbacks* são recebidas pelo setor de *Cache* e repassadas para o setor de *Batch*. O setor de *Batch* registra alguns de seus componentes na rede de maneira a torná-los remotamente acessíveis utilizando *RMI*[10]. Este setor também realiza comunicação com o setor de *Input*. O setor de *Input* notifica o setor de *Batch* quando há modificações nas bases de dados, como por exemplo, quando um novo usuário é inserido na base de dados. Quando houver a necessidade de utilização da arquitetura distribuída, o setor de *Batch* torna-se um intermediador entre os demais setores e o *cluster*. Desta maneira quem passa a ser responsável pelo processamento das recomendações são as máquinas disponíveis no *cluster*.

3.1.3 Setor de *Input*

O setor de *Input* permite a comunicação entre o mundo externo e a base de dados utilizada. Através da fachada deste setor, o usuário pode realizar operações de inserção, remoção e atualização dos itens e dados de usuários. O setor de *Input* foi criado a fim de dissociar a produção de recomendações das tarefas de gerenciamento das bases de dados. Como as duas tarefas são custosas computacionalmente, o ideal é manter estes setores trabalhando de maneira separada, utilizando mais recursos de *hardwares* disponíveis.

[10]Remote Method Invocation, recurso oferecido pela linguagem Java afim de permitir o acesso a objetos remotos

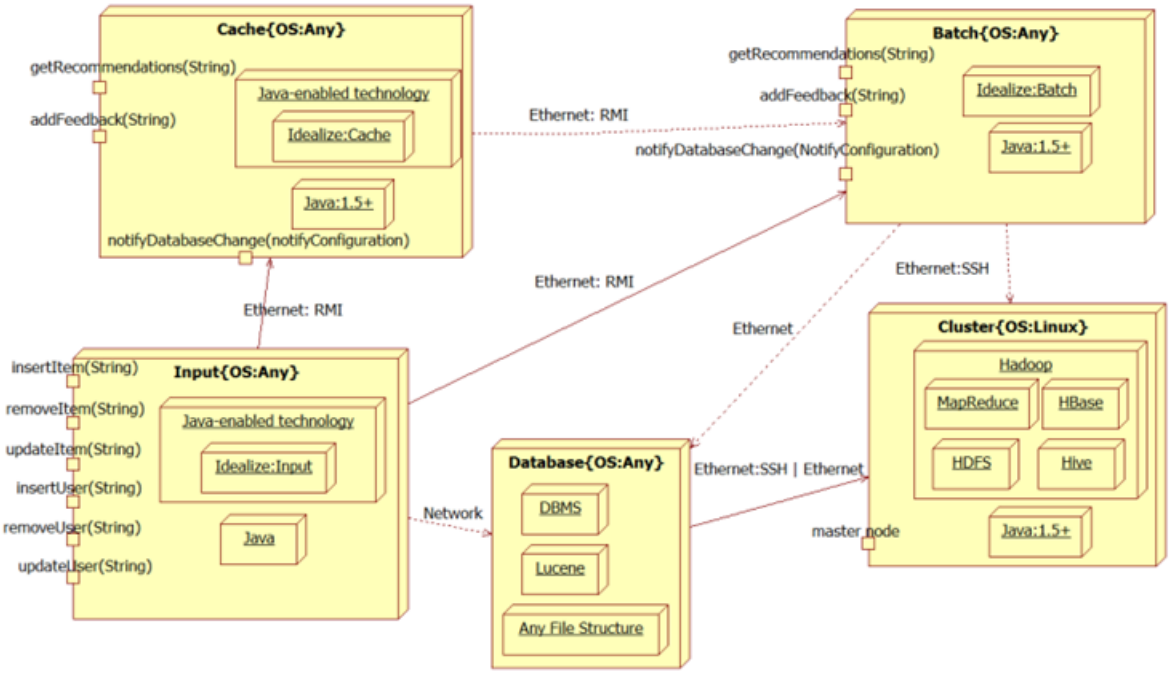

Figura 3.3: Arquitetura utilizada no modelo de produção

3.2 Componentes do *Idealize Recommendation Framework*

O *Idealize Recommendation Framework* possui os seguintes pacotes: *base*, *hot_ spots* e *distributed_ hotspots*. No pacote *base* estão os *frozen spots* [13], pelo fato de possuírem implementações únicas e comuns que definem o fluxo do processamento de recomendações. Neste pacote também estão classes de auxílio a operações comuns, como por exemplo operações sobre *strings*.

Os componentes do pacote *hot_ spots* devem ser implementados de acordo com cada aplicação a ser desenvolvida, em geral são classes abstratas ou interfaces.

No pacote *distributed_ hotspots* estão os componentes que são utilizados somente em aplicações distribuídas e serão detalhados no Cápitulo 4, "*Idealize Recommendation Framework* Distribuído e Escalável".

Os componentes existentes no *IRF* separados por pacotes são:

- Pacote *base*

  Instantiator, IdealizeClassLoader, PropertiesLoader, RemoteFacade, RemoteIdealizeRecommenderFacade, RemoteIdealizeInputFacade, IdealizeRecommenderFacade, IdealizeInputFacade, Cache, CacheObserver, DataModelStrategyContext,

IdealizeCoreException, IdealizeUnavailableResoucerException, IdealizeConfigurationException, IdealizeInputException, IdealizeLogg.

- Pacote *hot spots*

  InstantiatorWorker, AbstractInstantiator, IdealizeDataModel, DataModelLoaderStrategy, InputBean, BaseBean, BaseInputBean, InputInterpreter, Controller, InputController, RemoteBatchProcessor, BatchProcessor DataManipulator, RecommendationSerializer, IdealizeRecommender, Restorable, BaseStorable.

Na Figura 3.4 os métodos de acesso externo a fachada estão representados em forma de círculos. Os símbolos em formato de pasta representam os pacotes que armazenam as superclasses, ou seja, o os pacotes que contém as classes abstratas ou interfaces que devem ser implementadas pelo desenvolvedor de aplicações sobre o *IRF*. As superclasses estão representadas na Figura 3.4 pelos retângulos com os nomes em negrito.

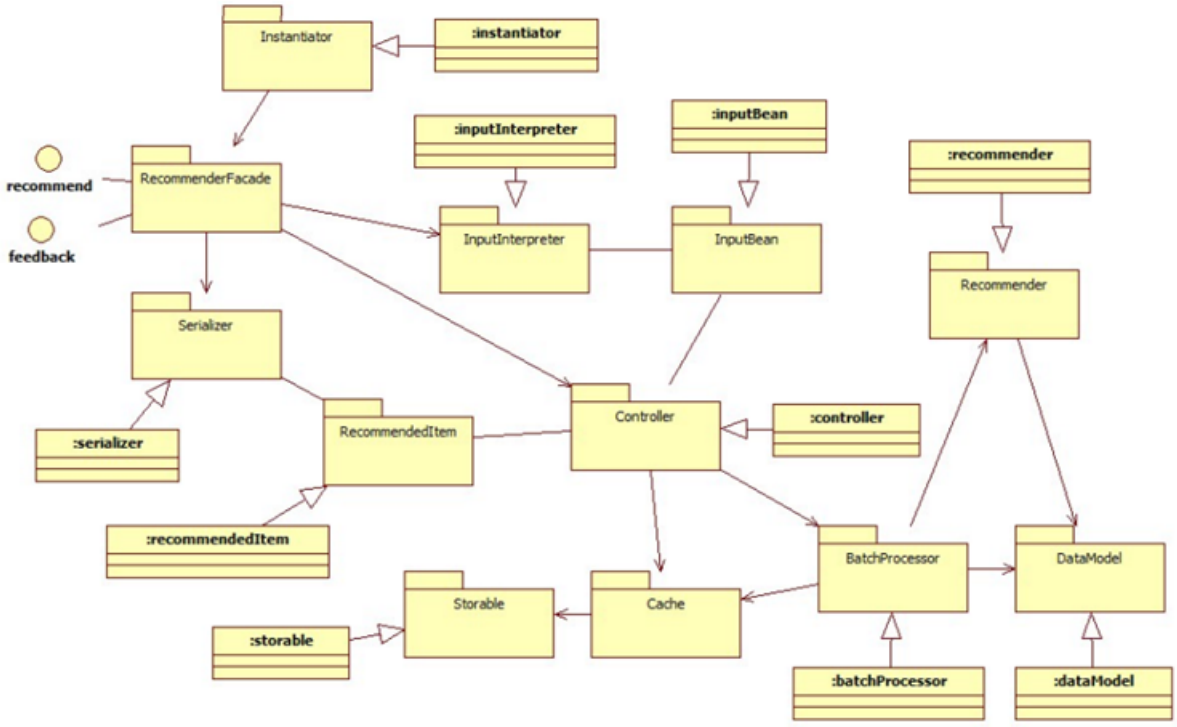

Figura 3.4: Arquitetura alto nível dos componentes utilizados no setor de *Batch e Cache*

A Figura 3.5 apresenta a arquitetura alto nível dos componentes utilizados no setor de *Input*.

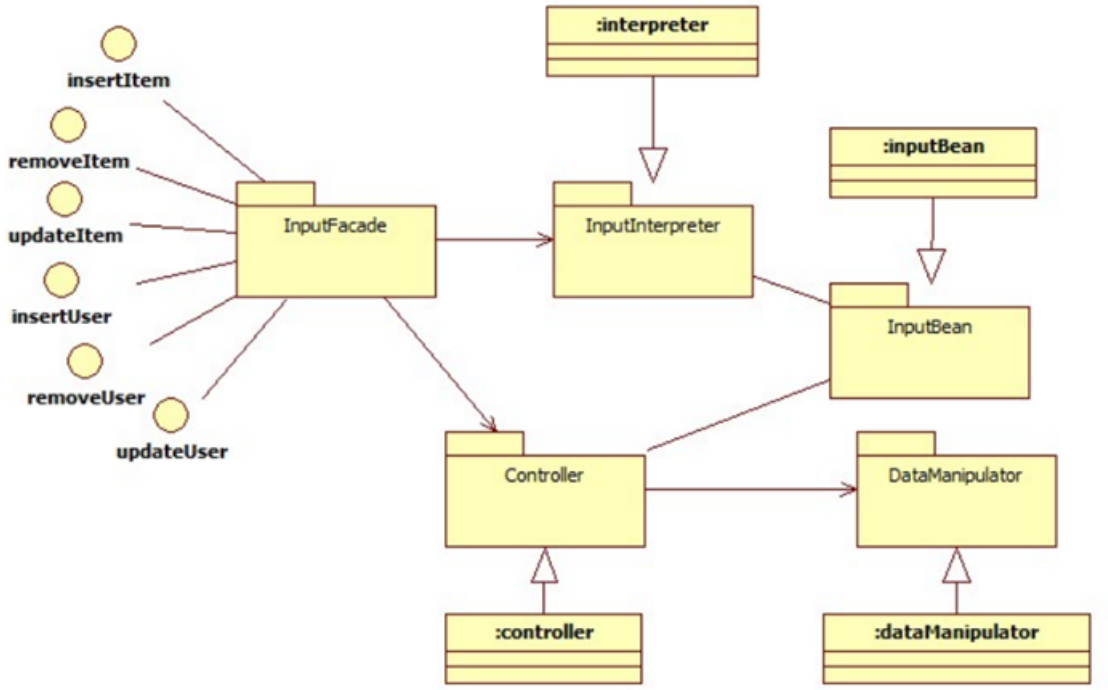

Figura 3.5: Arquitetura alto nível dos componentes utilizados no setor de *Input*

3.2.1 Descrição dos componentes do pacote *base*

Apesar da complexidade e detalhamento envolvidos na explicação do funcionamento de cada um dos componentes do *IRF*, para uma compreessão completa destes componentes, é necessário descrever todos eles pelo menos uma vez, mesmo que seja de forma sucinta.

- *classes.properties:* Arquivo de configurações utilizado para definir as propriedades das aplicações. Diferentes aplicações de recomendação devem ser executadas para cada uma das abordagens de recomendações (*CF, CB, UD e HB*). O arquivo de propriedades permite que as aplicações sejam configuradas sem a necessidade de modificações no código-fonte. Este arquivo define quais são as classes concretas a serem utilizadas como implementação de cada um dos componentes do *framework*. Também possui informações sobre qual será o método de recomendação a ser utili-

zado, separador da *string* enviada à fachada, o modelo de dados que será utilizado pelo recomendador, entre outras propriedades que definem quais classes serão instanciadas em tempo de execução. O desenvolvedor deverá apenas configurar o arquivo de propriedades de acordo com suas necessidades, e iniciar a execução do setor correspondente. Como exemplo, em sendo necessário criar um modelo de dados com 1000 usuários, este valor é configurado neste arquivo. Para carregar dados de 10000 usuários basta modificar a mesma propriedade no arquivo para o valor desejado.

- *Constants:* As informações configuradas no arquivo de propriedades, são em geral carregadas para a classe *Constants*. Desta forma cada aplicação deverá possuir uma implementação desta classe com os atributos que serão utilizados na aplicação. Esta classe serve para armazenar de maneira estática valores que são utilizados por todo o sistema, tais como definição do endereço IP de objetos a serem acessados remotamente, número de usuários a carregar para o modelo de dados, nome dos métodos de recomendações a serem utilizados, entre outros atributos referentes a cada aplicação.

- *PropertiesLoader:* Carregador do arquivo de propriedades *classes.properties*.

- *Instantiator:* Este componente é responsável pela leitura de cada propriedade do arquivo de configurações e instanciação das classes que serão utilizadas na aplicação. A sequência de instanciação de classes e quais propriedades serão lidas é definida pelo desenvolvedor em uma subclasse deste componente, e estas informações são dependentes do tipo de aplicação.

- *RemoteFacade:* É através de cada fachada que os clientes irão se comunicar com as aplicações, ou seja, a fachada é uma interface com o mundo exterior. Os pedidos de recomendação, informações de *feedback* ou operações de atualização, inserção ou remoção de itens chegam sempre por fachadas que são subclasses deste componente. Todos os dados que chegam e saem das fachadas estão em forma de *strings*. A interface *RemoteFacade* estende a interface *Remote*, uma interface de marcação da linguagem Java. A interface *Remote* permite que os objetos sejam registrados via *RMI* e que os outros setores acessem esta fachada remotamente. A ideia por trás de uma interface de marcação é que a semântica das relações seja mantida, porém a relação pode acontecer de diversas maneiras.

- *IdealizeRecommenderFacade:* Esta classe implementa a interface *RemoteIdealizeRecommenderFacade* e define o fluxo de produção de recomendações e fluxo re-

lacionado as atualizações dos *ratings* (*feedback* enviado ao sistema por parte dos usuários).

- *IdealizeInputFacade:* Esta classe também permite a comunicação entre o mundo externo e a aplicação, comunicação esta referente à manipulação da base de dados. Esta classe possui a implementação de métodos relacionados com as operações de inserção, atualização ou remoção de itens e usuários nas respectivas bases de dados.

- *Cache:* O componente *Cache* é utilizado em todos os três setores. Existe o componente *Cache* e também existe um setor totalmente dedicado a políticas de *caching*, denominado setor de *Cache*. O componente *Cache* serializa em disco componentes do tipo *Storable*. Geralmente, pela grande quantidade de dados que estão sendo manipulados, faz-se necessário armazenar objetos com valores já processados em memória secundária. O componente *Cache* possui o método *setData(BaseStorable storable, boolean serialize)*, que permite indicar se o armazenamento em disco de um objeto deve ou não ocorrer.

- *CacheObserver:* Este componente guarda a informação de qual foi o último objeto do tipo *storable* armazenado em disco. O componente *Cache* faz a notificação ao *CacheObserver* indicando qual foi o último objeto armazenado em disco. Esta estratégia permite que o sistema se recupere automaticamente após ser reiniciado, a partir do ponto de restauração criado ao gravar os objetos em disco.

- *IdealizeExceptions:* As exceções do *IRF* apresentam basicamente a mesma implementação de sua superclasse, a classe *Exception* provida pela linguagem Java. Estas exceções foram criadas para melhorar a semântica de manipulação de exceções, mantendo o encapsulamento, de acordo com os padrões de manipulação de exceções existentes [9].

3.2.2 Descrição dos componetes do pacote *hot_ spots*

Em um *framework*, as classes extensíveis são chamadas de *hot spots* [12, 13]. Estas classes devem ser implementadas de acordo com as necessidades presentes em cada aplicação desenvolvida.

- *AbstractInstantiatorWorker:* Classe abstrata para todas as estratégias de instanciação das classes concretas que serão utilizadas por cada um dos setores. Quando o

desenvolvedor implementar uma estratégia de instanciação dos componentes e classes a serem utilizadas, é possível tanto implementar a interface *InstantiatorWorker*, quanto criar subclasses desta.

- *InputInterpreter:* Como mencionado anteriormente, todos os dados que chegam e saem da fachada são *strings*. As classes implementadas a partir deste *hot spot* são responsáveis pela interpretação dos diferentes formatos de *strings* que chegam à fachada, podendo ser estas *strings* requisições de recomendações, *feedbacks* ou operações a serem realizadas sobre as bases de dados.

- *BaseBean:* Um objeto do tipo *BaseBean* é retornado pelo componente *InputInterpreter* após a interpretação da *string* de entrada. Este objeto encapsula os dados da *string* de entrada, recebida pela fachada.

- *IdealizeDataModel:* Esta é uma interface de marcação para objetos que representam modelos de dados. Um exemplo de interface de marcação é a interface *Serializable* da linguagem Java, que indica que os objetos são passíveis de serem armazenados em disco ou transmitidos pela rede. Embora os objetos possam ter diversas interfaces ou maneiras de serem serializados, a interface *Serializable* sinaliza para a máquina virtual que esse objeto pode ser serializado e isto basta, sendo que cabe ao objeto que irá armazená-lo em disco, definir as regras de armazenamento. Da mesma forma, *IdealizeDataModel* marca um objeto como sendo um modelo de dados, e logo, passível de ser utilizado por algum recomendador. Os métodos de recomendação precisam manipular modelos de dados. Para encontrar os itens a serem recomendados, porém cada tipo de recomendação requer diferentes modelos de dados. Assim, fica a cargo do desenvolvedor definir o modelo de dados, marcá-lo como *IdealizeDataModel* e realizar o *casting* para o modelo adequado dentro do recomendador. Por exemplo, o *DataModel* hoje implementado para recomendações baseadas em conteúdo utiliza o *Apache Lucene*, o que difere dos modelos de dados de outras abordagens de recomendações.

- *IdealizeRecommender:* Este componente constitui a base para os algoritmos de recomendação. Segue o mesmo raciocínio apresentado para *IdealizeDataModel*, ou seja, *IdealizeRecommender* constitui apenas uma interface de marcação a ser atribuída a todos os métodos de recomendação. Um novo método de recomendação deve implementar diretamente a interface *IdealizeRecommender* ou tornar-se subclasse de algumas das subclasses já presentes no *IRF* como *IdealizeAbstractDistributedRecommender, IdealizeCFAbstractRecommender, IdealizeCBAbstractRe-*

commender, IdealizeUDAbstractRecommender e *IdealizeHBAbstractRecommender*. Para cada uma destas cinco classes foram definidos os formatos dos métodos de recomendação para a sua respectiva abordagem de recomendação.

- *Controller:* Controla o fluxo da aplicação de recomendação. Este componente recebe um *Input-Bean* vindo do *InputIntepreter* e devolve uma lista de *RecommendedItem*[11], pois para ambientes de manipulação de dados o fluxo é diferente das recomendações e dos *feedbacks*. O controlador é implementado pelo criador dos novos métodos, pois a sequência de operações a serem realizadas é definida pelo desenvolvedor e cada aplicação de recomendação possui o seu controle de fluxo específico.

- *BatchProcessor:* Responsável por realizar o processamento em lote das recomendações e dos *feedbacks*. Este componente também possui a tarefa de realizar a comunicação com a parte distribuída, quando esta arquitetura for utilizada. O desenvolvedor deve implementar sua lógica de processamento em lote implementando uma subclasse de *BatchProcessor*.

- *BaseStorable:* Interface a ser implementada por objetos que podem tanto ser armazenados temporariamente em *Cache* para melhoria de desempenho, quanto serem armazenados em disco para recuperação do objeto caso o sistema venha a tornar-se indisponível em algum momento. Um objeto do tipo *BaseStorable* deve ser capaz de ter seus dados atualizados, ser armazenado no disco e ser recuperado a partir do disco, e para isto, esta interface define os métodos *update, serialize* e *restore*.

- *RecommendationSerializer:* Como já mencionado, todas as informações retornadas pela fachada estão em formato de *string*. Este componente é responsável por concatenar valores e colocar em formato de *string* a lista de itens recomendados (*RecommendedItem*) retornados pelo controlador. Desta forma a *string* de retorno possui o padrão esperado por um outro aplicativo que fará o uso destas informações. Assim o desenvolvedor implementa a interface *RecommendationSerializer* de acordo com sua necessidade. Em muitas aplicações a *string* de retorno tem o seguinte formato "itemId1,itemId2,itemId3...", sendo cada valor separado por vírgula o identificador de um determinado item sendo recomendado, porém este formato pode ser modificado de acordo com as necessidades do cliente.

[11] É uma interface reutilizada do *Mahout*, para representar os itens que são recomendados.

- *DataManipulator:* Este componente está presente apenas no setor de *Input*, e tem o papel de realizar a manipulação da base de dados. Por exemplo, no caso de um banco de dados relacional, a subclasse deste componente deve conhecer as estruturas das tabelas e saber operar sobre elas. Por outro lado, se o conjunto de dados estiver contido no *Apache Lucene*, o componente deve saber como manipular os documentos do *Lucene*, e assim de acordo com a estrutura de armazenamento de dados utilizada pela aplicação desenvolvida.

3.3 Implementação da aplicação baseada em conteúdo

A implementação desta aplicação teve como objetivo principal o entendimento dos componentes presentes no *IRF*. A recomendação baseada em conteúdo (CB) tem sua origem na área de Recuperação da Informação [3]. Em geral, recomendações que utilizam esta abordagem podem ser calculadas a partir da semelhança entre itens e podem também gerar recomendações de acordo com as informações definidas no perfil de um usuário [4, 6, 1].

Existem duas principais abordagens para a realização de recomendação baseada em conteúdo. Na primeira, uma aplicação apresentará ao usuário uma lista de itens similares a um item específico sendo visualizado ou escolhido pelo usuário, em um determinado momento. Na segunda abordagem, são utilizadas diversas informações do usuário, informações estas que definem um perfil de interesses, sendo recomendados então itens que correspondem a este perfil.

A aplicação baseada em conteúdo foi realizada utilizando a arquitetura pseudo-distribuída. Levando em consideração que a implementação de uma aplicação pseudo-distribuída foi um dos objetivos propostos neste trabalho, a descrição da aplicação baseada em conteúdo tem por finalidade detalhar como são implementados os componentes do *IRF* para este tipo de aplicação. Os componentes de uma aplicação de recomendação que utilize a arquitetura pseudo-distribuída, assim como esta aplicação baseada em conteúdo, possuem a mesma finalidade dos componentes de uma aplicação que utilize a arquitetura totalmente distribuída. O que diferencia a arquitetura distribuída da pseudo-distribuída em termos de implementação de componentes, é o fato que em uma aplicação totalmente distribuída mais componentes devem ser implementados.

Na aplicação baseada em conteúdo desenvolvida sobre o *IRF*, foi utilizado o *Apache Lucene*[12]. O *Lucene* é uma ferramenta para indexação e recuperação de informações textuais. Na fase de indexação, os dados originais são processados gerando uma estrutura

[12] http://lucene.apache.org/

de dados inter-relacionada eficiente para a pesquisa baseada em palavras-chave (*tokens*), denominada índice-invertido. A pesquisa por sua vez, consulta o índice a partir destas palavras-chave e organiza os resultados pela similaridade dos textos indexados com a consulta. O próprio *Lucene* possibilita a consulta e análise de similaridade textual.

Os itens de recomendação baseado em conteúdo são tratados como documentos do *Lucene*. Um documento do *Lucene* contém um ou mais campos. A Tabela 3.1 representa estes documentos. As linhas representam os documentos e as colunas os campos.

id	**Nome**	**Descrição**	**Preço**
10	Mini adaptador estéreo	Converte um mini pino 1/8 estéreo em pino duplo.	R$ 4.99
11	Bateria de 3V	Bateria simples de 3V, ideal para máquinas fotográficas.	R$ 12.29
12	Bateria de 9V	Bateria de 9V alcalina.	R$ 15.99

Tabela 3.1: Exemplo de 3 documentos gravados no *Lucene*.

3.3.1 Algoritmo *k-nearest neighbor*(*K-NN*)

Este foi um algoritmo implementado para realizar recomendações baseadas em conteúdo. O *K-Nearest Neighbor* é um método usado em várias aplicações para classificação textual [16]. Em resumo, classifica documentos de acordo com os K vizinhos mais próximos deste documento. Para utilizar o *K-NN* na produção de recomendações, foi utilizado o mesmo princípio, porém foram necessárias algumas adaptações para recomendar itens baseado na similaridade de seu conteúdo.

A recomendação é feita pela comparação dos campos de um determinado documento com os outros documentos indexados no *Lucene* utilizando-se a métrica *TF-IDF* [11, 17] que retorna um vetor de pesos de acordo com as palavras *(tokens)* do campo do documento. Em seguida calcula-se a distância entre este documento e outros documentos, sendo que para realizar o cálculo desta distância é utilizado o recurso do *framework Mahout* que possui as métricas de distância. A partir deste ponto, recomendam-se os K vizinhos mais próximos deste documento, sendo que quanto mais o valor de distância entre dois documentos se aproxima de zero, mais semelhantes são estes documentos. As métricas de distância, de forma não distribuída, implementadas no *Mahout* são:

CosineDistanceMeasure, EuclideanDistanceMeasure, ManhattanDistanceMeasure, SquaredEuclideanDistanceMeasure, TanimotoDistanceMeasure, WeightedEuclideanDistanceMeasure, WeightedManhattanDistanceMeasure.

3.3.2 Implementação dos *hot spots*

A implementação dos *hot spots* de cada setor para a aplicação de recomendação baseada em conteúdo são descritos a seguir. O detalhamento destes componentes é válido também para aplicações que possuem arquitetura distribuída, pois em geral têm a mesma finalidade dentro do processo de cálculo de recomendação.

Setor de *Batch*

O setor de *Batch* é responsável por produzir as recomendações e enviar a lista de itens recomendados ao o setor de *Cache*.

- *InstantiatorWorker:* Esta classe é uma subclasse de *AbstractInstantiatorWorker* e foi implementada de forma a manter o fluxo para que os outros componentes da aplicação baseada em conteúdo sejam carregados e para que seja instanciada a fachada do setor de *Batch*. Para fazer a instanciação é utilizado o arquivo de configuração com especificações de quais componentes devem ser instanciados. Este componente também realiza o carregamento dos campos estáticos para a classe *Constants*.

- *InputInterpreter:* Esta classe possui a função de interpretar a *string* de requisição de recomendações. Nesta aplicação, este setor recebe pedidos de recomendações feitos pelo setor de *Cache*, que será detalhado posteriormente. A entrada esperada é um *string* que possui o seguinte formato *ITEMID<sep>HOWMANY<sep>SepRecommendations<sep>SepBatch*. Ao separar a *string* de entrada de acordo com *<sep>*, que é o separador definido na classe *Constants*, obtém-se acesso as informações da requisição. O campo *ITEMID* é o identificador do item enviado que deverá ser único nos documentos indexados pelo *Lucene*. O campo *HOWMANY* indica a quantidade de itens requisitados. *SepRecommendations* é o separador de itens recomendados retornados para a fachada. Por exemplo, ao requisitar as recomendações e a fachada retorne uma *string* com o seguinte formato "itemId1,ItemId2,...", neste caso o *SepRecommendations* foi definido como uma vírgula. E *SepBatch* é o separador de recomendações de acordo com algum critério especificado pelo desenvolvedor de aplicação sobre o *IRF*.

- *InputBean:* Este componente é criado pelo interpretador de entrada toda vez que houver um pedido de recomendação feito para a fachada. Esta é uma subclasse de *BaseBean*. Como em aplicações de recomendação baseda em conteúdo, existe um item a ser comparado com outros existentes na base. O interpretador de entrada

configura o identificador do item a ser comparado com os outros, a quantidade de itens a serem recomendados, o separador dos identificadores dos itens que serão recomendados e o separador de lote. Estas informações são coletadas a partir da *string* recebida pela fachada e encapsuladas no *bean* criado.

- *RecommendedItem:* Esta classe implementa a interface *RecommendedItem* criada no *Mahout*. Este componente foi implementado para encapsular as informações a respeito dos itens recomendados. Estas informações são um identificador único do item e um campo numérico utilizado para armazenar o valor da distância de similaridade entre itens. O valor de similaridade é necessário para a ordenação dos itens, de forma a recomendar os mais relevantes e que portanto possuem um maior grau de similaridade.

- *Serializer:* Como já mencionado todos os valores que chegam e saem da fachada estão em formato de *strings*. Este componente foi implementado a fim de transformar em *string* as informações a respeito dos itens que serão recomendados. A lista de recomendações é serializada em uma *string* com o seguinte formato: "ItemId1<sep>ItemId2<sep>ItemId3...".

- *Storable:* Nesta aplicação, o setor de *Batch* possui uma instância do componente *Storable* do recomendador para que este possa ser colocado no componente *Cache*. Este componente é armazenado em *cache* utilizando o método *setData* do componente *Cache*, passando a propriedade de serialização em disco como *false*. O *CBStorableRecommender* não é serializado em disco, pois a construção dos recomendadores para métodos baseados em conteúdo é instantânea. Além disto, o *DataModel* utiliza o *Lucene* e este possui sua própria política de armazenamento de dados em disco. Existem classes do *Lucene* que não implementam a interface *Serializable* do Java e portanto os objetos não são passíveis de serem serializados no disco.

- *BatchProcessor:* Responsável pela realização do processamento mais custoso. Este componente dificilmente ficará ocioso, pois a todo tempo estará processando recomendações antes que estes pedidos sejam feitos pelos usuários. Este componente torna-se acessível remotamente via *RMI*, recurso disponível pela linguagem Java. Assim toda alteração na base implica em uma notificação para esta classe, indicando que novas recomendações devem ser processadas.

- *Controller:* Este componente controla o fluxo das recomendações neste setor. Foram implementados métodos de forma a recuperar o identificador e a quantidade

de recomendações do *bean* no parâmetro do método *getRecommendations* deste componente, que é invocado pela fachada. Nesta implementação este componente verifica se existe um recomendador no componente *Cache*. Se houver, o controlador requisita o processamento de recomendações. O controlador implementado também tem a finalidade de tratar as exceções quando estas são lançadas pelos componentes acionados por ele, e também por componentes que estão em um nível mais baixo, como por exemplo, os recomendadores ou o modelo de dados.

- *Recommender:* Para gerar recomendações baseadas em conteúdo, utiliza-se o método *recommend* da super-classe *IdealizeCBAbstractRecommender*. Este método recebe o identificador do item e a quantidade de recomendações a serem calculadas. O método *recommend* possui a seguinte assinatura *List<RecommendedItem>recommend(itemId, Howmany)*, onde o *itemId* é o identificador do item de referência e *HowMany* é a quantidade de recomendações que devem ser retornadas. Todos os algoritmos de recomendação utilizando esta abordagem, devem ser invocados a partir deste método. Os algoritmos de recomendação ordenam a lista de forma crescente em relação a distância de similaridade entre o item passado como parâmetro e os outros itens da base *Lucene*.

- *DataModel:* Esta é uma subclasse de *IdealizeDataModel*. O *DataModel* utilizado para esta aplicação possui uma instância do *Lucene* que é responsável por indexar os documentos a serem utilizados. Ao criar o *CBDataModel*, este recebe o caminho do diretório onde estão localizados os arquivos de índice do *Lucene*, definido no arquivo de propriedades. Quando os dados são alterados no *Lucene*, o *DataModel* da aplicação baseada em conteúdo também é alterado, pois o *CBDataModel* possui um objeto do *Lucene*.

Esta aplicação não utiliza *feedback*, pois os recomendadores baseados em conteúdo desconsideram informações de *ratings* enviados aos itens.

Setor de *Cache*

Este setor é responsável por responder diretamente às requisições dos usuários ou às aplicações que requisitam recomendações. Como as recomendações são cacheadas, esta resposta é provida em *O(1)* [13]. Para a aplicação baseada em conteúdo, a fachada do setor de *Cache* e a fachada do setor de *Batch* possuem alguns componentes com a mesma implementação. Estes componentes são *Serializer, RecommendeItem* e *InputInterpreter*,

que apesar de serem utilizados neste setor não estarão descritos nesta seção, uma vez que já foram detalhados.

- *InstantiatorWorker:* Este componente realiza a instanciação dos componentes deste setor de acordo com o arquivo de propriedades. A instanciação recupera o endereço de IP da máquina onde a fachada do setor de *Batch* foi registrada, para que esta possa ser acessada remotamente. Ao acessar a fachada *Batch* remotamente, os pedidos de recomendações são enviados a ela para posteriormente serem armazenados em memória principal por este setor de *Cache*.

- *Storable:* O *Storable* deste setor possui recomendações pré-calculadas. Estas recomendações são armazenadas em um *HashMap* que contém os identificadores dos itens como chave e uma lista de recomendação contendo o identificador de outros itens que estão sendo recomendados baseado no item contido na chave. O tamanho da lista que contém os itens recomendados a serem colocados em *cache* é definido no arquivo de configuração. Esta estrutura é mantida em memória principal e os itens são acessados com complexidade de tempo *O(1)*.

- *Controller:* Este componente foi implementado com a função de controlar o fluxo das recomendações no setor de *Cache*. Quando chega um pedido de recomendação na fachada, esta requisição é passada para o controlador. Este último por sua vez verifica se há recomendações calculadas em *Cache* para aquele item. Se não houver, a requisição é feita para a fachada do setor de *Batch*. Assim que a recomendação é calculada, é armazenada em *Cache* para atender futuras requisições de recomendação para aquele item.

- *BatchProcessor:* Este componente requisita recomendações para a fachada do setor de *Batch*. Toda vez que chegar um pedido de recomendação para este componente, este repassará este pedido para a fachada remota e quando recebida a lista de itens recomendados, estes são armazenados na estrutura de *cache*.

O setor de *cache* não possui implementação para os *hot spots DataModel, DataManipulator, Recommender* e *InputController*. Outros setores são responsáveis por implementar e utilizar estes componentes quando se implementa aplicações baseadas em conteúdo sobre o *IRF*.

Setor de *Input*

Este setor é responsável por realizar as manipulações dos dados inseridos, atualizados, ou removidos no *Lucene*. Na maioria das vezes ocorre a pré-indexação de documentos de uma base de dados antes mesmo que os dados passem por este setor de *Input*. Quando a máquina de recomendação está executando, este é o setor responsável por remover, atualizar ou inserir novos documentos no *Lucene*, e consequentemente no *DataModel* da aplicação baseada em conteúdo.

- *InputInstantiatorWorker:* Este componente trabalha basicamente da mesma forma que em outros setores, portanto tem o objetivo de instanciar os componentes necessários para este setor, carregando as classes concretas e os atributos necessários para a classe *Constants* a partir do arquivo de configuração.

- *DocInterpreter:* Tem a finalidade de interpretar a entrada de documentos a serem modificados, removidos ou inseridos no *Lucene*. Os dados a serem atualizados são passados em forma de *string* pela fachada. Este componente então interpreta esta *string* de forma a devolver um *InputBean* para a fachada que passará o fluxo para o *InputController*. Um exemplo de entrada para este interpretador: *FIELD_NAME = VALUE_FIELD<sep>TYPE = TYPE_FIELD<sep>FIELD_NAME = VALUE_FIELD<sep>TYPE = TYPE_FIELD....* Esta *string* possui a quantidade de campos que contém um documento *Lucene* criado. O *<sep>* será o separador que foi definido no arquivo de configurações. O tipo do campo indica se é um campo do tipo textual ou palavra-chave, informações estas utilizadas pelo *Lucene*.

- *InputBean:* Os campos do objeto recebem os valores a partir da *string* que chega na fachada deste setor, encapsulando estas informações em um único objeto. O *InputBean* implementado para a aplicação de recomendação baseada em conteúdo possui o identificador do item, e além deste atributo possui um *HashMap* onde as chaves são os nomes dos campos e os valores, são os valores dos campos do documento que será indexado pelo *Lucene*.

- *InputController:* Este é o controlador do setor de *Input*. Possui um *BatchProcessor* remoto referenciando o componente que está no setor do *Batch* e outro que está no setor de *Cache*. O controlador passa o fluxo da aplicação para o *DataManipulator* que sabe como trabalhar com estas novas informações. Os documentos são inseridos em uma lista no *DataManipulator*, que ao atingir uma determinada quantidade de documentos grava os documentos na base de dados (*Lucene*) e em seguida o controlador notifica os *BatchProcessor's* remotos a respeito das alterações na base de dados.

- *DataManipulator:* O *DataManipulator* é o componente responsável por inserir, remover ou atualizar os dados no *Lucene*. Para isto foi implementado um objeto que contém o documento a ser modificado e qual operação será realizada (inserção, remoção ou alteração) sobre o documento no *Lucene*.

Capítulo 4

Idealize Recommendation Framework Distribuído e Escalável

Levando em consideração que muitos problemas implementados de forma sequencial diferem de implementações distribuídas para o mesmo problema, ainda assim, muitos algoritmos podem ser estruturados para trabalhar de maneira distribuída. Desta forma, muitos algoritmos de recomendação que operam de forma sequencial podem ser estruturados para que os cálculos das recomendações sejam realizados de forma distribuída.

4.1 Avaliando recursos de uma arquitetura distribuída e uma arquitetura pseudo-distribuída

Em alguns casos a arquitetura pseudo-distribuída é suficiente para a implementação da aplicação de recomendação, levando em consideração o tamanho da base de dados a ser utilizada. Porém, dependendo do volume de dados a ser processado o tempo de processamento das recomendações torna-se muito elevado. Além disto, o volume de dados pode impossibilitar a operação da máquina de recomendação utilizando a arquitetura pseudo-distribuída.

Muitas vezes o grande volume de dados ultrapassa o tamanho máximo de *heap* que pode ser configurado para uma máquina virtual Java. Assim, a escolha da arquitetura deve ser feita de forma cautelosa, pois em algum momento a quantidade de *ratings* ou itens poderá ultrapassar a quantidade de memória disponível em uma só máquina dentro de uma arquitetura pseudo-distribuída. Este problema pode ser resolvido com uma arquitetura distribuída, porém ainda assim, deve-se pensar em vantagens e desvantagens de um sistema distribuído e escalável, pois muitas vezes um sistema que realiza muitos

acessos ao disco (como no caso do *Hadoop*), poderá tornar-se menos eficiente que um sistema que mantém os dados em memória principal. Outra questão que deve ser levada em consideração nesta escolha é a questão financeira.

Muitas vezes a utilização de uma máquina com alto poder de processamento e armazenamento possui custo benefício inferior à obtenção de máquinas de menor custo, mas que podem ser arranjadas na forma de um *cluster*. Ainda assim, esta única máquina acaba se tornando um ponto único de falha, o que pode levar todo o sistema a falhar quando esta máquina falhar [14]. Em geral quando se utiliza recursos de *hardwares* separados que operam de maneira conjunta, é garantida uma maior segurança devido as políticas de transparência[1] [18] implementadas em sistemas distribuídos.

Em resumo deve-se verificar as características do sistema de recomendação e avaliar as vantagens e desvantagens de cada arquitetura.

4.2 Descrição dos componentes do pacote *distributed_hot_spots*

Quando se implementa uma aplicação distribuída, além de utilizar as classes dos pacotes *base* e *hot_spots*, deve-se utilizar as classes do pacote *distributed_hotspots*. As classes deste pacote são específicas para aplicações distribuídas, onde a aplicação para realização dos cálculos de recomendação é executada sobre o *Cluster Hadoop HBase*.

As classes do pacote *distributed_hotspots* são:

IdealizeAbstractJob, IdealizeConfiguration, IdealizeHBaseConfiguration, IdealizeCFAbstractJob, IdealizeAbstractDistributedRecommender, InputBeanDistributed, IdealizeHBaseDataModel, IdealizeCFHBaseDataModel, CreateDistributedDataModel, IdealizeWritable.

A Figura 4.1 apresenta a arquitetura alto nível dos componentes utilizados na aplicação que é executada no *cluster*. Os dois círculos na Figura 4.1 representam os métodos de acesso externo ao *cluster*. Cada um dos desenhos em formato de pasta representam os pacotes que armazenam as classes abstratas ou interfaces que devem ser implementadas. Assim, dentro de cada pacote há uma superclasse que deve ser implementada, estas superclasses a serem implementadas estão representadas na Figura 4.1 pelos retângulos com os nomes em negrito.

[1] Com maior importância para a replicação de dados

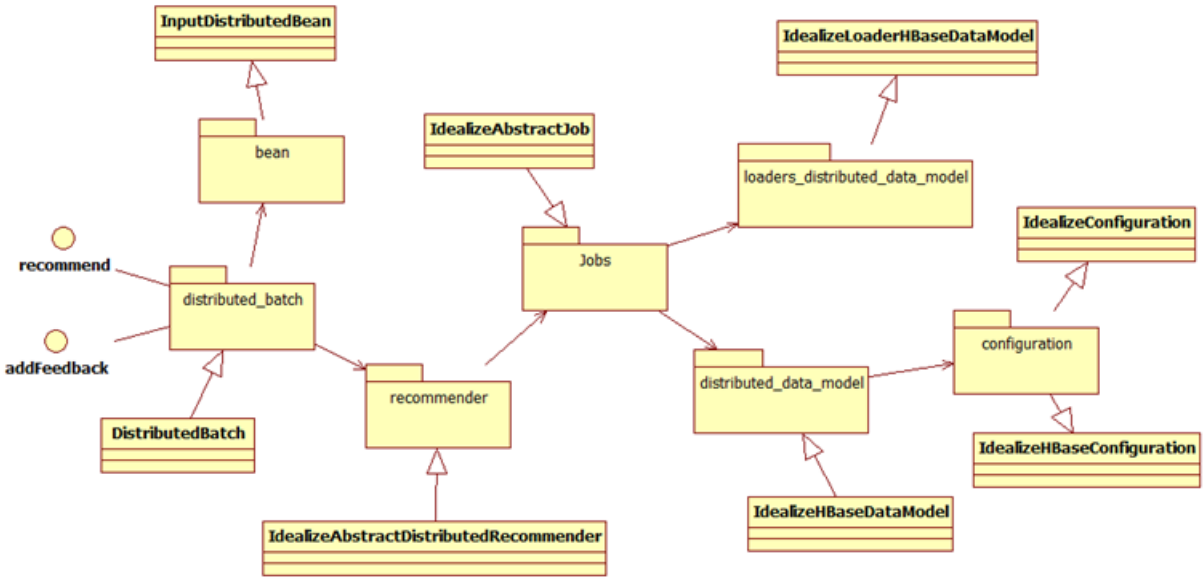

Figura 4.1: Arquitetura alto nível da aplicação executada no *cluster*

Descrição dos componentes utilizados em aplicações distribuídas.

- *IdealizeAbstractJob:* Este é um componente utilizado pelas diferentes estratégias de recomendação distribuída. As subclasses deste componente são responsáveis por realizar os cálculos de recomendações de forma distribuída, ou seja, são definidas as sequências de *Maps* e *Reduces* que devem ser executados. Toda aplicação de recomendação distribuída deverá ter pelo menos uma implementação desta classe.

- *IdealizeConfiguration:* Este componente também é uma interface de marcação, assim como vários outros componentes aqui descritos. Uma implementação deste componente é onde o desenvolvedor deverá definir o tipo de configuração que será utilizada na recomendação distribuída, setando quantos *MapReduces* serão utilizados, quais os locais dos arquivos de entrada e saída, entre outras propriedades de configuração da tarefa a ser realizada. Desta forma, fica a cargo do desenvolvedor definir quais serão os parâmetros de configuração utilizados para o processamento de recomendação de maneira distribuída.

- *IdealizeHBaseConfiguration:* Esta classe é uma subclasse de *IdealizeConfiguration*, e possui as configurações referentes à utilização de um modelo de dados baseado no *HBase*. Possui métodos que permitem a criação de tabelas no *HBase*, a recuperação de *column families* de uma determinada tabela e também a operação

sobre as estruturas das tabelas, tais como quantidade de *column families*, nomes das *HTables*.

- *IdealizeAbstractDistributedRecommender:* Este componente possui a mesma finalidade de *IdealizeRecommender* e portanto é uma subclasse de *IdealizeRecommender*. Porém, este componente é específico para recomendadores distribuídos, uma vez que são mantidos nesta classe os atributos e métodos voltados para este tipo de arquitetura. Todo método de recomendação distribuído deve ter pelo menos um objeto do tipo *IdealizeAbstractJob* para executar e portanto esta classe possui este objeto.

- *IdealizeHBaseDataModel:* Componente definido para os recomendadores que utilizam o *HBase* como modelo de dados distribuído. As subclasses desta devem conter uma lista de *HTables* e métodos que sejam capazes de operar sobre estas tabelas. Assim ao implementar subclasses deste componente, é definida uma lista de tabelas que serão utilizadas pelo recomendador.

- *DistributedBatch:* Este componente deve ser implementado para que seja realizada a comunicação com o recomendador distribuído, logo, a política de processamento em lote para cálculos de recomendações faz parte deste componente. Assim, este componente se torna um intermediador entre o *cluster* e os componentes do setor de *Batch* propriamente implementados.

- *InputBeanDistributed:* Este é uma subclasse de um *Bean* do pacote *base*. Este componente, assim como os outros, foi implementado a fim de manter características comuns para as aplicações de recomendação distribuídas.

- *LoaderToHBaseDataModel:* Neste componente são definidos métodos que permitem o carregamento de dados para o *HBase*. Assim, a política de carregamento de dados para o *HBase* deverá estar em uma classe que seja implementação desta superclasse

4.3 Aplicação distribuída baseada em filtragem colaborativa

Nesta seção descrevo quais recursos e componentes foram utilizados para a implementação da aplicação de recomendação distribuída utilizando a abordagem por filtragem colaborativa.

Como descrito anteriormente, a abordagem de filtragem colaborativa tem a sua origem na mineração de dados [3]. Esta abordagem de recomendação leva em consideração a ação de múltiplos usuários [15, 19, 7].

Alguns componentes criados para o módulo distribuído do *IRF* foram baseados na implementação de classes do *Apache Mahout*. Os recomendadores distribuídos do *Mahout* utilizam diretamente o *Hadoop Distributed File System*, o que muitas vezes impossibilita a atualização dos *ratings* presentes na base de dados, visto que a política do *HDFS* é voltada para grandes arquivos que não devem ser modificados. Em sistemas de recomendação a todo tempo os usuários enviam *ratings* aos novos itens, desta forma se faz necessário a modificação nos dados da base utilizada.

4.3.1 Descrição dos componentes distribuídos implementados

Descrição dos componentes implementados para a aplicação de recomendação distribuída baseada em filtragem colaborativa.

- *IdealizeAbstractJob:* Os *Jobs* implementados definem a sequência de *Maps* e *Reduces* a serem executados. Foram implementados *Jobs* para os três métodos de recomendação distribuídos (*MostPopularJob, MostPopularOverRatingJob, ItemBasedSimilarityJob*), definindo assim a sequência de *Map* e *Reduce* para cada um deles. Para o método *ItemBasedSimilarityJob*, foi implementado um *job* adicional chamado *IdealizeRowSimilarityJob*, que tem como finalidade realizar cálculos de similaridade entre os dados de entrada. Estes dados de entrada podem ser usuários ou itens.

- *IdealizeConfiguration:* Para os métodos de recomendação distribuídos foi implementado uma classe de configuração que utiliza o objeto *Configuration* do *HBase*. Este objeto permite configurar valores de atributos de forma que estes possam ser recuperados nos *Maps* e *Reduces* a serem utilizados. O objeto *configuration* possui o valor de quantos *ratings* máximo por usuário será utilizado, qual a classe de similaridade será utilizada.

- *IdealizeHBaseDataModel:* Este é o modelo de dados utilizado pelos métodos de recomendação distribuídos implementados. Possui duas tabelas, uma tabela que tem como chave os identificadores de usuários e seus respectivos itens e valores de ratings. Uma segunda tabela que tem como chave os identificadores dos itens e seus valores sendo os identificadores de usuários e valores de *ratings* dados por estes usuários. Possui também métodos que facilitam a recuperação destas tabelas e a recuperação das estruturas destas tabelas. Para a aplicação de recomendação por filtragem colaborativa um modelo de dados com estas duas tabelas é criado, pois alguns métodos utilizam os dados do formato da primeira tabela e outros métodos utilizam do formato da segunda tabela para calcular as recomendações.

- *IdealizeAbstractDistributedRecommender:* Uma implementação deste componente é a classe utilizada por recomendadores de abordagem por filtragem colaborativa. Todos os métodos desta abordagem possuem a mesma assinatura de método para pedidos de recomendações e pelo menos um objeto do tipo *IdealizeCFAbstractJob*. A classe tem a finalidade de manter valores que são comuns para os métodos recomendação distribuídos que utilizam a abordagem por filtragem colaborativa.

- *DistributedBatch:* O *DistributedBatch* implementado para esta recomendação realiza a comunicação com o método de recomendação distribuído *IdealizeDistributedCFRecommender*, e consequentemente com o *cluster*, uma vez que o método está sendo executado no *cluster*. Assim que as recomendações são recebidas por este componente, estas são repassadas para o controlador do setor de *Batch*, e a partir daí o fluxo segue normalmente como em uma arquitetura pseudo-distribuída.

- *IdealizeWritable:* As classes implementadas a partir deste componente podem ser vistas como classes de utilidade geral, uma vez que podem ser utilizadas por quaisquer *Maps* e *Reduces*. Os *Writables* implementados foram *IdealizeWeightRowPair, IdealizeVectorOrPrefWritable, IdealizeWeightOccurrence* e *PrefAndSimilarityColumnWritable*. Estas classes são utilizadas pelos *Maps* e *Reduces* dos métodos de recomendação por filtragem colaborativa. Estas classes têm a finalidade de serializar objetos em disco pelas classes *TableMap*, de forma que estes possam ser recuperados e utilizados na fase do *TableReduce*.

- *LoaderToHBaseDataModel:* Uma classe implementada a partir desta classe abstrata é chamada *LoaderCFDistributedDataModel*, que tem por finalidade criar o modelo de dados usado para a recomendação por *CF*. Nesta classe é realizado o

carregamento dos dados a partir de um SGBD[2] para o *HBase*. Quando estes valores são gravados no *HBase* uma vez, toda nova inserção, atualização ou remoção ocorre diretamente sobre o próprio *HBase*.

4.3.2 O método de recomendação distribuído *MostPopularOverRating*

Foram implementados três métodos de recomendação distribuídos: *MostPopularJob*, *MostPopularOverRatingJob* e *ItemBaseSimilarityJob*. Aqui descrevo o *MostPopularOverRatingJob*, por ser simples e com a explicação deste método é possível entender o modelo de implementação dos métodos de recomendação distribuídos.

Este método possui somente um *job* e portanto somente um *TableMap* e *TableReduce*, o que não acontece com o *ItemBaseSimilatirityJob*, que precisa de cinco *TableMaps* e cinco *TableReduces* para ser executado. Não existe um padrão no número de *MapReduces* a ser implementado, pois este número varia de acordo com o funcionamento de cada método de recomendação.

O funcionamento deste método segue em princípio a mesma idéia de um contador de palavras, pois o que é feito é a contagem de quantos usuários rankearam o mesmo item. Neste método só são considerados os *ratings* que estão acima de um determinado valor passado como parâmetro. Supondo que os *ratings* variam de 1 a 5, como exemplo, poderíamos contar somente os *ratings* com um valor maior ou igual a 4, recomendando os itens que mais receberam *ratings*.

[2]Sistema de Gerenciamento de Banco de Dados

Algoritmo 4.1: MostPopularOverRatingMap

```
1  @Override
2  public void map(ImmutableBytesWritable row, Result values,
3                  Context context) throws IOException {
4
5          LongWritable itemId = new LongWritable();
6          List<KeyValue> prefsItemIds = values.list();
7          int size = prefsItemIds.size();
8          List<KeyValue> listItems = values.list();
9
10         try {
11         // runs through all values of a given registry
12                 for (int i = 0; i < size; i++) {
13                         itemId.set(Bytes.toLong(
14                                         listItems.get(i).getQualifier()));
15                         float value = Bytes.toFloat(
16                                         listItems.get(i).getValue());
17
18                         if (value >= this.valueRating)
19                         //record this value as input to Reduce
20                                 context.write(itemId, one);
21                 }
22         } catch (InterruptedException e) {
23                 throw new IOException(this.canonicalName
24                           + ".Map: Could not possible complete Map:"
25                           + e.toString());
26         }
27 }
```

A entrada para este método (*K1, V1*) são os seguintes valores:

K1 é o valor do identificador de um determinado usuário, *V1* são os identificadores dos itens e valores de *ratings* dados a estes itens pelo usuário *K1*. Na linha 8 os identificadores de itens e valores de *ratings* são convertidos em uma lista. Na linha 12 pode-se observar o início de um *loop for* que realiza a iteração sobre a lista, verificando se o valor de cada *rating* é maior ou igual ao valor do *rating* desejado. Se o valor do *rating* para o item for maior que o valor desejado, então é gravado como chave de saída do *Map* o valor do identificador do item e o valor *one*. Como os objetos de saída do *Map* devem ser do tipo *Writable*, o valor do objeto *one* é do tipo *IntWritable* e tem seu valor de inteiro setado para 1. Assim, como todos os objetos de saída do *Map* devem ser do tipo *Writable*, o identificador do item é um objeto do tipo *LongWritable*. Após a fase do mapeamento o *Hadoop* realiza a ordenação e agrupamento de todos os valores que possuem o mesmo

valor de chave (*K2*). No caso deste algoritmo são agrupados e ordenados pelo *Hadoop* todos os itens que possuem o mesmo valor de identificador. De acordo com a sequência da Figura 2.1, observamos que os valores de saída do *Map* tornam entrada para o *Reduce*.

Algoritmo 4.2: MostPopularOverRatingReduce

```
1  /**
2   * This is the class reduce used to store how many times
3   * the items were ranked.
4   *
5   * @author Alex Amorim Dutra
6   */
7  public static class DistributedMostPopularOverRatingReducer
8          extends TableReducer<LongWritable, IntWritable,
9          ImmutableBytesWritable> {
10
11         @Override
12         public void reduce(LongWritable key,
13                   Iterable<IntWritable> values, Context context)
14                   throws IOException, InterruptedException {
15
16                Long id = key.get();
17                ImmutableBytesWritable itemID =
18                          new ImmutableBytesWritable(Bytes.toBytes(0));
19
20                int sum = 0;
21
22                // make the counting
23                for (IntWritable itr : values)
24                         sum++;
25
26                Put put = new Put(Bytes.toBytes(id));
27                put.add(Bytes.toBytes("timesValueItemId"),
28                          Bytes.toBytes(id), Bytes.toBytes(sum));
29                context.write(itemID, put);
30         }
31 }
```

No código 4.2, a entrada é (*K2, V2*), onde *K2* é o identificador de um item. *V2* é um *Iterator* de valores do tipo *IntWritable*. Ao receber a entrada no código do *TableReduce*, no *loop for* da linha 23 é realizada a iteração sobre os valores. Para cada um dos valores do *iterator* é somado 1 na variável *sum*, indicando quantos *ratings* o item recebeu. Por fim, na linha 26 é criado um objeto do tipo *Put*, que é um objeto de armazenamento em uma *HTable* do *HBase*, adicionado a este objeto o identificador do item e quantos *ratings* ele recebeu. Em seguida este objeto é gravado pelo *Hadoop* na respectiva *HTable*.

Capítulo 5

Experimentos

5.1 Descrição da base de dados

A base de dados utilizada para realização dos experimentos foi disponibilizada pela empresa *Netflix*. *Netflix* é uma empresa americana que permite aos usuários assistir transmissões tanto de programas de TV quanto filmes pela internet[1].

A base de dados foi disponibilizada pela *Netflix*, durante um concurso criado pela empresa no ano de 2006, no qual o prêmio de US$1.000.000 foi concedido à equipe que desenvolveu um algoritmo de recomendação 10% mais eficaz que aquele criado pela própria empresa.

Para o presente trabalho, foi utilizada uma tabela desta base que contém um total de 96.721.844 (aproximadamente 97 milhões) de *ratings*, 480.189 usuários e 17.770 filmes, porém, foram executadas sequências de testes variando a quantidade de *ratings* da base de dados. Os dados da tabela de entrada encontram-se no seguinte formato *<id usuário>:<id do item><valor de rating>*, onde os valores de *rating* variam entre 1 e 5.

5.2 Resultados obtidos

Para realização dos experimentos foram utilizados os seguintes softwares:

Sistema Operacional Linux Ubuntu 10.10
HBase versão 0.90.4
Hadoop versão 0.20.205

[1] www.netflix.com

Ecplise Galileo versão 3.5

Os computadores utilizados no *cluster* possuem as seguintes configurações de *hardware*:
*P*rocessador AMD Athlon Dual Core 5400B - 1000 MHz
Memória *RAM 2GB*

Os testes foram executados em um *cluster* contendo 18 máquinas, porém, também foram executadas sequências de testes utilizando tamanhos diferentes do *cluster*.

O método utilizado para realização dos experimentos foi o *ItemBasedSimilarity*. A escolha deste método deve-se à possibilidade de realizar comparações com a implementação existente no *Mahout*. Desta forma, foi possível realizar a comparação de um método que utiliza o *HBase* com um método do *Mahout* que utiliza diretamente o *HDFS*.

O método de recomendação *ItemBasedSimilartiy* recebe três atributos como parâmetros: tipo de similaridade entre os itens, quantidade máxima de *ratings* por usuário e quantidade máxima de itens que co-ocorrem. Os itens são co-ocorrentes quando são *rankeados* pelo mesmo usuário.

As métricas de similaridade para métodos de recomendação distribuídos implementadas no *Mahout* são: *CoocurrenceDistributedMeasure, EuclideanDistributedMeasure, PearsonDistributedMeasure, LoglikelihoodDistributedMeasure* e *TanimotoDistributedMeasure*.

A similaridade de itens utilizada no método *ItemBasedSimilarity* para a execução dos experimentos foi a métrica de *Pearson*. O atributo referente à quantidade máxima de *ratings* de um usuário foi configurado com o valor 100. O número máximo de co-ocorrência de itens também foi definido com o valor 100.

Os resultados obtidos e apresentados são referentes ao tempo de execução para a realização do cálculo de recomendações para todos os usuários pertencentes à base. Foram realizados três testes para a mesma configuração referente ao número de máquinas e o tamanho da base, em seguida calculado o valor médio entre os três testes realizados para a mesma configuração do *cluster* e da base.

A próxima etapa dos experimentos foi realizada fixando-se o número de máquinas em 18 e variando-se o tamanho da base. Os resultados apresentados na Tabela 5.1 foram obtidos executando o método implementado sobre o *IRF*, que utiliza o *HBase* como armazenamento de dados. O *cluster* foi configurado com 18 máquinas e variada a quantidade de *ratings* conforme apresentado na tabela.

18 Máquinas - Tempo em minutos (IRF)				
Ratings	**Teste 1**	**Teste 2**	**Teste 3**	**Média**
30 Milhões	140,62	138,46	136,24	138,44
60 Milhões	148,98	143,56	142,19	144,91
90 Milhões	152,12	153,45	158,79	154,79

Tabela 5.1: Resultados obtidos variando a quantidade de *ratings* (*IRF*)

Como pode-se observar o resultado do tempo obtido varia para a mesma configuração do *cluster* e da base de dados. Esta variação deve-se a alguns fatores, sendo eles: gerenciamento de memória do sistema operacional, comunicação de rede entre as máquinas, políticas de balanceamento de carga do *Hbase*, políticas de cache do *Hadoop* e *Hbase*.

Os resultados apresentados na Tabela 5.2 foram obtidos executando o método implementado sobre o *Mahout* e portanto utiliza diretamente o *Hadoop Distributed File System* (*HDFS*). O *cluster* foi configurado com 18 máquinas e variada a quantidade de *ratings* conforme apresentado na Tabela 5.2.

18 Máquinas - Tempo em minutos (Mahout)				
Ratings	**Teste 1**	**Teste 2**	**Teste 3**	**Média**
30 Milhões	83,14	79,22	78,81	80,39
60 Milhões	111,32	105,59	104,76	107,22
90 Milhões	126,57	131,12	127,29	128,33

Tabela 5.2: Resultados obtidos variando a quantidade de *ratings* (*Mahout*)

Como era de se esperar o tempo aumenta à medida que aumenta o número de *ratings* utilizados para realizar os cálculos de recomendações. A Tabela 5.3 apresenta o resultado comparativo entre os valores obtidos no *Mahout* e *IRF* para a variação do número de *ratings*. Podemos observar em comparação dos resultados da Tabela 5.3 e da Figura 5.1 que o *IRF* apresentou um tempo com menores alterações a medida que aumenta-se o número de *ratings*. Este fato é explicado devido a classe do *IRF* que realiza a poda dos itens co-ocorrentes ser a primeira classe executada, enquanto no *Mahout* a classe que realiza a poda dos itens co-ocorrentes é executada somente depois da execução de quatro tarefas *MapReduces*. Desta forma o *Mahout* executa mais tarefas com uma quantidade maior de dados até que a poda destes itens seja realizada.

***IRF X Mahout* 18 Máquinas - Tempo em minutos**			
Ratings	**30 Milhões**	**60 Milhões**	**90 Milhões**
IRF	138,44	144,91	154,12
Mahout	80,39	107,22	128,33

Tabela 5.3: Resultados obtidos variando a quantidade de *ratings* (*IRF X Mahout*)

A Figura 5.1 representa a comparação entre os resultados obtidos executando o *Mahout* e o *IRF* variando a quantidade de *ratings*.

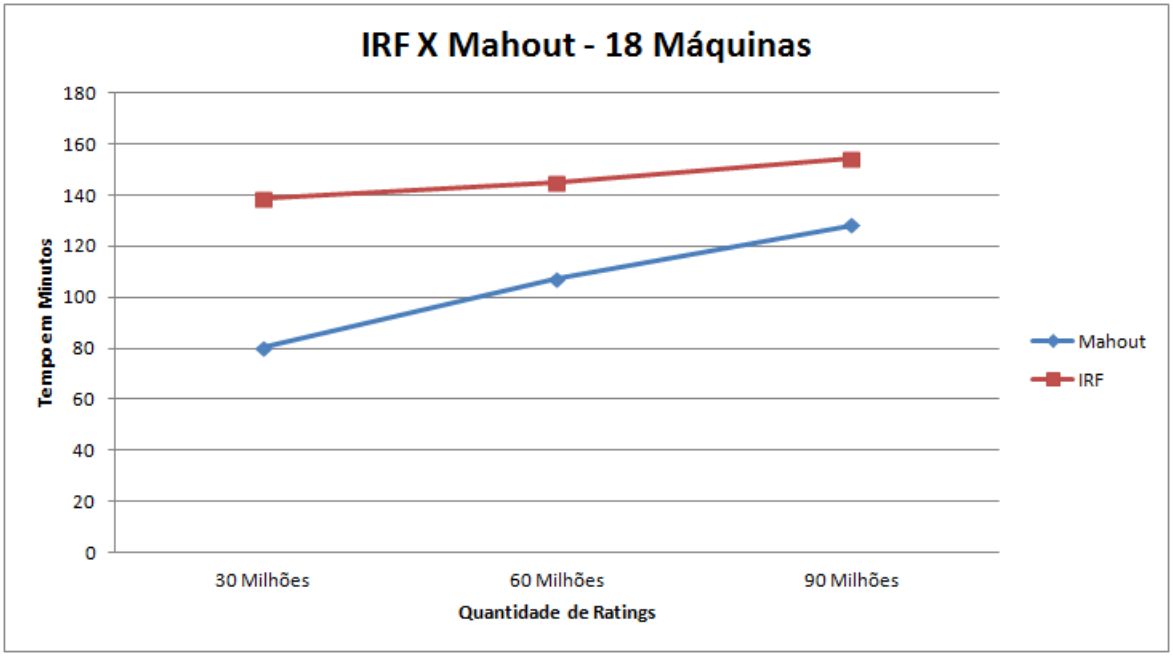

Figura 5.1: Resultados obtidos variando a quantidade de *ratings* (*IRF X Mahout*)

A próxima etapa dos experimentos foi realizada fixando-se o tamanho da base em 90 milhões de *ratings* e variando-se o número de máquinas. Os resultados apresentados na Tabela 5.4 foram obtidos executando o método *ItemBasedSimilarity* implementado sobre o *IRF*. A quantidade de máquinas foi configurada no *cluster* conforme apresentado na Tabela 5.4.

90 Milhões de *ratings* - Tempo em minutos (IRF)				
Ratings	**Teste 1**	**Teste 2**	**Teste 3**	**Média**
12 Máquinas	178,68	182,13	179,56	179,98
15 Máquinas	159,23	164,45	163,27	162,32
18 Máquinas	152,12	153,45	158,79	154,79

Tabela 5.4: Resultados obtidos variando a quantidade de máquinas (*IRF*)

Os resultados apresentados na Tabela 5.5 foram obtidos executando o método implementado sobre o *Mahout*. Conforme apresentado na tabela foi variada a quantidade de máquinas no *cluster*.

90 Milhões de *ratings* - Tempo em minutos (Mahout)				
Ratings	**Teste 1**	**Teste 2**	**Teste 3**	**Média**
12 Máquinas	147,12	143,45	152,79	147,78
15 Máquinas	136,58	133,76	134,47	134,93
18 Máquinas	126,57	131,12	127,29	128,33

Tabela 5.5: Resultados obtidos variando a quantidade de máquinas (*Mahout*)

A Tabela 5.6 apresenta o resultado comparativo entre os resultados obtidos para o *Mahout* e o *IRF*, variando a quantidade de máquinas.

***IRF X Mahout* 90 Milhões de *ratings* - Tempo em minutos**			
Máquinas	**12 Máquinas**	**15 Máquinas**	**18 Máquinas**
IRF	179,98	162,32	154,79
Mahout	147,78	134,93	128,33

Tabela 5.6: Resultados obtidos variando a quantidade de *máquinas* (*IRF X Mahout*)

De acordo com resultados apresentados na Tabela 5.6, pode-se observar uma melhoria no tempo de execução à medida em que executa-se os experimentos em um *cluster* com uma quantidade maior de máquinas. Os mesmos dados estão apresentados graficamente na Figura 5.2.

Observa-se que o método provido pelo *Mahout* apresentou melhores resultados também para um número menor de máquinas. Estes resultados foram obtidos pelo fato de o *HDFS* não possuir políticas de indexação de dados, assim como o *HBase* possui. Vale

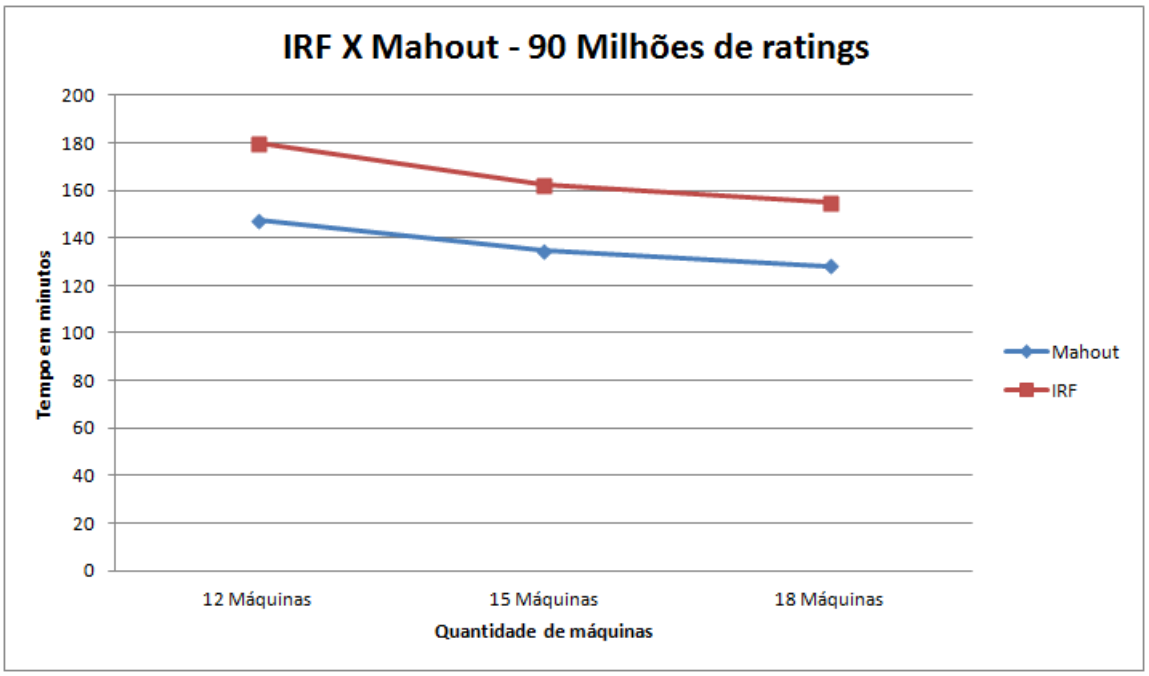

Figura 5.2: Resultados obtidos variando a quantidade de máquinas (*IRF X Mahout*)

lembrar que, no caso do *IRF* a resposta de recomendação aos usuários é instantânea, uma vez que o *IRF* possui o setor de *Cache* onde armazena recomendações pré-calculadas [13].

O que justifica a utilização do *HBase* é o fato de possuir indexação dos dados. Em sistemas de recomendação, uma vez que estes dados estão indexados, é permitida a atualização em tempo real do modelo de dados utilizado pelo método de recomendação. Ao falar sobre atualização em tempo real em sistemas de recomendação, refere-se principalmente à inserção de novos *ratings*, remoção de *ratings* e atualização de *ratings* existentes na base de dados.

A atualização da base de dados é imprescindível em sistemas de recomendação, uma vez que os usuários a todo tempo realizam operações sobre este tipo de aplicação, sendo assim, todas estas operações geram impactos diretamente no modelo de dados utilizado pelo método de recomendação.

Na estrutura dos métodos de recomendação distribuídos existentes no *Mahout*, não tem a possibilidade ou métodos que permitam fazer a atualização dos dados sobre o arquivo de entrada que está no *HDFS* (inserção de *ratings*, atualização de *ratings*, remoção de *ratings*). Imagine um sistema de recomendação onde a todo tempo os usuários desejam enviar novos *ratings*, atualizar *ratings* antigos ou remover *ratings*, e todos estes dados estejam em uma estrutura sem qualquer tipo de indexação, assim como em um

simples arquivo texto. No *Mahout*, para modificar a base de dados de forma que esta utilize as novas informações, o que deve ser realizado é o pré-processamento gerando uma nova base de dados, esta por vez é armazenada como entrada no *HDFS*.

A próxima etapa dos experimentos foi realizada modificando a base de dados. Os resultados apresentados na Tabela 5.7 são referentes ao tempo de inserção, atualização e de remoção de *ratings* no modelo de dados utilizado pelo recomendador implementado sobre o *IRF*.

***IRF* 90 Milhões de *ratings* 18 Máquinas**			
	Inserção	**Atualização**	**Remoção**
1000 Ratings	153 segundos	364 segundos	217 segundos
3000 Ratings	417 segundos	1089 segundos	660 segundos
6000 Ratings	822 segundos	2148 segundos	1290 segundos

Tabela 5.7: Resultados obtidos referente a modificações na base de dados (*IRF*)

A Tabela 5.8 apresenta a média de cada uma das operações realizadas no modelo de dados.

***IRF* 90 Milhões de *ratings* 18 Máquinas**			
	Inserção	**Atualização**	**Remoção**
1000 Ratings	153 ms/rating	364 ms/rating	217 ms/rating
3000 Ratings	139 ms/rating	363 ms/rating	220 ms/rating
6000 Ratings	137 ms/rating	358 ms/rating	215 ms/rating

Tabela 5.8: Resultados obtidos em milisegundos por *rating* modificado no modelo de dados

A Figura 5.3 representa os resultados apresentados na Tabela 5.6.

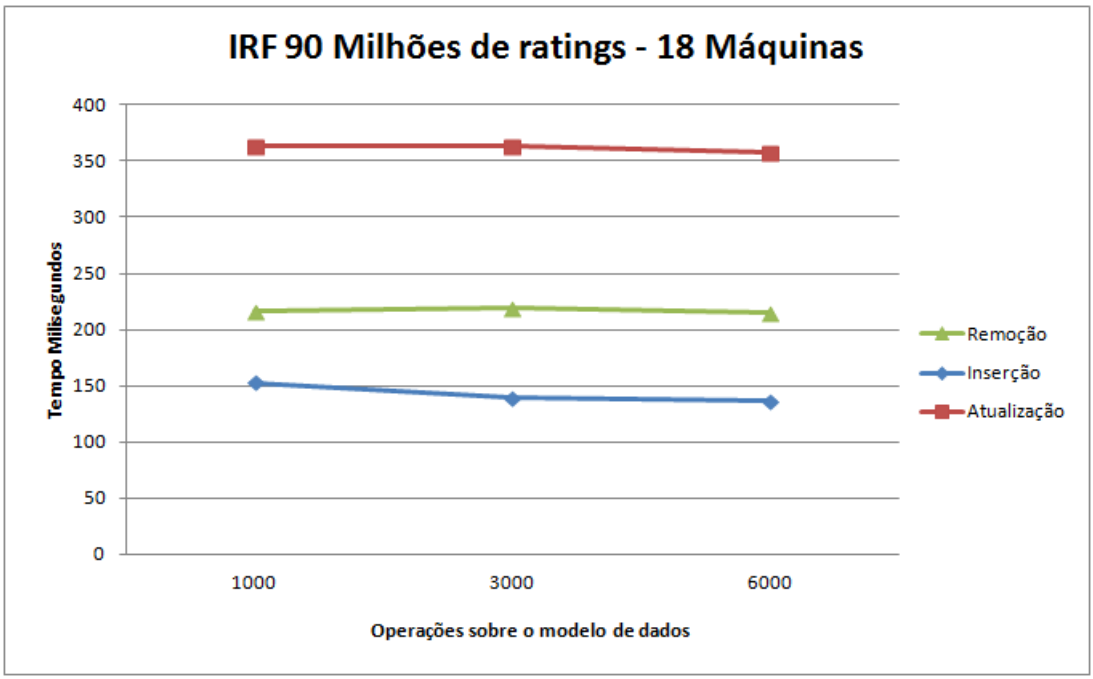

Figura 5.3: Resultados obtidos para modificações realizadas no modelo de dados

Como se pode observar na Tabela 5.8 e na Figura 5.3 o tempo médio por item sendo modificado no modelo de dados é independente da quantidade de itens. O pior tempo de operação encontrado foi o tempo de realizar um *update* de um *rating* no modelo de dados, o fato que explica este tempo é devido ao realizar uma atualização de um determinado *rating* no *Hbase* este primeiramente deve ser removido e então inserido.

Outra comparação realizada com a arquitetura distribuída do *Mahout* é, ao utilizar o *HBase* no modelo de dados do *IRF* (componente *DataModel*), implementa-se menos tarefas de *MapReduces*. O método *ItemBasedSimilarity* implementado sobre o *IRF* possui cinco tarefas de *MapReduces*, enquanto o método de recomendação provido pelo *Mahout* possui 9 tarefas de *MapReduces* para obter o mesmo resultado.

Capítulo 6

Conclusões

Para a realização deste trabalho, foi proposta a implementação de um módulo que permite calcular recomendações de forma distribuída.

Com base nas justificativas deste trabalho, pode-se concluir que a maioria das aplicações atuais devem ser escaláveis devido ao aumento no volume de dados. Pode-se observar também o crescimento exponencial de dados disponíveis na *Web*. Por causa disto, em alguns casos, ser ou não escalável acaba não se tornando uma opção, mas sim uma necessidade, por questões de limitação física (limitações de *hardware*) ou de processamento (tempo de resposta). Por fim, pode-se concluir que as vantagens de se desenvolver sistemas escaláveis e que utilizam *hardwares* de baixo custo possuem natureza tanto técnica quanto financeira.

Com o desenvolvimento deste trabalho pode-se concluir também sobre as vantagens de desenvolver e utilizar um *framework*, uma vez que um *framework* tem por objetivo facilitar que novas aplicações sejam desenvolvidas sobre uma determinada família de problemas na qual o *framework* se aplica.

Com base no desenvolvimento de um módulo que seja capaz de oferecer escalabilidade às aplicações de recomendação desenvolvidas sobre o *IRF*, diversas vertentes de trabalhos futuros foram identificadas. Um destes trabalhos identificados é a implementação de aplicações distribuídas que utilizam de outras abordagens de recomendação, tais como, baseada em conteúdo, dados de uso ou abordagem híbrida.

Um trabalho futuro que pode ser executado de forma paralela a estes já citados é a implementação de novos métodos de recomendação distribuídos para aplicação de filtragem colaborativa. Com um número maior de métodos de recomendação por filtragem colaborativa, um outro trabalho que poderá ser realizado são testes de acurácia. E com os resultados obtidos a partir deste teste escolher o método que apresente melhores

resultados de acurácia.

Outra vertente de trabalho futuro identificado é a implementação de um meta-recomendador. Um meta-recomendador deve ser capaz de escolher o melhor método de recomendação que se aplica a determinados tipos de bases de dados e usuários.

Referências Bibliográficas

[1] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Eng.*, 17:734–749, June 2005.

[2] Chris Anderson. The long tail. Hyperion, 2006.

[3] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] Marko Balabanovic' and Yoav Shoham. Content-based, collaborative recommendation. *Communications of the ACM*, 40:66–72, 1997.

[5] Ranieri Baraglia, Carlos Castillo, Debora Donato, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Aging effects on query flow graphs for query suggestion. In *CIKM'09*, pages 1947–1950, 2009.

[6] Chumki Basu, Haym Hirsh, and William Cohen. Recommendation as classification: Using social and content-based information in recommendation. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 714–720. AAAI Press, 1998.

[7] John S. Breese, David Heckerman, and Carl Myers Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In Gregory F. Cooper and Serafín Moral, editors, *UAI*, pages 43–52. Morgan Kaufmann, 1998.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Conference on Usenix Symposium on Operating Systems Design and Implementation - Volume 7*, pages 205–218, 2006.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe, 2008.

[11] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[12] Marcus Eduardo Markiewicz and Carlos J. P. de Lucena. Object oriented framework development. *Crossroads*, 7:3–9, July 2001.

[13] Felipe Martins Melo and Álvaro R. Pereira Jr. Idealize recommendation framework - an open-source framework for general-purpose recommender systems. In *14th international ACM Sigsoft symposium on Component based software engineering*, pages 67–72, June 2011.

[14] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action (MEAP)*. Manning, 2010.

[15] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The adaptive web: methods and strategies of web personalization*, volume 4321, pages 291–324. Springer, 2007.

[16] Pascal Soucy and Guy Mineau. A simple KNN algorithm for text categorization. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 647–648, 2001.

[17] Kazunari Sugiyama, Kenji Hatano, and Masatoshi Yoshikawa. Refinement of tf-idf schemes for web pages using their hyperlinked neighboring pages, 2003.

[18] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[19] Jun Wang, Arjen P. De Vries, and Marcel J. T. Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 501–508. ACM Press, 2006.