Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

ADICIONANDO ESCALABILIDADE AO *FRAMEWORK DE RECOMENDAÇÃO IDEALIZE*

Aluno: Alex Amorim Dutra
Matricula: 07.1.4149

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

Ouro Preto
3 de dezembro de 2010

Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

ADICIONANDO ESCALABILIDADE AO *FRAMEWORK DE RECOMENDAÇÃO IDEALIZE*

Relatório de atividades desenvolvidas apresentado ao curso de Bacharelado em Ciência da Computação, Universidade Federal de Ouro Preto, como requisito parcial para a conclusão da disciplina Monografia I (BCC390).

Aluno: Alex Amorim Dutra
Matricula: 07.1.4149

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

Ouro Preto
3 de dezembro de 2010

Resumo

Palavras-chave: Escalabilidade. *Hadoop*. *Framework*. Recomendação. Idealize. *Framework de Recomendação Idealize*.

Sistemas de recomendação na Web têm a cada dia deixado de ser uma novidade e têm se tornado uma necessidade para os usuários da Web, devido ao grande volume de dados disponíveis. Estes dados tendem a crescer cada vez mais, o que poderá ocasionar uma perda de tempo considerável ao realizar buscas manualmente para encontrar conteúdos relevantes na Web. Sistemas de recomendação têm a finalidade de levar conteúdo relevante a seus utilizadores. Para isto são utilizados algoritmos de aprendizagem de máquina e outras técnicas de recomendação. Para atuarem de maneira eficiente, 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. No trabalho referente às disciplinas Monografia I (BCC390) e Monografia II (BCC391) será tratada a escalabilidade do *Framework de Recomendação Idealize*. Para tornar o *Framework de Recomendação Idealize* escalável será feito a utilização do *framework Hadoop*, pelo fato de ser *open-source* e estar presente em grandes sistemas que utilizam computação escalável e distribuída.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Com o crescimento da produção de dados, principalmente na Web [6], temos ao alcance informações relevantes em diversas áreas. Algumas vezes quando estamos realizando buscas a respeito de determinado assunto, produto, ou qualquer outro item acabamos não encontrando o que desejamos, devido 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 *Framework de Recomendação Idealize* (IRF) foi desenvolvido para suportar qualquer estratégia de recomendação [8]. As aplicações de recomendação desenvolvidas sobre o IRF até o momento possuem as seguintes abordagens: baseada em conteúdo [5], filtragem colaborativa [8], dados de uso [5] e Híbrida [5]. Em resumo recomendações baseadas em conteúdo recomendam um item para um usuário com base na descrição dos itens mais similares ao item sendo acessado, ou recomendam itens que possuem características similares as definidas no perfil do usuário [3, 1]. Recomendações por filtragem colaborativa tem sua origem na mineração de dados [2] e constituem o processo de filtragem ou avaliação dos itens através de múltiplos usuários [1, 9, 12], 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 [4], por exemplo, a sequência de links clicados por um usuário quando navega em um site de compras. A abordagem híbrida é a combinação de outros tipos de recomendação, podendo ser um modelo único abordando outros tipos de recomendação ou adicionando características de um tipo de recomendação em outra. A recomendação híbrida é interessante, pois possibilita que as limitações de cada técnica sejam supridas por características das demais [1].

Neste trabalho foi desenvolvido uma aplicação de recomendação baseada em conteúdo utilizando o IRF [8], esta aplicação será detalhada na seção de desenvolvimento.

2 Justificativa

Além da importância de sistemas de recomendação por levar o que realmente interessa aos usuários, existem outras necessidades que justificam a criação de um *framework* e da realização de armazenamento e processamento distribuído.

Um *framework* é de grande importância, pois provê uma solução para uma família de problemas semelhantes, usando um conjunto em geral de classes abstratas e interfaces que mostra como decompor a família de problemas, e como objetos dessas classes colaboram para cumprir suas responsabilidades. O conjunto de classes 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* [8]. 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* [8], pois constituem as decisões de *design* já tomadas dentro do domínio ao qual o *framework* se aplica.

A importância da escalabilidade está relacionada ao volume de dados processados. Grandes empresas como *Facebook, Yahoo, Google, Twitter e Amazon* armazenam volumes de dados da ordem de Petabytes[1], de onde podem ser extraídas informações relevantes. Assim, a escalabilidade tem a função principal de distribuir os componentes e serviços de forma a aumentar o desempenho, diminuindo o tempo de processamento das recomendações.

[1] http://escalabilidade.com/2010/05/18/

3 Objetivos

3.1 Objetivo geral

- O objetivo ao final das disciplinas Monografia I (BCC390) e Monografia II (BCC391) é apresentar o *Framework de Recomendação Idealize* com componentes que facilitam a construção de aplicações de recomendação escaláveis. A figura 1 ilustra a distribuição física do sistema em seu modelo de produção, onde existem os setores de *Input*, *Batch* e *Cache*, o local de armazenamento de dados e a representação de um *cluster* responsável pelo armazenamento e processamento distribuído. Os setores de *Batch*, *Cache* e *Input* estão detalhados na seção de desenvolvimento.

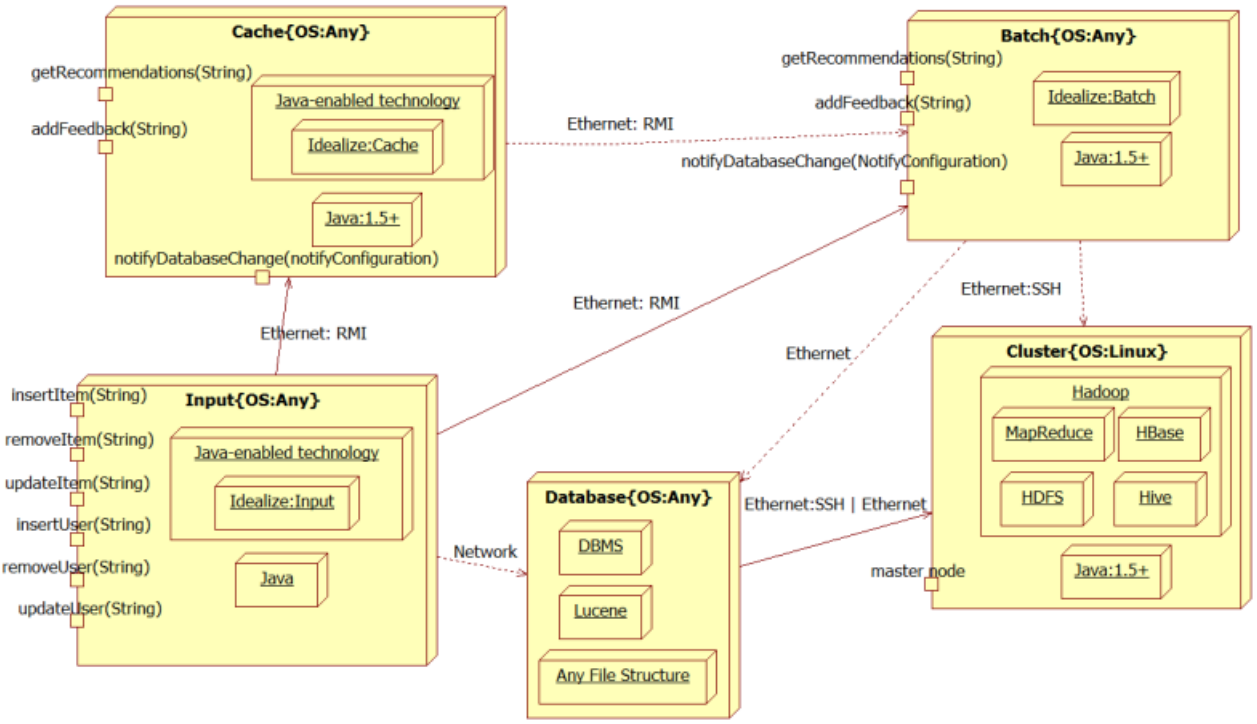

Figura 1: Arquitetura final

3.2 Objetivos específicos

- O primeiro objetivo é desenvolver uma aplicação de recomendação baseada em conteúdo utilizando o IRF. Esta aplicação não será escalável por meio de um *cluster*. Ao invés disso será pseudo-distribuída, com a implementação de cada um dos setores, *Batch*, *Cache* e *Input*, sendo cada setor executado em uma máquina.

- O segundo objetivo é implementar uma arquitetura escalável no *Framework de recomendação Idealize*.

  Esta tarefa será realizada nas disciplinas Monografia I (BCC390) e Monografia II (BCC391).

Nesta fase serão realizados estudos sobre escalabilidade. O objetivo principal nesta fase é utilizar o *Hadoop*[2] e construir um *cluster*[3] para realizar o processamento e armazenamento distribuídos.

- Desenvolver uma aplicação que utilize a arquitetura escalável.

  Por fim será desenvolvida uma aplicação distribuída que utilize alguma abordagem de recomendação, podendo ser de filtragem colaborativa, baseada em conteúdo, dados de uso ou híbrida. Esta nova aplicação servirá de modelo para outros desenvolvedores que desejarem utilizar a nova arquitetura escalável.

[2] http://hadoop.apache.org/

[3] Um cluster é formado por um conjunto de computadores, que realizam processamento em paralelo e distribuído.

4 Metodologia

A metodologia aqui descrita abrange o que será apresentado nas disciplinas Monografia I (BCC390) e Monografia II (BCC391). Este trabalho é de caráter exploratório. Deseja-se melhorar o tempo de processamento das recomendações. O trabalho será dividido em seis fases. A primeira consistirá de um estudo sistematizado do modo de produção baseado apenas em três máquinas, realizado com base em publicações, livros, artigos, revistas, e análise da aplicação de filtragem colaborativa desenvolvida sobre o IRF, que utiliza o modo composto por três máquinas.

A segunda fase será a implementação de uma aplicação de recomendação baseada em conteúdo para ser executada também de acordo com o modelo de produção baseada apenas em três máquinas. A terceira fase será a realização de testes e análise dos resultados encontrados para a aplicação de recomendação baseada em conteúdo, desenvolvida na segunda fase.

A quarta fase será o estudo aprofundado do *framework Hadoop*, feito com base em aplicações, publicações, livros, artigos. A quinta fase será a implementação de uma arquitetura que utilize o *Hadoop* para tornar IRF escalável e distribuído.

A sexta fase será a realização de novos testes envolvendo a escalabilidade e análise comparativa com os resultados obtidos na arquitetura que dispõe somente de três máquinas.

5 Desenvolvimento

O IRF foi implementado utilizando a linguagem de programação Java. Ao realizar pesquisas foram encontrados *frameworks* relacionados que possuem código aberto tais como, *Apache Taste*[4] e *MyMedia*[5]. O IRF foi construído a partir do *Framework Mahout*[6]. Foram reutilizados componentes de recomendação (*Taste*) e algoritmos de aprendizagem de máquina. O *Apache Taste* oferece componentes para criação de modelos de dados e alguns algoritmos de recomendação que utilizam filtragem colaborativa. Um Modelo de dados (*DataModel*) é a forma como os dados ficam acessíveis aos métodos de recomendação. O *Mahout* está construído sobre o *framework Apache Hadoop*[7]. A Figura 2 apresenta a estrutura em alto nível dos *frameworks* que compõem o IRF e a camada de dados manipulados.

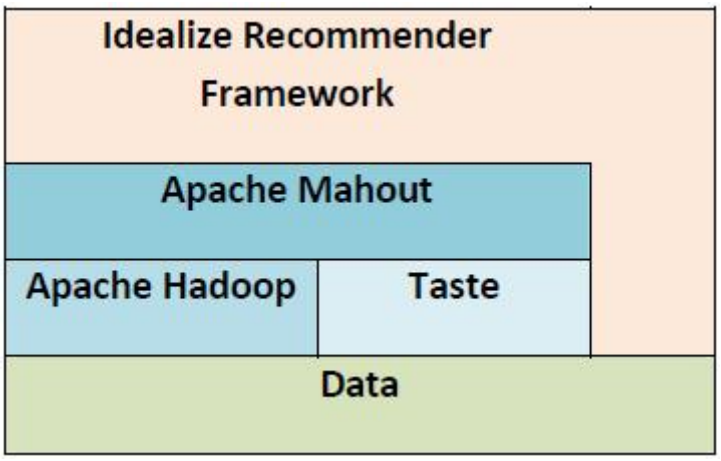

Figura 2: Arquitetura alto nível dos *frameworks* utilizados

As principais diferenças entre o IRF e os outros *frameworks* de recomendação aqui citados são os componentes destinados ao ambiente de produção, tais como interpretadores de entrada e serializadores de saída. Uma característica importante do IRF é a separação dos setores, sendo que cada setor pode ser visto como uma máquina que utiliza uma instância do IRF. Os setores são separados, de forma que o setor de *Batch* (processamento em lote) é separado do setor de *Cache* (fornece a recomendação para o mundo exterior). Desta forma o ambiente está preparado para ser distribuído e permitir que o processamento em lote seja executado em um *cluster* e opere independente dos outros setores. Outra diferença entre o IRF e o *Mahout*, é que o *Taste* está focado em fazer recomendações utilizando somente filtragem colaborativa. Sendo assim ao implementar aplicações que utilizam abordagens de recomendação diferentes faz-se necessária a implementação de métodos, modelos de dados e componentes próprios para estas aplicação.

[4]http://taste.sourceforge.net
[5]http://mymediaproject.codeplex.com
[6]http://lucene.apache.org/mahout
[7]O *Apache Hadoop* é um software de código aberto para aplicações de computação escalável e distribuída.

5.1 Descrição dos três setores de recomendação

Em geral a maioria das informações referentes ao *Framework de Recomendação Idealize* como a arquitetura e descrição dos componentes foram baseadas em [8].

A finalidade dos três setores é dissociar os componentes que possuem operações custosas computacionalmente, e permitir que as recomendações sejam fornecidas de forma instantânea. O fornecimento das recomendações em *O(1)* é possível devido ao setor de *Cache*. Como está exposto, existem três setores e cada um deles é composto por diversos *hot spots* que são instanciados ao iniciar a execução de cada setor. Posteriormente será detalhada a função de cada um destes *hot spots* que compõem os setores. A comunicação remota entre os componentes destes setores mostrada na Figura 1 é feita utilizando *Remote Method Invocation* (RMI)[8], um recurso da linguagem Java para comunicação entre objetos remotos.

A Figura 1 mostra a relação entre os três setores, e a relação de cada setor com a base de dados que pode ser um banco de dados, ou o *Apache Lucene*[9], ou uma estrutura de arquivo, etc.

5.1.1 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*. É destinado a armazenar recomendações pré-calculadas, de tal forma a fornecer respostas instantâneas aos pedidos de recomendações que chegam a sua fachada. Assim como as recomendações, as informações de *feedback* não são processadas neste setor. Ao invés disso, são encaminhados ao setor de *Batch*, para que o mesmo faça o processamento necessário. O setor de *Cache* tem inteligentes heurísticas de gestão de cache para decidir quais informações devem ser mantidas em memória principal e quais devem ser armazenadas no disco.

5.1.2 Setor de *Batch*

Este setor é responsável pelo processamento de recomendações e *feedbacks* referentes aos itens, que são repassados pelo setor de *Cache* para este setor. Hoje o processamento é realizado apenas por uma máquina, porém está destinado a ser executado em um *cluster*. O setor de *Batch* registra alguns de seus componentes de maneira a torná-los remotamente acessíveis utilizando RMI. O setor de *Cache* acessa o setor de *Batch* para requisitar novas recomendações e enviar informações de *feedback*. O setor de *Input* notifica o setor de *Batch* quando há modificações na base de dados. A base de dados contém as informações sobre os itens que estão sendo recomendados, esta base de dados pode ser constituída por um *SGBD*[10], ou documentos indexados pelo *Lucene*[11] ou uma estrutura de arquivos.

[8] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

[9] Lucene é um *framework* de indexação de documentos, permite realizar buscas dos documentos após a indexação destes.

[10] Sistema de Gerenciamento de Banco de Dados

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

5.1.3 Setor de *Input*

O setor de *Input* permite a comunicação entre o mundo externo e as diversas bases de dados a serem utilizadas. 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 e processamento de *feedback* das tarefas de gerenciamento das bases de dados. Como as duas tarefas são custosas computacionalmente, o ideal é manter esses setores trabalhando em máquinas separadas. Além disso, separar a gerência de dados de outras tarefas facilita a criação de interfaces administrativas, de modo a tornar a gestão de dados de forma mais fácil para o usuário.

5.2 Componentes do IRF

O *Framework de Recomendação Idealize* possui dois pacotes *base* e *hot spots*. Os *hot spots* devem ser implementados de acordo com cada aplicação a ser desenvolvida, porém componentes que constituem *frozen spots* tais como, *IdealizeRecommenderFacade, IdealizeInputFacade, Cache e CacheObserver*, possuem implementações comuns para todas aplicações e devem ser utilizados da forma como estão implementados. Este fato ocorre porque os *frozen spots* são responsáveis pela comunicação entre os diversos *hot spots* da aplicação. Por exemplo, o componente *IdealizeRecommenderFacade* que designa o fluxo inicial quando chegar um pedido de recomendação, determinando o componente que deve interpretar a *string* de requisição. Os *frozen spots* estão contidos em um pacote chamado *base*, juntamente com outras classes de auxílio. Entre as classes de auxílio podemos destacar as classes utilizadas para lançar exceções ou para carregar o arquivo de propriedades que será detalhado em seguida.

Os componentes hoje presentes no IRF 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.

5.2.1 Descrição dos componentes do pacote *base*

- *classes.properties:* Arquivo de configurações da aplicação. Diferentes aplicações de recomendação devem ser executadas para cada estratégia de recomendação (CF, CB, UD e HB). O arquivo de propriedades permite que as aplicações sejam configuradas sem a necessidade de fazer modificações no código, independente

da estratégia de recomendação. 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 outros. Estas propriedades estáticas são carregadas para os respectivos atributos estáticos da classe *Constants*, para que estes possam ser utilizados por outras classes da aplicação. O desenvolvedor pode apenas configurar o arquivo de acordo com suas necessidades, e iniciar a execução do setor correspondente. Esta abordagem foi escolhida para que não exista necessidade de mudanças no código quanto à utilização de aplicações desenvolvidas sobre o mesmo *framework* para diferentes clientes.

- *Constants:* As informações configuradas no arquivo de propriedades, que não são referentes a classes concretas a serem instanciadas, são em geral carregadas para a classe *Constants*, onde cada aplicação deverá possuir uma implementação desta classe. Esta classe serve para armazenar de maneira estática valores que são utilizados por todo o sistema, assim como, definição do endereço IP de objetos a serem acessados remotamente, separadores de campos e número de itens a serem calculados suas recomendações ao iniciar a máquina, afim de serem armazenados em cache, entre outros.

- *PropertiesLoader:* Carrega o arquivo de propriedades *classes.properties*.

- *Instantiator:* Esta classe é responsável por ler cada propriedade do arquivo de configurações e instanciar os *hot spots* de acordo com as informações no arquivo de configurações. A política de carregamento e quais propriedades serão lidas é definida pelo desenvolvedor 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 ou dados de usuário chegam sempre por fachadas que são subclasses desta. Todos os dados que chegam e saem das fachadas estão em forma de *string*. 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 de forma a serem acessados remotamente. A idéia por trás de uma interface de marcação é a de que a semântica das relações seja mantida, porém a relação pode acontecer de diversas maneiras.

- *IdealizeRecommenderFacade:* Esta classe implementa a interface *RemoteIdealizeRecommenderFacade* que define os métodos *getRecommendations e addFeedback.* Além dos métodos de sua superclasse, esta possui métodos que configuram as referências de *hot spots*, tais como *Controller, RecommendationSerializer e InputInterpreter.*

- *IdealizeInputFacade:* Esta classe também permite a comunicação entre o mundo externo e a aplicação. Além dos métodos de sua super classe que são relacionados com as operações de inserção, atualização ou remoção de itens e usuários, possui

métodos que configuram as referências de *hot spots* tais como, *InputController* e *InputInterpreter* sendo um interpretador de entrada para operações relacionadas aos usuário e outro interpretador de entrada para operações relacionadas aos itens, ambos presentes na base de dados.

- *Cache:* O componente *Cache* é utilizado em todos os setores. Este componente tem a função de armazenar os dados em cache para acelerar o tempo de resposta. Além desta funcionalidade, o *Cache* apresenta um mecanismo para permitir que o sistema se recupere automaticamente. Geralmente, por causa da enorme quantidade de dados que estão sendo manipulados, a construção de uma recomendador pode levar horas ou mesmo dias. Se por exemplo, devido a uma queda de energia o sistema é desligado, seria inviável esperar todo o tempo novamente para reiniciá-lo. O componente *Cache* possui um método *setData(BaseStorable storable, boolean serialize)*, que permite indicar se deseja armazenar ou não um objeto em disco (objeto do tipo *BaseStorable*, *hot spot* detalhado posteriormente). Uma vez que objetos que implementam *Serializable* da linguagem Java, possuem a capacidade de serem armazenados e recuperados do disco, quando se coloca um objeto *BaseStorable* em cache este pode ser armazenado automaticamente em disco, o que estabelece um ponto de recuperação para o sistema.

- *CacheObserver:* Este é um outro padrão de projeto utilizado, onde o *CacheObserver* guarda a informação de qual foi o último objeto armazenado em *Cache*. O *Cache* faz a notificação ao *CacheObserver* indicando qual foi o último objeto colocado em *Cache*. Esta estratégia permite que o sistema se recupere automaticamente a partir do último objeto armazenado pelo componente *Cache*.

- *IdealizeExceptions:* As exceções do IRF implementadas basicamente apresentam 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.

5.2.2 Descrição dos *hot spots*

- *AbstractInstantiatorWorker:* Super classe abstrata para todas as estratégias de implementação de instanciação das classes concretas. Esta classe basicamente contém todos os campos a serem recuperados a partir do arquivo de configuração. Então quando o desenvolvedor implementar uma estratégia de instanciação dos componentes 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*. Este *hot spot* é responsável por interpretar os diferentes formatos de *strings* que chegam à fachada, para pedidos de recomendações, *feedback* e operações realizadas sobre a base de dados de itens e usuários. O desenvolvedor deve implementar a interface *InputInterpreter* de acordo com a forma que irá interpretar as strings de entrada da fachada. A classe concreta deve ser informada no arquivo de configuração.

- *BaseBean:* Um objeto do tipo *BaseBean* é retornado pelo componente *InputInterpreter*. Este objeto que encapsula os dados da *string* de entrada, vinda da fachada. Levando em consideração que todos os *beans* devem ter um campo que indica o id do item, este *hot spot* contém este atributo e métodos *getters e setters* para este atributo.

- *IdealizeDataModel:* Esta é uma interface de marcação para objetos que representam modelos de dados. Um exemplo de interface de marcação é a *Serializable* do Java que marca objetos que podem ser armazenados em disco. Embora os objetos possam ter diversas interfaces ou maneiras de serem serializados, a interface *Serializable* diz à máquina virtual que esse objeto pode ser serializado e isso 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. 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 *Lucene*, já o *DataModel* dos métodos de recomendação por filtragem colaborativa possuem uma matriz em memória principal, porém todos implementam esta interface de marcação, sendo que cada um implementa da sua forma.

- *IdealizeRecommender:* Este componente é 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, pelas mesmas razões apresentadas para *IdealizeDataModel*. Um novo método de recomendação deve ou tornar-se subclasse da superclasse adequada ou implementar diretamente a interface *IdealizeRecommender*. Nas aplicações desenvolvidas foram implementadas as seguintes subclasses de *IdealizeRecommender: IdealizeCFAbstractRecommender, IdealizeCBAbstractRecommender, IdealizeUDAbstractRecommender e IdealizeHBAbstractRecommender*. Cada uma destas foi utilizadas por métodos de recomendação que possuem as respectivas abordagens de recomendação, filtragem colaborativa, baseada em conteúdo, dados de uso e híbrida.

- *Controller:* É o componente que controla a execução do fluxo da aplicação instanciada. Este componente recebe um *Input-Bean* vindo do *InputIntepreter* e devolve uma lista de *RecommendedItem*[12] para ambientes de recomendação. O controlador deve ser implementado pelo criador dos novos métodos, pois a política de armazenamento de dados em *cache*, ou sequência de operações a serem realizadas é definida por ele de maneira diferente para cada aplicação.

- *BatchProcessor:* Responsável por realizar o processamento em lote e o processamento em paralelo. Este componente irá se comunicar com o sistema distribuído. Existe uma referência para um objeto desse tipo dentro de *Controller* e uma referência para um objeto deste tipo acessado remotamente em *InputController*. O

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

desenvolvedor deve implementar sua lógica de processamento em lote construindo uma subclasse de *BatchProcessor*. Esta é uma subclasse de *RemoteBatchProcessor* que é uma interface criada para que componentes deste tipo possam ser registrados e acessados remotamente.

- *BaseStorable:* Interface a ser implementada por objetos que podem tanto ser armazenados temporariamente em *Cache* para melhorias de desempenho, quanto serem armazenados em disco para recuperação 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 isso, esta interface define os métodos *update, serialize e restore*, além da classe interna *UpdateInput*. A classe *UpdateInput* define os dados de entrada para o método *update*, responsável por atualizar os dados do objeto. O desenvolvedor deve implementar nas extensões de *BaseStorable*, tanto a classe *UpdateInput* quanto o método *update*.

- *RecommendationSerializer:* Como já mencionado todas informações retornadas pela fachada estão em forma de *strings*. Este *hot spot* é responsável por serializar em uma *string* a lista de itens recomendados (*RecommendedItem*) retornados pelo controlador, colocando as recomendações na *string* com padrão esperado por qual outro aplicativo fará o uso destas informações. O desenvolvedor deve implementar a interface *RecommendationSerializer* de acordo com sua necessidade. Muitas vezes a string de retorno tem o seguinte formato "itemId1,itemId2,itemId3...", sendo os itens recomendados separados por vírgula, porém este formato pode ser modificado facilmente por meio de arquivo de configuração.

- *DataManipulator:* Este componente está presente apenas no setor de *Input*, e tem o papel de manipular a base de dados. Por exemplo, no caso de um banco de dados relacional, este componente deve conhecer as estruturas das tabelas e saber operar sobre elas. Por outro lado, se o conjunto de dados está contido no *Apache Lucene*, o componente deve saber como manipular os documentos do *Lucene*, e assim por diante, de acordo com a estrutura de armazenamento de dados utilizada pela aplicação sendo desenvolvida. Para operações de inserção, atualização ou remoção de itens ou usuários deve ser implementada uma subclasse de *DataManipulator*.

5.3 Implementação da aplicação baseada em conteúdo

As aplicações de recomendação baseadas em conteúdo, dados de uso e híbrida foram desenvolvidas sobre o IRF e geraram um artigo [5] no período letivo 2010/02. Em geral as informações aqui detalhadas foram baseadas no artigo com foco na seção da aplicação baseada em conteúdo.

Recomendação baseada em conteúdo tem sua origem na área de Recuperação da Informação [2]. Em geral, recomendações que utilizam esta abordagem podem ser feitas a partir da semelhança entre itens e podem também gerar recomendações de acordo com as informações definidas no perfil de um usuário [1, 5].

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. Já na segunda abordagem são utilizados dados diversos do usuário, que definem um perfil de interesses do usuário, sendo recomendados então itens que correspondem a esses interesses. Devido a restrições de identificação de usuário, apenas a primeira abordagem foi utilizada [5].

Normalmente os itens encontram-se armazenados em um banco de dados. Porém, na aplicação baseada em conteúdo (CB) desenvolvida sobre o IRF foi utilizado o *Lucene*[13]. A indexação processa os dados originais gerando uma estrutura de dados interrelacionada 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. Uma vez que para possibilitar a consulta e análise de similaridade textual o próprio *Lucene* já realiza o armazenamento dos dados, optou-se pela utilização deste recurso em detrimento ao armazenamento em banco de dados, evitando assim a replicação dos dados.

Os itens de recomendação baseada em conteúdo serão tratados como documentos do *Lucene*. Um documento do *Lucene* contém um ou mais campos. A Tabela 1 representa estes documentos. As linhas representam os documentos e as colunas os campos.

id	**Nome**	**Descrição**	**Preço**
10	Mini adaptador estéreo	Converte um min pino 1/8 estéreo em pino dupla.	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 1: Documentos Lucene.

5.3.1 Algoritmo K-NN

O *K-Nearest Neighbor* é um método usado em várias aplicações para classificação textual [10]. Em resumo, classifica documentos de acordo com os K vizinhos mais próximos deste documento. Para utilizar o K-NN na reprodução de recomendações foi utilizado o

[13]Lucene é um *framework* de indexação e buscas textuais.

mesmo princípio, porém foram necessárias algumas adaptações para recomendar itens baseado em 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 [7, 11] 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 *Mahout*. A partir deste ponto recomenda-se os K vizinhos mais próximos deste documento, sendo que quanto mais a 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*

5.3.2 Implementação dos *hot spots*

A implementação dos *hot spots* de cada setor para aplicação de recomendação baseada em conteúdo são descritos a seguir.

Setor de *Batch*

O setor de *Batch* é responsável por produzir as recomendações e enviar a lista de recomendações para o setor *Cache*. O setor responsável pelo desempenho do sistema na aplicação de CB é dependente deste setor.

- *InstantiatorWorker:* Esta classe é uma subclasse de *AbstractInstantiatorWorker* e foi implementada para que os outros componentes da aplicação 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. Também são realizados o carregamento dos campos estáticos para a classe *Constants*. Nesta implementação a fachada é registrada para que seja remotamente acessível pelo setor de *Cache*.

- *InterpreterInput:* 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 de requisição. O campo *ITEMID* é o id do item passado 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 para a *string* "itemId1,ItemId2,..."neste caso o *SepRecommendations* foi definido como uma vírgula. E *SepBatch* é o separador de lote das recomendações, utilizado quando o programador desejar separar por algum critério as recomendações retornadas.

- *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*. O interpretador de entrada configura o id do item, a quantidade de itens a serem recomendados, o separador dos id's 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* que é criado para facilitar a recuperação destas informações através dos métodos *getters*.

- *RecommendeItem:* 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 o item de referência cujo identificador é enviado na requisição das recomendações e os itens recomendados. O valor de similaridade é necessário para a ordenação dos itens, de forma a recomendar os mais relevantes, sendo estes os que possuem os valores mais próximos de zero.

- *Serializer:* Este componente foi implementado a fim de transformar em uma *string* as informações a respeito dos itens que serão recomendados. A lista de recomendação é serializada em uma *string* com o seguinte formato: "ItemId<sep>ItemId<sep>ItemId...". Tendo esta *string* a quantidade de id's de itens referente a requisição realizada pelo setor de *Cache*. Esta classe implementa a interface *RecommendationSerializer* definida no IRF.

- *Storable:* Nesta aplicação, o setor de *Batch* possui um *Storable* do recomendador para que este possa ser colocado no componente*Cache*. O *CBStorableRecommender* não é serializado em disco, pois a construção dos recomendadores para métodos baseados em conteúdo é instantânea. Além disso, 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. A presente implementação possui uma classe interna para atualização dos dados que contém um atributo, sendo uma lista de documentos e um caractere para dizer qual a operação que foi realizada na base, sendo 'i' representação de inserção, 'r' para remoção e 'u' para representar atualização.

- *Controller:* Este componente controla o fluxo das recomendações neste setor. Foram implementados métodos de forma a recuperar o id e a quantidade de recomendações do *bean* passado no 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.

- *Recommender:* Para gerar recomendação baseada em conteúdo implementamos o método *recommend* que recebe o id 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 do 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 do 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 *Lucene* é um objeto instanciado dentro do *DataModel*.

  Esta aplicação não utiliza *feedback*, pois os recomendadores baseado em conteúdo desconsideram estas informações.

Setor de *Cache*

Este setor é responsável por responder diretamente às requisições dos usuários ou às aplicações que requisitam recomendações. 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 InterpreterInput* que apesar de serem utilizados neste setor não estarão descritos nesta seção uma vez que foram detalhados na seção do setor de *Batch*.

- *InstantiatorWorker:* Este componente foi implementado com a finalidade de instanciar a fachada de *Cache*. Ele 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 este possa ser acessado remotamente e processar as recomendações para serem armazenadas em memória principal. Também são recuperados o separador de campos, o tempo necessário para recalcular as recomendações e o número máximo de recomendações a serem carregadas na inicialização deste setor.

- *Storable:* O *Storable* deste setor possui recomendações pré-calculadas. Estas recomendações são armazenadas em um *hash map* que contém os id's dos itens

como chave e uma lista de recomendação contendo o id de outros itens que estão sendo recomendados baseados 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 uma heurística que mantém as recomendações dos itens mais relevantes em memória principal, observando a quantidade de pedidos de recomendações.

- *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 então realiza o pedido para a fachada do setor de *Batch*. Assim que a recomendação é calculada, é armazenada em *Cache* para atender futuras requisições de recomendação.

- *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 é retornada a recomendação este componente coloca em *cache*. Uma vez que as recomendações estiverem em *Cache* neste setor, podem ser acessadas rapidamente para atender as requisições dos usuários.

Este setor não possui implementação para os *hot spots* tais como, *DataModel, DataManipulator, Recommender, InputController*. Outros setores são responsáveis por implementar e utilizar estes componentes quando se implementa aplicações baseadas em conteúdo sobre o *framework Idealize*.

Setor de *Input*

Este setor é responsável por fazer 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 sem a utilização deste setor de forma a possuir um diretório com uma grande quantidade de documentos já indexados pelo *Lucene*. Quando a máquina estiver rodando este setor será responsável por remover, atualizar ou inserir novos documentos no *Lucene* e consequentemente no *DataModel* de CB, já que o *CBDataModel* possui o objeto do *Lucene*.

- *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*.

- *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*, o qual decidirá quem irá processar o *InputBean* criado. 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* tem a quantidade de campos que um documento contém. 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 id do item, e além deste atributo possui um *hash map* que tem as chaves são os nomes dos campos e os valores são os valores dos campos do documento que será indexado pelo *Lucene*. Desta forma obtém maior facilidade na recuperação destas informações do *bean* devido aos métodos implementados para recuperar estas informações.

- *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 tamanho grava os documentos na base de dados *(Lucene)* e em seguida o controlador notifica os *BatchProcessor's* remotos indicando que houve alterações na base de dados.

- *DataManipulator:* Todos os componentes são importantes, porém neste setor este componente deve receber atenção especial. *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 foram modificados é armazenada. Quando esta quantidade atingir um valor definido no arquivo de propriedades ou após transcorrido um tempo determinado também definido no arquivo de propriedades, o *BatchProcessor* do setor de *Batch* será notificado para proceder com as atualizações necessárias, levando em consideração as modificações ocorridas na base de dados. O *BatchProcessor* do setor de *Cache* será notificado e realizará pedidos para calcular as recomendações dos itens alterados na base de dados.

6 Trabalhos Futuros

Um trabalho futuro a ser realizado é a implementação da arquitetura escalável no *Framework de recomendação Idealize*. Desenvolver uma aplicação que utilize a nova arquitetura escalável, podendo ser esta aplicação baseada em conteúdo, filtragem colaborativa, baseada em dados de uso ou híbrida.

7 Cronograma de atividades

Na Tabela 2, segue o cronograma das atividades realizadas.

Atividades	**Ago**	**Set**	**Out**	**Nov**	**Dez**
Implementação dos setores de CB	X	X	X		
Redigir e apresentar a Monografia			X	X	
Estudo de escalabilidade					X

Tabela 2: Cronograma de Atividades.

8 Reconhecimentos

Gostaria de fazer os reconhecimentos as instituições e pessoas que desempenharam papel importante para realização deste trabalho. Primeiro ao Laboratório Idealize por possuir pessoas competentes e que apoiaram o desenvolvimento do trabalho. Em seguida a UpLexis[14] por estar patrocinando o desenvolvimento dos projetos realizados no Laboratório Idealize. E por fim, a Universidade Federal de Ouro Preto (UFOP)[15] por possuir uma infra-estrutura que hospede o Laboratório Idealize, onde foi possível desenvolver este trabalho.

[14] http://www.uplexis.com
[15] http://www.ufop.br

Referências

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

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

[4] Carlos Castillo, Debora Donato, Ranieri Baraglia, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Aging effects on query flow graphs for query suggestion.

[5] Reinaldo Fortes, Saul Delabrida, Felipe Melo, Álvaro R. Pereira Jr., Alex Amorim Dutra, and Milton Stiilpen Jr. Applications of an open source general-purpose recommendation framework. In *20th World Wide Web Conference*, 2010. submitted.

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

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

[8] Felipe Melo and Álvaro Pereira Jr. Idealize recommendation framework - a framework for general-purpose recommender systems. In *20th World Wide Web Conference*, 2010. submitted.

[9] 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.

[10] 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.

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

[12] 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.