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 ajudando 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 qualquer estratégia 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]. Neste trabalho descrevo a implementação de uma aplicação de recomendação baseada em conteúdo que possui uma arquitetura denominada pseudo-distribuída. A aplicação de recomendação baseada em conteúdo foi implementada como parte deste trabalho para que fosse com-

preendido de forma mais detalhada o funcionamento e modo de implementação dos *hot spots* do *Idealize Recommendation Framework*. Posteriormente no capítulo 4 descrevo sobre os componentes e classes comuns para as aplicações de recomendações distribuídas. Em seguida, apresento detalhes sobre a implementação de uma aplicação totalmente distribuída baseada em filtragem colaborativa.

Capítulo 2

Justificativa

De acordo com Chris Anderson, editor chefe da revista *Wired*, *"We are leaving the age of information and entering the age of recommendation"* [2]. Com a mudança no modo de utilizarmos e obtermos informações, percebe-se o significado da realização de um busca que leva em consideração informações a respeito de quem a está realizando. Esta personalização exige processamentos computacionalmente caros e que dependem de grandes quantidades de dados para apresentarem uma boa acurácia. Assim como a importância de um sistema de recomendação por levar informações personalizadas aos seus utilizadores, 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. A função principal de um sistema escalável é distribuir os componentes e serviços de forma a aumentar o desempenho. No caso de sistemas de recomendação diminuir o tempo de processamento das recomendações a medida que aumentamos o número de máquinas. 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 deve possibilitar a adoção de

[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/

diferentes métodos de recomendação, o que justifica o desenvolvimento de um *framework* distribuído e escalável para sistemas de recomendação.

Capítulo 3

Objetivos

Os objetivos deste trabalho foram:

- A criação de uma aplicação de recomendação baseada em conteúdo [4] utilizando a arquitetura do *IRF* pseudo-distribuída.
- A derivação de classes comuns que facilitem a criação de aplicações de recomendação distribuídas sobre o *IRF*.
- A implementação de uma aplicação de recomendação distribuída baseada em filtragem coltaborativa [15]. Esta aplicação foi implementada utilizando o módulo de distribuição do *IRF* e será melhor detalhada na seção 4.5.2. Desta forma ao criar novas aplicações distribuídas poderá ser seguido o exemplo da aplicação implementada.
- Realização dos experimentos variando o número de máquinas em um *cluster* e a comparação entre os resultados obtidos.

Capítulo 4

Desenvolvimento

O *Idealize Recommendation Framework*[1] foi implementado utilizando a linguagem de programação Java. Ao realizar pesquisas relacionadas à área de recomendação foram encontrados apenas dois *frameworks* que possuem código aberto sendo eles, *Apache Taste*[2] e *MyMedia*[3]. O *IRF* foi construído a partir do *Framework 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.

Foram utilizados dois outros *frameworks* para implementar a arquitetura distribuída, *Apache Hadoop*[5] e *HBase*[6].

Um *framework* provê uma solução para uma família de problemas semelhantes [12], usando um conjunto em geral de classes abstratas e interfaces que mostra como decompor a família destes problemas, e como 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 dentro do mesmo domínio mais rapidamente, 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 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

[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/

ao qual o *framework* se aplica.

A figura 4.1 apresenta a estrutura em alto nível dos *frameworks* que compõem o *IRF*.

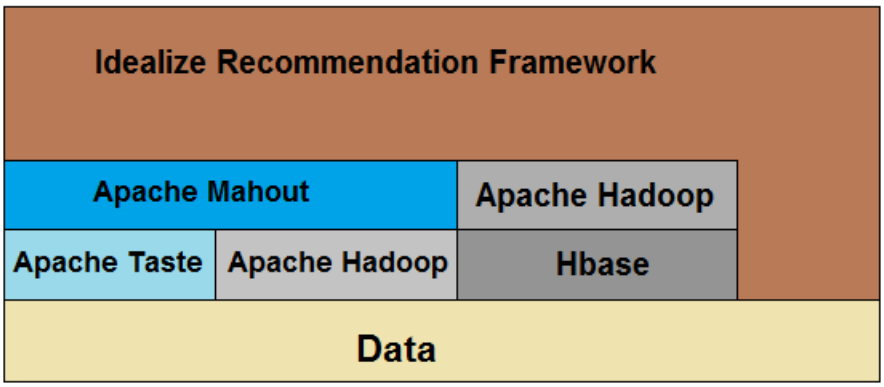

Figura 4.1: Arquitetura alto nível dos *frameworks* utilizados

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 na recomendação, trabalhando de forma independente. Os setores criados são setor de *Batch*, setor de *Cache* e setor de *Input*, a serem apresentados na seção 4.1.

- Processamento e armazenamento distribuído, quando comparado ao *MyMedia* e utilização do *framework HBase* para armazenamento distribuído, quando comparado ao *Apache Taste*.

4.1 Descrição dos setores do modelo de produção

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 setor de *Input* responsável por fazer a manipulação da base de dados [13]. A figura 4.3 ilustra a comunicação entre cada um dos setores e a comunicação do setor de *Batch* com o *cluster*[7] de computadores. Além destes setores podemos ver também na figura 4.3 a comunicação com a base de dados. A base de dados representada na figura pode ser um banco de dados, arquivos de texto, um índice do *Apache Lucene*[8] 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 distribuída.

Antes de descrever os setores e os componentes presentes em cada um dos setores, pode-se observar na figura 4.2 como segue o fluxo de uma requisição de recomendação.

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

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

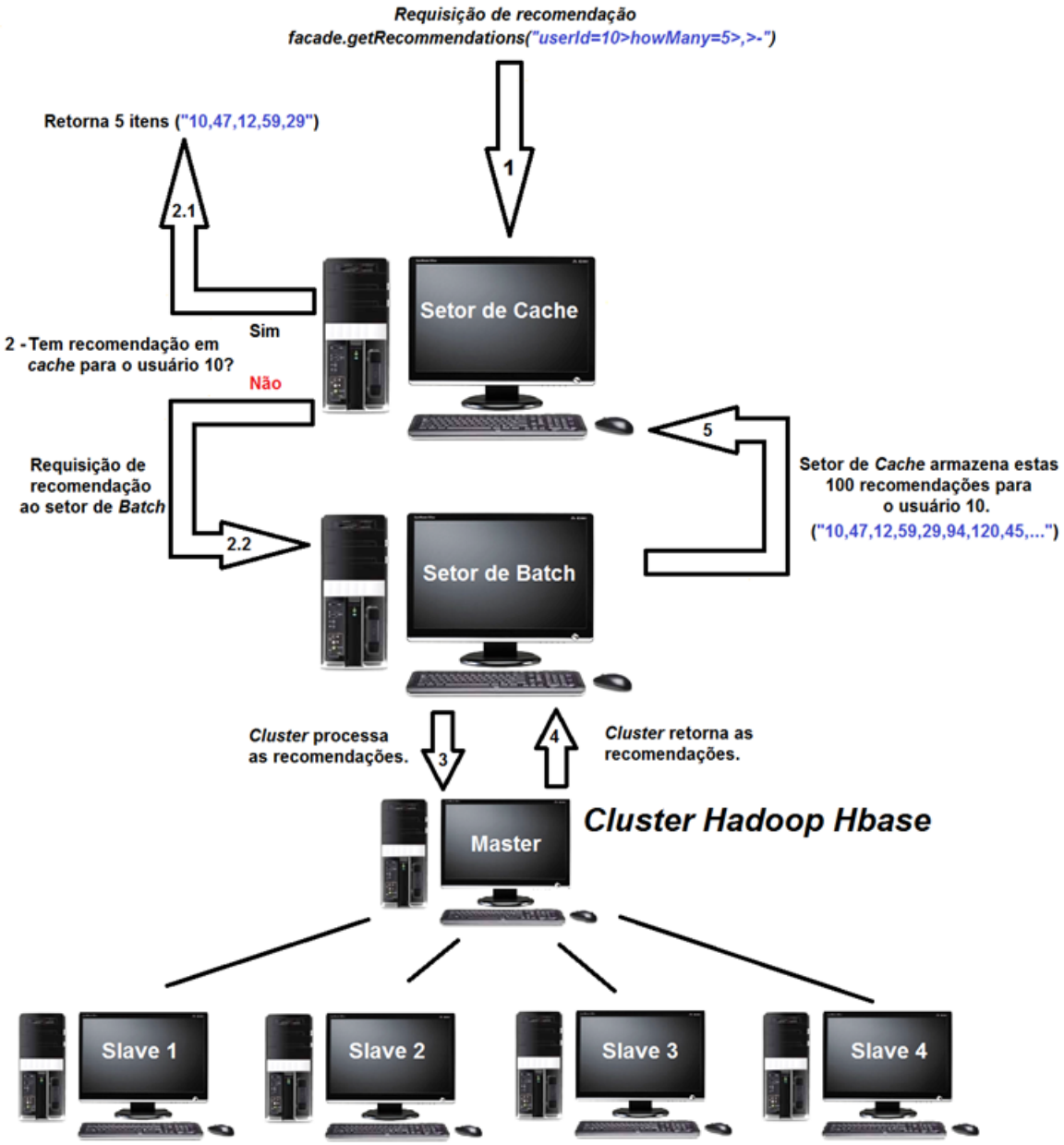

Figura 4.2: Fluxo de requisição de recomendação

4.1.1 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

4.1.2 Setor de *Batch*

Quando não se utiliza a arquitetura distribuída este setor é responsável pelo o processamento das recomendações e dos *feedbacks* recebidos dos usuários. Estas informações são recebidas pelo setor de *Cache* e repassadas para este setor de *Batch*. O setor de *Batch* registra alguns de seus componentes na rede de maneira a torná-los remotamente acessíveis utilizando *RMI*[9]. A comunicação representada na figura 4.3 entre os setores de *Batch* e *Cache* justifica-se o porque as requisições de recomendações realizadas pelo setor de *Cache* ao setor de *Batch*. 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 exemplo, quando um 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*, ou seja, o sistema atua de maneira inteiramente distribuída.

4.1.3 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.

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

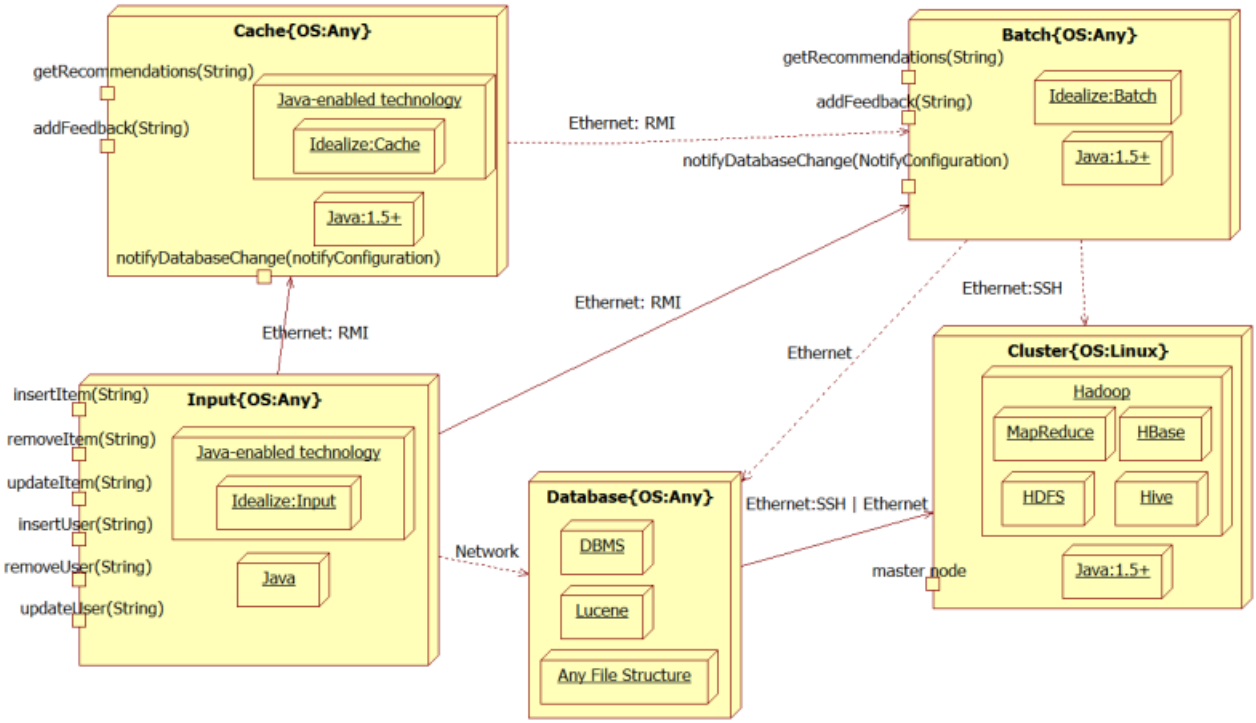

Figura 4.3: Arquitetura utilizada no modelo de produção

4.2 Recursos utilizados para implementação da arquitetura distribuída

Levando em consideração que muitos casos de implementações não distribuídas diferem de implementações distribuídas para o mesmo algoritmo, ainda assim muitos problemas podem ser estruturados para trabalhar de forma a usar o paradigma de programação *MapReduce*[10], como por exemplo, os métodos de recomendação.

4.2.1 Pesando recursos de uma arquitetura distribuída e 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 alto, além do que, o volume de dados

[10] http://hadoop.apache.org/mapreduce/

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 selecionado em uma máquina virtual do Java. 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. Problema este que só pode ser resolvido com uma arquitetura distribuída. Mas ainda assim deve-se pensar em vantagens e desvantagens de um sistema 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 todos 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[11] [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.

4.2.2 Introduzindo o *Hadoop e MapReduce*

Para adicionar um módulo com capacidade de processamento de recomendações de forma distribuída ao *IRF*, foram utilizados os *frameworks Apache Hadoop*[12] e *HBase*[13]. 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

[11] Com maior importância para a replicação de dados
[12] http://hadoop.apache.org/
[13] http://hbase.apache.org/

um *cluster* para que seja confiável por possuírem replicação e possa realizar cálculos de forma rápida a partir da localidade espacial[14].

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[15]. 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 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.

4.2.2.1 Estrutura de execução do *HBase*

HBase é o repositório de dados utilizado pelo *Hadoop*. Este repositório de dados deve ser usado quando da necessidade de acessos aleatórios de leitura ou escrita em tempo real referente a uma grande quantidade de dados. O objetivo do *HBase* é suportar tabelas muito grandes com milhões de linhas sobre *hardwares* de baixo custo[16]. O *Hbase* é *open-source*, trabalha de forma distribuída e possui um modelo de armazenamento orientado a colunas modelado como o *Bigtable*[8] utilizado pelo *Google File System*[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* prove operações sobre *HBase*. O *MapReduce* quando utiliza-se o *HBase* é chamado de *TableMap* e *TableReduce* e possui 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 analoga a um registro de um banco de dados relacional. Porém os valores de entrada passam a ser os valores que estão em tabelas, tabelas estas 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

[14] http://hadoop.apache.org/hdfs/
[15] http://hadoop.apache.org/common/docs/current/mapred_tutorial.html
[16] http://hbase.apache.org/

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 função *TableReduce* opera sobre a entrada recebida do *TableMap* e faz os devidos cálculos sobre estes dados. A partir destes cálculos 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 um objeto de gravação em uma *HTable*.

4.3 Componentes do IRF

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, são em geral classes abstratas ou interfaces.

No pacote *distributed_ hotspots* estão os componentes que são utilizados somente em aplicações distribuídas. 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 *Cluster Hadoop HBase*.

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.

- Pacote *distributed_ hotspots*

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

A figura 4.4 apresenta a arquitetura alto nível dos componentes utilizados no setor de *Batch* e setor de *Cache*.

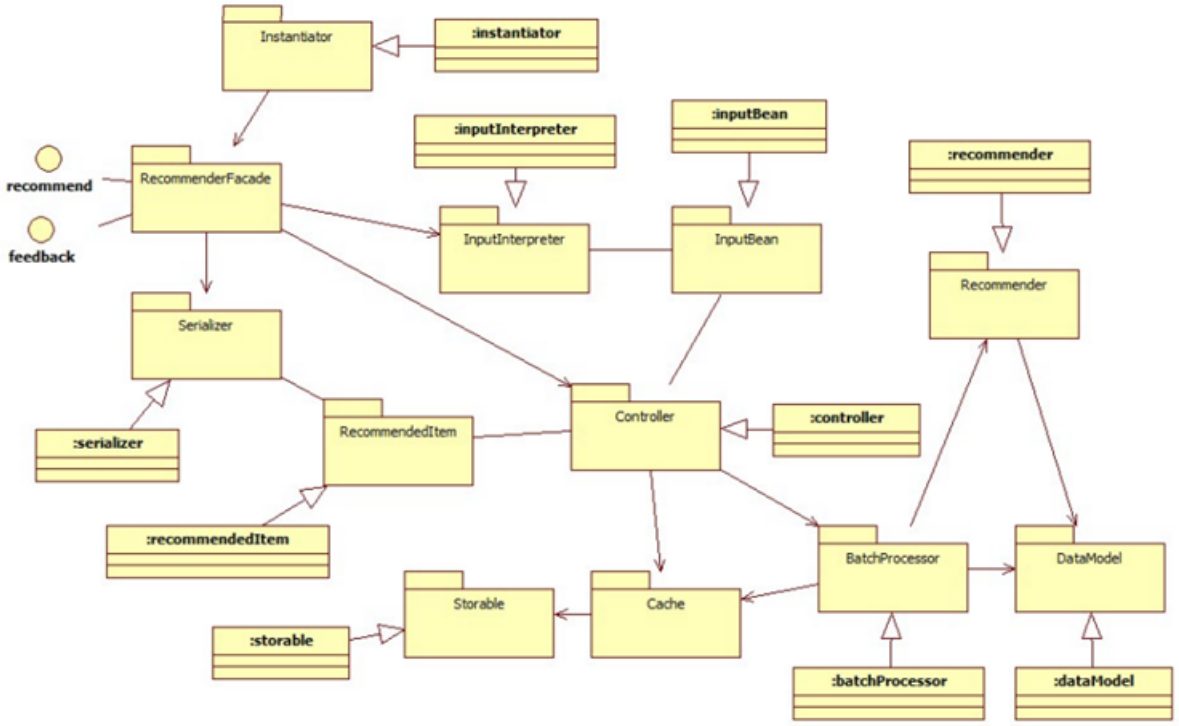

Figura 4.4: Arquitetura alto nível dos componentes utilizados no setor de *Batch* e *Cache*

A figura 4.5 apresenta a arquitetura alto nível dos componentes utilizados no setor de *Input*.

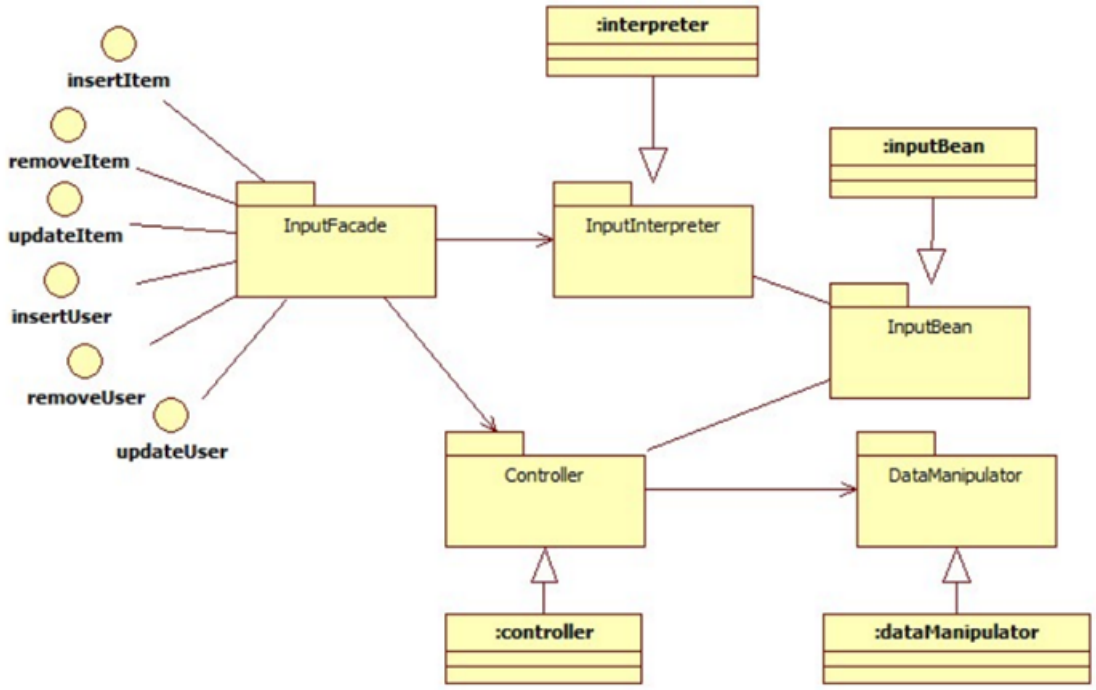

Figura 4.5: Arquitetura alto nível dos componentes utilizados no setor de *Input*

A figura 4.6 apresenta a arquitetura alto nível dos componentes utilizados na aplicação que é executada no *cluster*.

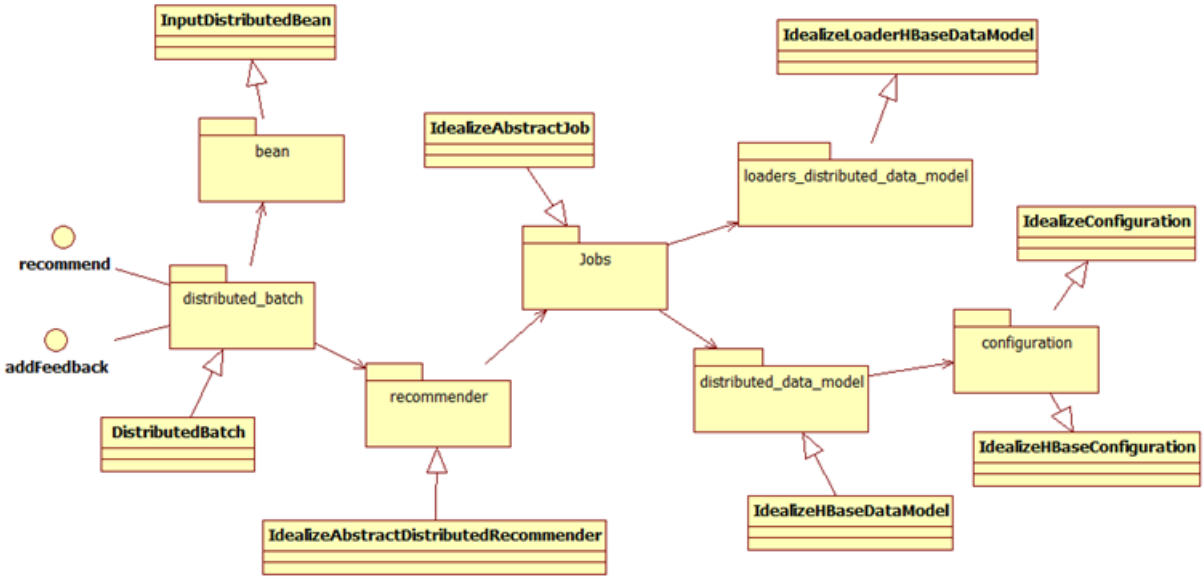

Figura 4.6: Arquitetura alto nível da aplicação executada no *cluster*

4.3.1 Descrição dos componentes do pacote *base*

Afim de reduzir a complexidade e detalhamento na explicação do funcionamento de cada um dos componentes do *IRF*, evito ao máximo a repetição de informações. Mas para compreessão de forma 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 utilizado, separador da *string* enviada à fachada, modelo de dados que será carregado, entre outras propriedades que definem quais classes serão instanciadas em tempo de execução. O desenvolvedor deverá apenas configurar o arquivo 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 idéia 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 *RemoteIdealize-RecommenderFacade* e define o fluxo de produção de recomendações e fluxo relacionado 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 a 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 apenas para melhorar a semântica de manipulação de exceções, mantendo o encapsulamento, de acordo com os padrões de manipulação de exceções existentes [9].

4.3.2 Descrição dos componetes do pacote *hot_ spots*

Apesar de uma série de componentes neste pacote serem utilizados em uma aplicação de recomendação distribuída, existem especificidades que serão descritas na explicação do pacote *distributed_ hot_ spots*.

- *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 *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, IdealizeCBAbstractRecommender, 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:* Este componente controla o fluxo da aplicação de recomendação. Este componente recebe um *Input-Bean* vindo do *InputIntepreter* e devolve uma lista de *RecommendedItem*[17], 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 política de armazenamento de dados em *cache*, ou sequência de operações a serem realizadas é definida pelo desenvolvedor e cada aplicação possui o seu controle de fluxo específico.

- *BatchProcessor:* Responsável por realizar o processamento em lote das recomendações e dos *feedbacks*. Este componente também possui o dever de realizar a comunicação com a parte distribuída quando optar pela utilização da arquitetura distribuída. 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...", porém este formato pode ser modificado de acordo com as necessidades do cliente.

- *DataManipulator:* Este componente está presente apenas no setor de *Input*, e tem o papel de fazer 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

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

documentos do *Lucene*, e assim de acordo com a estrutura de armazenamento de dados utilizada pela aplicação sendo desenvolvida.

4.3.3 Descrição dos componentes do pacote *distributed_hot_spots*

A maioria dos componentes descritos acima são utilizados também para a implementação de uma aplicação distribuída, porém ao realizar este tipo de implementação ainda deve-se implementar componentes específicos para este tipo de arquitetura.

- *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*. Toda aplicação de recomendação distribuída deverá ter pelo menos uma implementação desta classe ou implementação de uma classe que seja subclasse desta.

- *IdealizeConfiguration:* Este componente também funciona como uma interface de marcação assim como vários outros aqui descritos. Em uma implementação deste componente é permitido ao desenvolvedor definir o tipo de configuração que será utilizada na recomendação distribuída. Desta forma fica a cargo do desenvolvedor definir quais serão os parâmetros utilizados para a manipulação dos dados utilizados pelo *dataModel* e pelas tarefas de *MapReduce*.

- *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 especifico para recomendadores distribuídos, uma vez que são mantidos nela os atributos e métodos voltados a este tipo de arquitetura. Todo método de recomendação distribuído deve ter pelo menos um *IdealizeAbstractJob* para executar, e portanto este componente possui um objeto do tipo *IdealizeAbstractJob* que deve ser instanciado em seu construtor.

- *IdealizeHBaseDataModel:* Componente definido para os recomendadores que utilizam o *HBase* como modelo de dados. 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 componente 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. Desta forma, se houver necessidade de novas mudanças, estas não causarão grandes impactos.

- *IdealizeWritable:* Todos os dados utilizados em aplicações distribuídas devem ser passíveis de serem armazenados em disco, pois ao manipular grandes volumes de dados, parte destes dados são transformados em *bytes* e armazenados em disco de acordo com a política do *Hadoop*. As novas classes que sejam *Writable* utilizados no *IRF* devem ser subclasses desta e implementar seus métodos.

- *LoaderToHBaseDataModel:* Aqui é definido um componente que possui métodos que permitem o carregamento de dados para o *HBase*. Assim, toda política de carregamento de dados para o *HBase* deverá estar em uma classe que seja um componente deste tipo. Desta forma, todas as subclasses que façam carregamento de dados para o *HBase* implementarão os métodos de comunicação com o *HBase* definidos nesta classe.

4.4 Implementação da aplicação baseada em conteúdo

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 realizadas 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].

A presente aplicação foi realizada utilizando a arquitetura pseudo-distribuída e tem a finalidade de detalhar como é o funcionamento da implementação de cada um dos componentes.

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.

Na aplicação baseada em conteúdo desenvolvida sobre o *IRF* foi utilizado o *Apache Lucene*. O *Lucene* é uma ferramenta para indexação e recuperação de informações textuais. Na fase de indexação os dados originais são processados gerando uma estrutura 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 4.1 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 4.1: Exemplo de 3 documentos gravados no *Lucene*.

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

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 são utilizadas classes providas pelo *framework Mahout*. 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 implementadas no *Mahout* são:

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

4.4.2 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 para aplicações que possuem arquitetura distribuída, pois em geral têm a mesma finalidade.

4.4.2.1 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 caminhos para os componentes que serão 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 dividir 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 pedir as recomendações e seja retornada 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 temos um item a ser comparado com os 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: "ItemId<sep>ItemId<sep>ItemId...".

- *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 *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:* Este componente é 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 assim 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, 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 id 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 instancia 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.

4.4.2.2 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)*. 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 na seção do setor de *Batch* 4.4.2.1.

- *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 *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. Parte desta estrutura é mantida em memória principal e pode ser acessada em complexidade de tempo *O(1)*. Para manter as

principais recomendações em memória principal utiliza-se heurísticas que mantém as recomendações dos itens mais relevantes em memória principal.

- *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, este pedido é passado 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 do mundo externo.

- *BatchProcessor:* Este componente requisita recomendações para a fachada da máquina que contém o setor de *Batch* instanciado. 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*.

4.4.2.3 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.

- *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 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 um determinado 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*. A informação de quantos documentos foi modificada é armazenada.

4.5 Implementação da aplicação distribuída baseada em filtragem colaborativa

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

A abordagem de filtragem colaborativa tem a sua origem na mineração de dados [3]. Como já mencionado é a recomendação que leva em consideração a ação de múltiplos usuários [15, 19, 7]. Alguns componentes criados para o módulo distribuído do *IRF* foram baseados em implementação de classes do *Mahout*. A parte distribuída do *Mahout* utiliza 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 sofrer alterações. Em sistemas de recomendação a todo tempo os usuários enviam *ratings* aos novos itens, desta forma se faz necessário a modificação da base de dados. Por este motivo a escolha do *HBase* ao desenvolver o módulo distribuído para o *IRF*, que é responsável por fazer o gerenciamento dos índices na base de dados.

4.5.1 Descrição da base de dados

A base de dados utilizada para os testes foi a base da empresa *Netflix*. *Netflix* é uma empresa americana que permite aos usuários assistir transmissões tanto de programas de TV quanto filmes pela internet[18].

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 seria dado à equipe que desenvolvesse 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 96721844 (aproximadamente 97 milhões) de *ratings*, 480189 usuários e 17770 filmes. Os dados de entrada encontram-se no seguinte formato *<id usuário>:<id do item><valor de rating>*. Esta quantidade de *ratings* é um tamanho razoável para fins dos experimentos realizados.

4.5.2 Descrição dos componentes distribuídos implementados

Em geral, vários componentes descritos na seção do desenvolvimento de uma aplicação baseada em conteúdo possuem o mesmo funcionamento ou a mesma idéia de outras aplicações desenvolvidas sobre o *IRF*, inclusive sobre aplicações de recomendações distribuídas. Assim, apenas serão descritos componentes utilizados para execução da aplicação sobre o *cluster*.

[18] www.netflix.com

- *IdealizeAbstractJob:* Os *Jobs* implementados definem a sequência de *Maps* e *Reduces* a serem executados. Foram implementados *Jobs* para os três métodos de recomendação mencionados, 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 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 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 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 *CF* criei um modelo de dados com estas duas tabelas, 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 *hot spot* 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 comum para os métodos da abordagem de recomendação por filtrabem 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 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*. De uma forma geral, todas estas classes são utilizadas pelos *Maps* e *Reduces* dos métodos de recomendação por *CF*. 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[19] para o *HBase*. Quando estes valores são gravados no *HBase* uma vez, todo novo processo de inserção, atualização ou remoção ocorre diretamente sobre o próprio *HBase*.

4.5.3 O método de recomendação distribuído *MostPopularOverRating*

Foram implementados três métodos de recomendação distribuídos: *MostPopularJob*, *MostPopularOverRatingJob* e *ItemBaseSimilarityJob*. Aqui descrevo o *job MostPopularOverRatingJob*, por ser simples e com a explicação deste método é possível passar o modelo de construção 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. Portanto nem sempre há um padrão de números de *Jobs* a serem implementados, 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 *ratings* existem para os itens. Neste método só são considerados os *ratings* que estão acima de um determinado valor passado como parâmetro.

[19] Sistema de Gerenciamento de Banco de Dados

Listing 4.1: MostPopularOverRatingMap

```
@Override
public void map(ImmutableBytesWritable row, Result values,
                Context context) throws IOException {

        LongWritable itemId = new LongWritable();
        List<KeyValue> prefsItemIds = values.list();
        int size = prefsItemIds.size();
        List<KeyValue> listItems = values.list();

        try {
        // runs through all values of a given registry
                for (int i = 0; i < size; i++) {
                        itemId.set(Bytes.toLong(
                                        listItems.get(i).getQualifier()));
                        float value = Bytes.toFloat(
                                        listItems.get(i).getValue());

                        if (value >= this.valueRating)
                        //record this value as input to Reduce
                                context.write(itemId, one);
                }
        } catch (InterruptedException e) {
                throw new IOException(this.canonicalName
                        + ".Map: Could not possible complete Map:"
                        + e.toString());
        }
}
```

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*.

No código 4.1 podemos ver o *Map* atuando sobre os registros da tabela e armazenando para cada item o valor 1. Consequentemente todos os valores são agrupados para todo item que possui o mesmo identificador. O próprio *Hadoop* é responsável por fazer este agrupamento.

Listing 4.2: MostPopularOverRatingReduce

```
/**
 * This is the class reduce used to store how many times
 * the items were ranked.
 *
 * @author Alex Amorim Dutra
 */
public static class DistributedMostPopularOverRatingReducer
        extends TableReducer<LongWritable, IntWritable,
        ImmutableBytesWritable> {

        @Override
        public void reduce(LongWritable key,
                  Iterable<IntWritable> values, Context context)
                  throws IOException, InterruptedException {

                Long id = key.get();
                ImmutableBytesWritable itemID =
                          new ImmutableBytesWritable(Bytes.toBytes(0));

                int sum = 0;

                // make the counting
                for (IntWritable itr : values)
                        sum++;

                long row = 0;
                Put put = new Put(Bytes.toBytes(row));
                put.add(Bytes.toBytes("timesValueItemId"),
                          Bytes.toBytes(sum), Bytes.toBytes(id));
                context.write(itemID, put);
        }
}
```

No código 4.2, a entrada é (*K2, V2*), onde *K2* é o identificador de um item e *V2* é um

Iterator de valores[20]. Ao receber esta entrada no código do *TableReduce*, os valores vão sendo somados de um em um, indicando quantos *ratings* o item recebeu. Por fim, com estes valores é criado um objeto do tipo *Put* e armazenado na tabela de saída do método. Este é um método que possui um funcionamento simples, portanto de fácil compreensão.

[20] http://wiki.apache.org/hadoop/MapReduce

Capítulo 5

Experimentos

Para a realização dos experimentos foi utilizado a base de dados da *Netflix*. Esta base contém um total de 96721844 *ratings*, 480189 usuários e 17770 filmes assim como está descrito na seção 4.5.1. Os testes foram executados um cluster contendo 9, 7 e 5 máquinas.

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

Ecplise Galileo versão 3.5

Os computadores utilizados no cluster possuem as seguintes configurações de hardware:

*P*rocessador AMD Phenom II X3 B75 - 3 Cores - 800 MHz

Memória *RAM 2GB*

O método utilizado para realização dos experimentos foi o *ItemBasedSimilarity*. A escolha deste método deve-se ao fato da possibilidade de realizar a comparação com o método implementado no *Mahout*, desta forma foi possível realizar a comparação com um método que utiliza diretamente *HDFS* implemetado sobre o *Mahout*.

O método de recomendação *ItemBasedSimilartiy* possui três atributos a serem passados, sendo eles: tipo de similaridade entre os itens, quantidade máxima de *ratings* por usuário e quantidade máxima de itens que co-ocorrem.

As classes de similaridade entre itens implementadas no *Mahout* sao: *SIMILARITY_ COOCCURRENCE, SIMILARITY_ EUCLIDEAN_ DISTANCE, SIMILARITY_ LOGLIKELIHOOD, SIMILARITY_ PEARSON_ CORRELATION, SIMILARITY_ TANIMOTO_ COEFFICIENT*

Os itens são co-ocorrentes quando são *rankeados* pelo mesmo usuário.

Para a similaridade de itens foi utilizada a classe *SIMILARITY_PEARSON_CORRELATION*. O atributo referente a quantidade máxima de *ratings* de um usuário foi configurado com o valor 100. O número máximo de co-ocorrência foi definido com o valor 100. O resultado obtido é o tempo de execução para a realização do cálculo de recomendação para todos os usuários pertencentes a base.

Foram realizadas três instâncias de teste para a mesma configuração e em seguida calculado o valor médio.

- *Cluster* contendo 9 máquinas.

  Resultados obtidos em um *cluster* com 9 máquinas executando o método implementado sobre o *IRF*.

Teste 1	**Teste 2**	**Teste 3**
273 Minutos	254 Minutos	263 Minutos

Tabela 5.1: Resultado sobre *IRF* com 9 Máquinas

  Média para os resultados da tabela 5.1 263,33 minutos.

  Resultados obtidos em um *cluster* com 9 máquinas executando o método do *Mahout*.

Teste 1	**Teste 2**	**Teste 3**
192 Minutos	213 Minutos	217 Minutos

Tabela 5.2: Resultado sobre o *Mahout* com 9 Máquinas

  Média para os resultados da tabela 5.2 207,33 minutos.

- *Cluster* contendo 7 máquinas.

  Resultados obtidos em um *cluster* com 7 máquinas executando o método implementado sobre o *IRF*.

  Média para os resultados da tabela 5.3 349,7 minutos.

  Resultados obtidos em um *cluster* com 7 máquinas executando o método do *Mahout*.

  Média para os resultados da tabela 5.4 281 minutos.

Teste 1	**Teste 2**	**Teste 3**
347 Minutos	365 Minutos	337 Minutos

Tabela 5.3: Resultado sobre *IRF* com 7 Máquinas

Teste 1	**Teste 2**	**Teste 3**
292 Minutos	279 Minutos	272 Minutos

Tabela 5.4: Resultado sobre o *Mahout* com 7 Máquinas

- *Cluster* contendo 5 máquinas.

  Resultados obtidos em um *cluster* com 5 máquinas executando o método implementado sobre o *IRF*.

Teste 1	**Teste 2**	**Teste 3**
412 Minutos	394 Minutos	401 Minutos

Tabela 5.5: Resultado sobre *IRF* com 5 Máquinas

Média para os resultados da tabela 5.5 402,33 minutos.

Resultados obtidos em um *cluster* com 5 máquinas executando o método do *Mahout*.

Teste 1	**Teste 2**	**Teste 3**
292 Minutos	319 Minutos	321 Minutos

Tabela 5.6: Resultado sobre o *Mahout* com 7 Máquinas

Média para os resultados da tabela 5.6 310,7 minutos.

A tabela 5.7 contém os resultados com os valores médios para o *cluster* com 5, 7 e 9 máquinas.

O gráfico mostra os resultados obtidos para o valor médio dos testes realizados.

De acordo com o gráfico 5.1 podemos ver a melhoria no tempo de execução a medida que executa-se os experimentos em um *cluster* com uma quantidade maior de máquinas.

Pode-se observar que o método do *Mahout* apresentou melhores resultados. Estes melhores resultados são obtidos devido ao fato do *HDFS* não possuir políticas de in-

	5 Máquinas	**7 Máquinas**	**9 Máquinas**
IRF	402,33 Minutos	349,7 Minutos	263,33 Minutos
Mahout	310,7 Minutos	281 Minutos	207,33 Minutos

Tabela 5.7: Valores médios obtidos

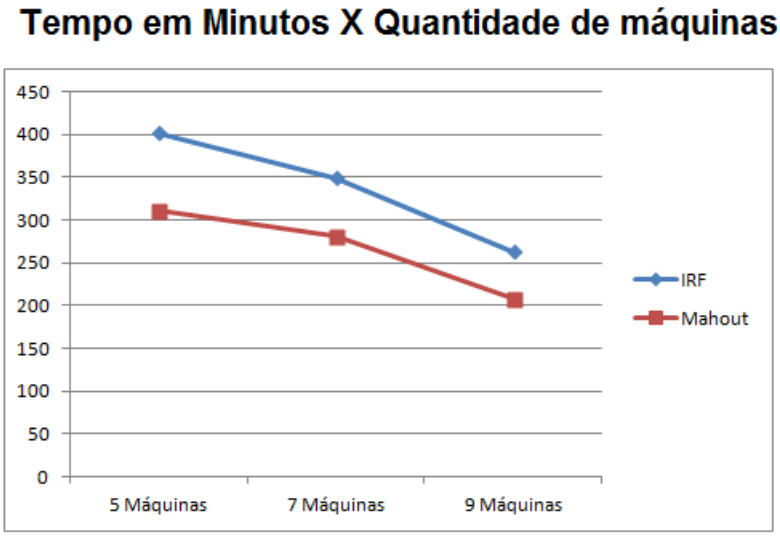

Figura 5.1: Tempo médio x Quantidade de máquinas

dexação como o *HBase*. O *HBase* em alguns casos poderá tornar uma aplicação até 4 vezes mais lenta que outra aplicação que execute diretamente sobre o *HDFS* [1]. Porém o *HBase* apresenta a vantagem de atualização de dados em tempo real e permite tanto a escrita quanto a leitura destes dados de forma aleatória, o que se faz necessário a todo momento em sistemas de recomendação. Levando em consideração o tempo de resposta aos usuários, este é de forma instantânea, uma vez que o *IRF* possui o setor de *Cache* que armazena recomendações pré-calculadas [13]. Outra vantagem ao se utilizar o *IRF* deve-se ao fato da implementação de menos *MapReduces* devido ao seu modelo de dados sobre o *HBase*.

[1] http://hbase.apache.org/book/perf.hdfs.html

Capítulo 6

Conclusões

Para a realização deste trabalho, foi proposto a implementação de um módulo que permita realizar cálculos de recomendação de forma distribuída. Com base nas argumentações dos itens desenvolvidos, pode-se concluir que a maioria das aplicações hoje em dia devem ser escaláveis devido ao grande aumento no volume de dados. Pode-se observar também que a cada dia há um crescimento de informações na *Web*. Em alguns casos, ser ou não escalável acaba não se tornando uma opção, mas sim uma necessidade. Necessidade esta por questões de limitação física, como por exemplo limite de memória *RAM*. Pode-se concluir também que as vantagens de se desenvolver sistemas escaláveis e que utilizam *hardwares* de baixo custo, possuem natureza tanto técnica quanto financeira.

6.1 Trabalhos Futuros

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 identificados.

Um destes trabalhos é a implementação de uma aplicação distribuída que utiliza a abordagem de recomendação baseada em conteúdo, pelo fato de estar entre as abordagens de recomendação que garantem bons resultados de acurácia.

Outro trabalho identificado é a implementação de aplicações distribuídas que utilizam de outras abordagens de recomendação, tais como, abordagem híbrida ou dados de uso.

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.

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.