UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

INTEGRANDO RECUPERAÇÃO DE INFORMAÇÃO EM BANCO DE DADOS COM HIBERNATE SEARCH

Gustavo Kendi Tsuji
Leonardo Tadashi Kamaura

São Paulo
2008

GUSTAVO KENDI TSUJI
LEONARDO TADASHI KAMAURA

INTEGRANDO RECUPERAÇÃO DE INFORMAÇÃO EM BANCO DE DADOS COM HIBERNATE SEARCH

Trabalho de Conclusão de Curso de Bacharelado em Ciências da Computação do Instituto de Matemática e Estatística

Orientador: João Eduardo Ferreira

São Paulo
2008

Agradecimentos

Agradecemos ao Prof. Dr. João Eduardo Ferreira, pela orientação e oportunidade oferecida para a elaboração deste trabalho. Sua atenção e interesse no desenvolvimento do projeto e suas múltiplas sugestões de aperfeiçoamento do texto foram fundamentais para a conclusão do TCC.

Agradecemos também ao pós-graduando Tiago Motta Jorge pela sua ajuda e contribuição com o seu trabalho na disciplina de Programação eXtrema, que serviu como ponto de partida para o desenvolvimento deste trabalho.

Finalmente, agradecemos ao Prof. Dr. Eduardo Colli e aos bibliotecários do IME, pela paciência e atenção dispensadas ao projeto Colméia, essencial para a aplicação do nosso estudo de caso.

Lista de Ilustrações

Lista de Tabelas

Sumário

1. Introdução

Um dos grandes desafios na área de banco de dados foi conseguir desenvolver sistemas que permitissem armazenar e recuperar dados de forma eficiente. Em paralelo a evolução de gerenciadores de banco de dados, surgiu a "tentativa de automatizar as buscas de registros" e a necessidade de extrair *informações* dessas fontes de dados, posteriormente denominada *recuperação de informação*.

Um exemplo que ilustra este cenário é o caso de sistemas de buscas na internet. Estima-se que houve um crescimento de 300%[1] na quantidade de usuários da Internet no período entre 2000 a 2008. Com o desenvolvimento acentuado da internet, se antes a estrutura dos dados era bem conhecida e seu volume considerado, atualmente temos dados semi-estruturados com um volume infinitamente maior. Portanto um importante desafio que se apresenta é a recuperação de dados *relevantes* utilizando mecanismos avançados de buscas.

Outro exemplo é o caso das pesquisas sobre os complexos processos biológicos que governam o corpo humano. Técnicas avançadas como os *microarrays* de DNA permitem que um enorme volume de dados sobre genes e proteínas possa ser gerado, o que requer métodos rápidos para pesquisar e interpretar de forma automatizada tais fontes abundantes de informação. (SHATKAY; EDWARDS; BOGUSKI, 2002, p. 45)

Este trabalho tem por objetivo expor uma visão geral sobre a recuperação de informação, explorando os mecanismos desse processo e principalmente, a integração de banco de dados e a indexação. Depois de apresentar conceitos sobre o tema, utilizaremos como caso de estudo a aplicação do arcabouço Hibernate Search no projeto Colméia, um sistema de gerenciamento de biblioteca.

[1] WORLD Internet Usage Statistics News and World population Stats. Disponível em : <http://www.internetworldstats.com/stats.htm>. Acesso em: 6 nov. 2008

2. Fundamentos

2.1. A diferença entre dado recuperado e informação

A *recuperação de informação* (RI) é uma subárea da ciência da computação que estuda como recuperar dados, informações ou documentos de alguma fonte de dados (tabelas relacionais, documentos, páginas na internet). Para Baeza-Yates e Ribeiro-Neto (1999, p.1), trata-se de um processo que se inicia com a representação e armazenamento e estende-se até a organização e acesso à informação.

No contexto de RI, a recuperação de dados consiste principalmente em determinar quais itens de uma coleção contêm as palavras-chaves de uma consulta a um determinado banco de dados. Já, recuperação de informação, extrai as informações, interpretando-as tendo em vista a semântica e a sintaxe.

A principal diferença é que a recuperação de informação lida com textos de linguagem natural que nem sempre está estruturada e semanticamente pode ser ambígua. Já a recuperação de dados trabalha com dados com cuja a estrutura e a semântica são muito bem definidas.

Para exemplificar um sistema de RI, suponha a seguinte busca:

> *Encontrar todos os livros que: 1) foram escritos em 2008 e 2) que tenham ligação com o FISL e para ser relevante, deve incluir informações sobre a obra, autores e editoras.*

Contudo, o sistema não conseguirá buscar os registros se o usuário descrevê-los dessa forma. Neste caso, é preciso traduzir esta descrição em uma linguagem (**queries)** para ser processada pelo sistema de busca por meio de informações fornecidas pelo usuário.

Usualmente, extraem-se termos que representem, de maneira generalizada e concisa, o que se deseja buscar (**palavras-chave)**. Assim, o sistema executa o processo de recuperação, retornando inúmeros resultados com informações relevantes e não relevantes.

2.2. Elementos de recuperação de informação

A visão global de um sistema de recuperação de informação pode ser representada pela Figura 1.

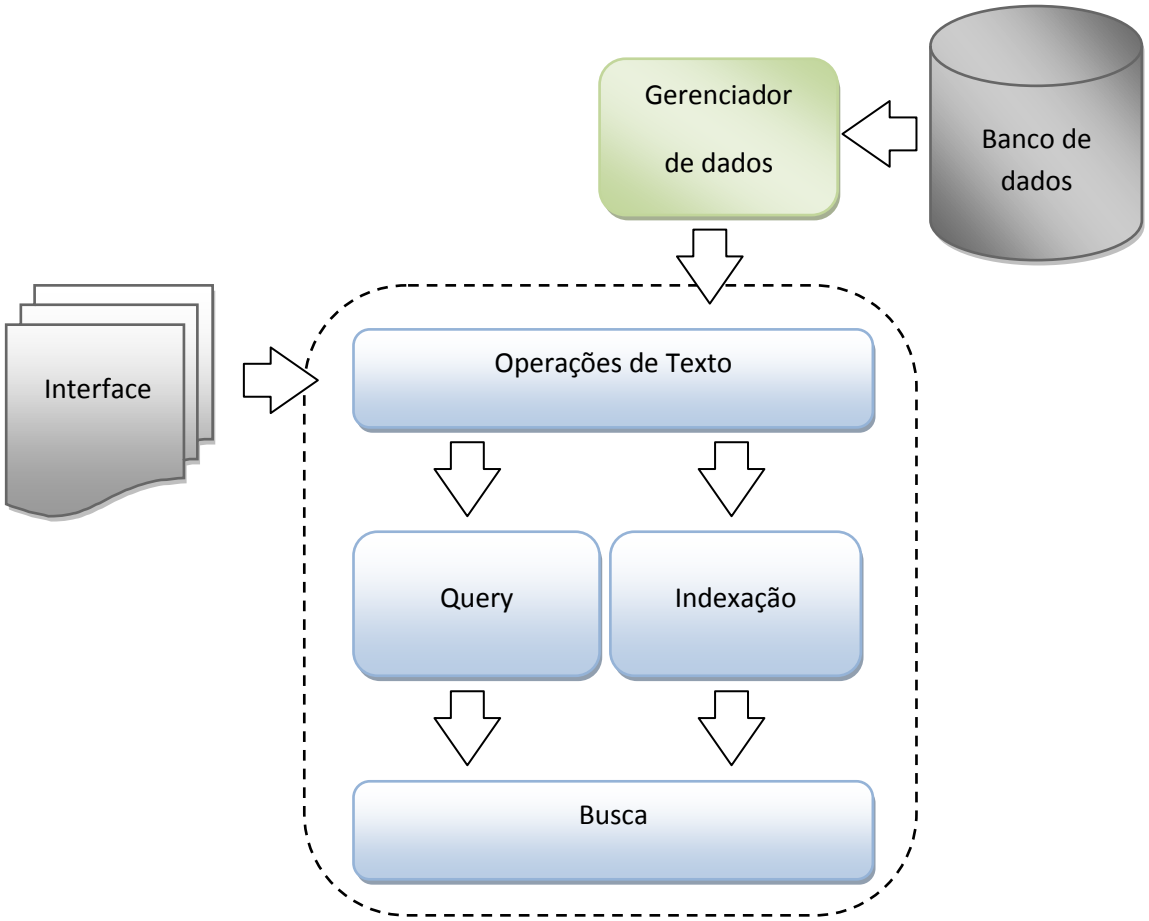

Figura 1 - Processos que envolvem indexação e busca

2.2.1. Documento

Segundo Baeza-Yates e Ribeiro-Neto (1999, p.5), **documentos** representam uma visão lógica da fonte de dados. Essas fontes podem ser representadas por conjuntos de palavras-chaves extraídas do próprio texto da fonte de dados. Em linhas gerais, pode ser um parágrafo, uma seção, um capítulo, uma página web, um registro do banco de dados ou um livro inteiro. O importante é que deve representar um conteúdo pesquisado.

Um documento de texto completo (*full text*) usa todas as palavras da fonte de dados em sua construção. Seria ideal poder trabalhar com documentos de texto

completo, uma vez que eles carregam a maior quantidade de informações. Contudo, é altamente custoso em termos computacionais manter todas as palavras no documento, principalmente quando há uma coleção de documentos muito grande. Assim, durante a geração desses documentos, algumas transformações são feitas para reduzir a quantidade de palavras. O texto sofre a eliminação de *stopwords*[2], como artigos e conectivos (preposições, conjunções, etc.), palavras são reduzidas para sua raiz gramatical (*stemming*[3]) e grupos de substantivos são identificados, retirando-se verbos, adjetivos e advérbios.

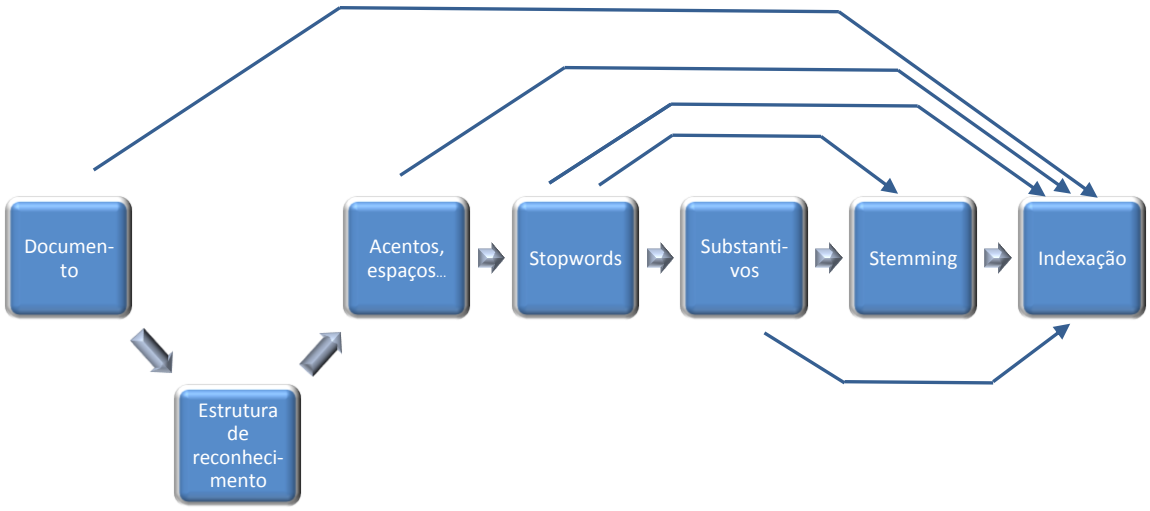

Figura 2 - Processo de tratamentos sobre o documento

(extraída e adaptada de BAEZA-YATES e RIBEIRO-NETO, 1999)

2.2.2. Índice

Os índices são termos pré-selecionados armazenados em estruturas associadas a documentos (BAEZA-YATES e RIBEIRO-NETO, 1999, p.20), permitindo um acesso aos dados de forma mais rápida.

[2] [3] Optamos por manter alguns termos em inglês devido à dificuldade de tradução e ou devido ao conhecimento consagrado do termo em inglês.

Por isso, a geração do índice deve ser feita de forma cuidadosa para que a busca não recupere falsos positivos ou deixe de retornar informações relevantes. A geração dos índices é um dos pontos centrais da RI. A avaliação da relevância dos documentos requer um algoritmo de "ranqueamento (seção 2.3.2, p. 15) eficiente". Um exemplo disso é o sistema de buscas que a empresa Google desenvolveu. Neste trabalho, iremos detalhar o arquivo invertido, que é a estrutura mais utilizada para ranqueamento e indexação.

2.3. Etapas da recuperação de informação

Os processos de recuperação de informação podem ser descritos em duas etapas: a criação de índices e o processo de recuperação.

2.3.1. Criação dos índices

Os dados são extraídos do banco e, utilizando operações de texto (Figura 1), gera-se o **documento**. Quando os documentos estiverem prontos, o passo seguinte é construir os índices. Com os **índices** criados, o processo de RI já está pronto para recuperar dados de forma eficiente e rápida.

Processo de Indexação

Quando usamos uma consulta básica, uma opção óbvia é usar a busca *online*, isto é, varrer o texto seqüencialmente, procurando pelas ocorrências dos termos da consulta. A busca *online* é apropriada quando trabalhamos com pequenos textos e é a única opção quando a coleção de textos é muito volátil, isto é, suscetível a alterações freqüentes ou quando não é possível armazenar o índice por falta de espaço (BAEZA-YATES e RIBEIRO-NETO, 1999, p.191).

Para coleções de textos grandes e semi-estáticas, passa a ser interessante utilizar uma estrutura de dados sobre o texto, os índices, para acelerar a busca. Por coleções semi-estáticas, entenda-se, coleções de textos que sofrem modificações em intervalos razoavelmente regulares (diariamente, por exemplo). Exemplos de coleções semi-estáticas são documentos na *Web*, bancos de dados de textos, dicionários e obras literárias.

Arquivo invertido

Conforme Baeza-Yates e Ribeiro-Neto (1999, p. 192), a técnica de indexação mais aplicada e apropriada para a maioria das aplicações é a do arquivo invertido. É um mecanismo, orientado a palavras, para a indexação de uma coleção de textos, cujo objetivo é acelerar o processo de busca. Um arquivo invertido é dividido em dois elementos estruturais: o **vocabulário** e as **ocorrências**. O vocabulário é o conjunto de todas as palavras distintas de um texto. As ocorrências são o conjunto de listas que contém as posições de cada uma das palavras do vocabulário, sendo uma lista para cada palavra. As posições podem se referir a caracteres, palavras, parágrafos, documentos, etc., ou seja, a granularidade é definida de acordo com a necessidade da aplicação.

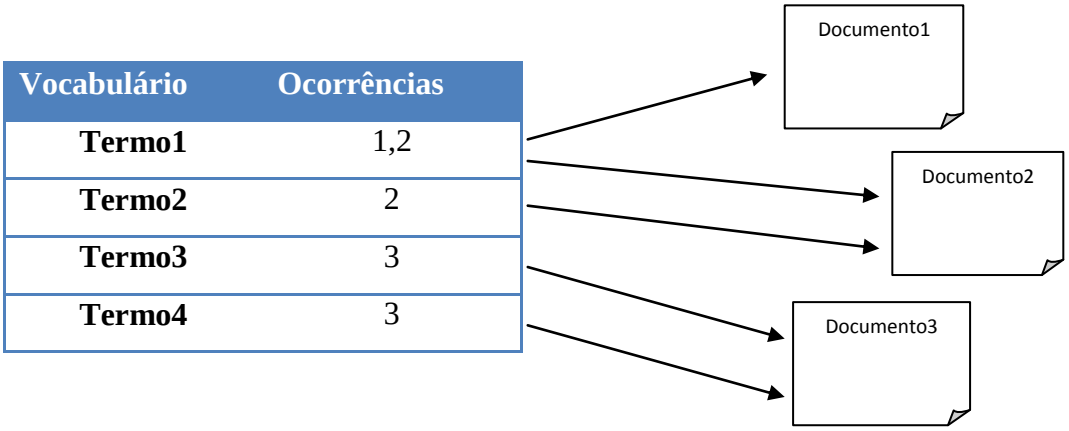

Figura 3 – Visão geral do arquivo invertido

Enquanto o vocabulário ocupa pouco espaço em relação ao tamanho do texto, a demanda por espaço da lista de ocorrências é muito maior. Uma técnica utilizada para reduzir os requisitos de espaço é a do endereçamento por bloco, na qual o texto é dividido em blocos, e as ocorrências apontam para blocos onde a palavra ocorre, ao invés da posição exata. Duas observações são importantes sobre esta técnica: primeira, caso seja necessária a posição exata da palavra, é preciso fazer uma busca *online* sobre o bloco em questão. Segunda, para que a técnica possa ser utilizada, é necessário que o texto esteja prontamente disponível no momento da busca, o que não é o caso, por exemplo, de textos na *Web*, ou de textos que estão em mídias removíveis que não estão montadas.

2.3.2. Processo de recuperação

Usando uma interface gráfica, o usuário descreve o que ele deseja procurar e o sistema processa esses dados, executando operações de texto e de consulta. O resultado é a tradução do que está sendo procurado pelo usuário em uma consulta. O sistema efetua uma busca sobre os índices e gera um ranqueamento, podendo avaliar o grau de relevância do registro recuperado.

Algoritmo de busca

O algoritmo de busca em um arquivo invertido pode ser dividido geralmente em três etapas:

- Busca no vocabulário: as palavras e padrões presentes na consulta são isoladas em termos simples e procuradas no vocabulário;
- Recuperação das ocorrências: a lista de ocorrências de todos os termos da consulta é recuperada;
- Manipulação das ocorrências: as ocorrências são processadas para encontrar frases, resolver relações de proximidade e operações booleanas.

Para Baeza-Yates e Ribeiro-Neto (1999, p. 196), o custo de resolver consulta é sublinear ao tamanho do texto. A complexidade é de $O(n^{\alpha})$, onde α varia conforme a consulta. Na prática, arquivos invertidos permitem realizar buscas em tempo sublinear, com espaço também sublinear.

A construção e manutenção de um arquivo invertido são tarefas de custo relativamente baixo. Um arquivo invertido pode ser construído em tempo linear ao tamanho do texto. Todo o vocabulário conhecido até então é armazenado em uma estrutura de dados *trie*, que contém para cada palavra, uma lista de suas ocorrências.

Uma *trie* é essencialmente uma árvore enária, cujos nós são vetores de M posições com componentes correspondendo a dígitos ou caracteres. Cada nó no nível *l* representa o conjunto de todas as chaves que começam com certa seqüência de *l* caracteres. O nó especifica um ramo de M-vias, dependendo do (*l*+1)-ésimo caractere (KNUTH, 1973, p. 481).

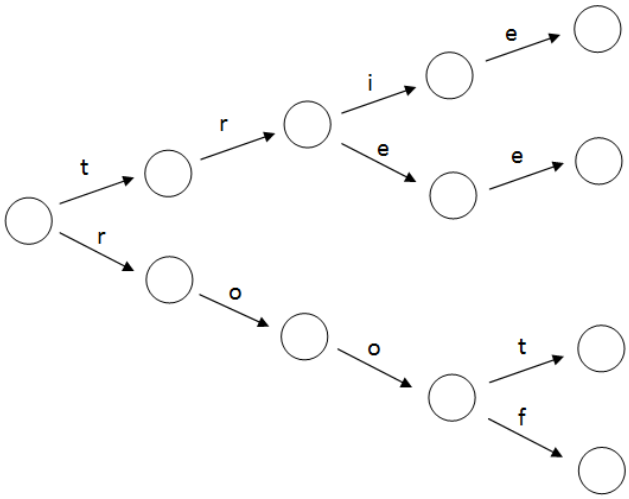

Figura 4 – Exemplo de uma *trie* para as chaves “trie”, “tree”, “root” e “roof”

A *trie* é populada à medida que o texto é percorrido. Se uma palavra é encontrada pela primeira vez, ela é armazenada na *trie* com uma lista de ocorrências vazias. Toda vez que uma palavra já existente na *trie* é lida no texto, a sua ocorrência é armazenada no fim da lista de ocorrências daquela palavra. Após o término da leitura do texto, a *trie* é armazenada no disco. Geralmente, o índice é dividido em dois arquivos: um contendo o vocabulário, em ordem lexicográfica e outro contendo a lista de ocorrências das palavras, contiguamente. Quando não há espaço na memória suficiente para armazenar todo o índice durante a sua construção, é feito uma divisão em índices parciais que são armazenados em disco e, posteriormente, unificados. A atualização de um índice, após a inclusão de um novo texto de tamanho N, leva O(n + N log (N/M)), no qual n é o tamanho do texto atual e M é a quantidade de memória principal disponível. A deleção de um texto pode ser feita em tempo linear ao tamanho do texto a ser removido, eliminando as ocorrências que apontam para áreas do texto removido e eliminando palavras cuja lista de ocorrências passe a ser vazia.

Ranqueamento

Um dos problemas centrais de sistemas de RI é saber quais documentos são relevantes à busca do usuário e quais não são. Esta tomada de decisão é feita geralmente por um algoritmo de ranqueamento que procura estabelecer uma ordenação crescente dos resultados obtidos: os primeiros documentos dessa ordenação são mais relevantes. (BAEZA-YATES e RIBEIRO-NETO, 1999, p. 19)

Um algoritmo de ranqueamento, portanto, depende de um conjunto de premissas que julgam o nível de relevância de um documento. Isto significa que diferentes conjuntos de premissas geram diferentes modelos de RI.

2.4. Modelos de RI

Os modelos clássicos consideram que cada documento é descrito como um conjunto de palavras-chave denominadas termos de índices. Um *termo de índice* é uma palavra cuja semântica resume o conteúdo de um documento. Para cada termo, pode-se atribuir um *peso*, isto é, um valor numérico que indicará o grau de relevância daquele termo ao descrever o conteúdo semântico de um documento.

Definem Baeza-Yates e Ribeiro-Neto (1999, p.23):

> *Um modelo de RI é uma quádrupla {D, Q, F, R(q_i, d_j)} onde*
>
> *(1) D é um conjunto de representações para os documentos de uma coleção*
> *(2) Q é um conjunto que representa as informações que o usuário procura (consultas)*
> *(3) F é o arcabouço para a representação da modelagem dos documentos, consultas e seus relacionamentos*
> *(4) R(q_i, d_j) é a função de ranqueamento que associa um número real com a consulta $q_i \in Q$ e a representação do documento $d_j \in D$. Esse ranqueamento irá definir a ordenação entre os documentos da consulta q_i.*

Existem três modelos clássicos na RI: os modelos booleano, vetorial e probabilístico. Além deles, existem também modelos alternativos que tentam refinar ou introduzir novos conceitos, mas neste trabalho nossa ênfase está no modelo vetorial por ser o mais utilizado.

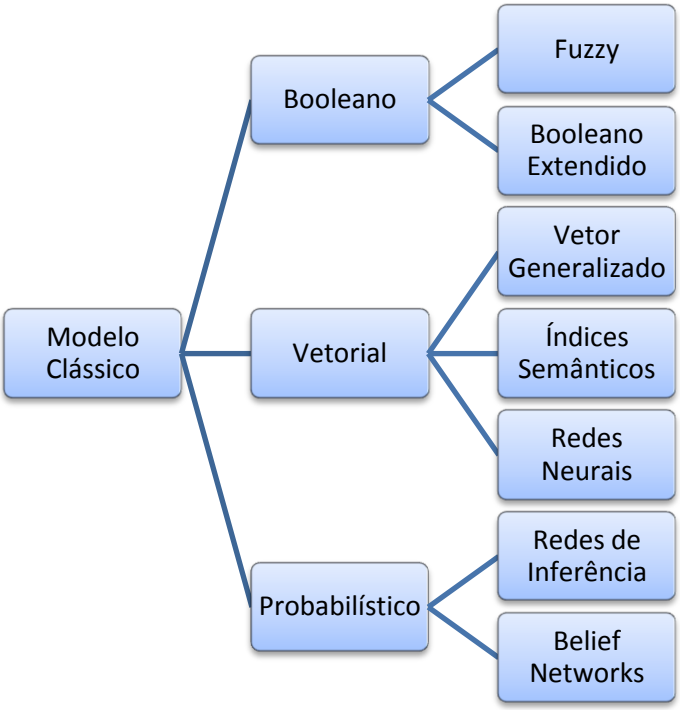

Figura 5 - Divisões do Modelo Clássico

2.4.1. O modelo Booleano

O modelo Booleano é um modelo simples baseado na teoria dos conjuntos e na álgebra Booleana. As consultas são definidas por meio de expressões Booleanas compostas por termos de índice e os conectores *not*, *and* e *or*. Tais expressões podem ser representadas na *forma disjuntiva normal* (DNF). Neste modelo, os termos de índice ou estão presentes num documento ou não estão, portanto, os pesos dos termos de índice assumem valores binários.

Definem Baeza-Yates e Ribeiro-Neto (1999, p.26-27):

> *No modelo Booleano, as variáveis do peso associado a um termo de índice assumem valores binários, isto é,* $w_{i,j} \in \{0, 1\}$. *Uma consulta q é uma expressão Booleana convencional. Seja* $\vec{q}_{dnf}$ *a forma normal disjuntiva da consulta q. Além disso, seja* $\vec{q}_{cc}$ *qualquer componente conjuntivo de* $\vec{q}_{dnf}$. *A similaridade do documento* d_j *à consulta q é definida por:*

$$sim(d_j, q) = \begin{cases} 1, & se\ \exists \vec{q}_{cc} \mid (\vec{q}_{cc} \in \vec{q}_{dnf}) \wedge (\forall k_i, g_i(\vec{d}_j) = g_i(\vec{q}_{cc})) \\ 0, & caso\ contrário \end{cases}$$

O modelo Booleano diz se cada documento é relevante ou não, ou seja, não existe a noção de parcialidade na verificação das condições da consulta.

As vantagens do modelo Booleano são a simplicidade e o formalismo claro subjacente ao modelo. No entanto, a simplicidade do critério binário de decisão acaba restringindo a qualidade dos resultados obtidos, retornando um conjunto de documentos ou muito pequeno ou muito grande. Além disso, apesar da semântica das expressões booleanas ser clara, traduzir uma requisição de informação para uma expressão booleana nem sempre é uma tarefa simples, principalmente para usuários comuns.

2.4.2. O modelo Vetorial

O modelo vetorial[4] é um modelo algébrico para representar os termos de índices dos documentos e das consultas num espaço vetorial. Associado a cada um dos termos, atribui-se um *peso*, um valor não binário (ao contrário do modelo booleano) que, posteriormente, será utilizado no cálculo do *grau de similaridade* entre o documento armazenado no sistema e a consulta do usuário. Assim, dada uma consulta, o modelo vetorial avalia de forma ponderada o quanto um documento é relevante, possibilitando uma correspondência parcial entre a consulta e o documento, fato que não ocorre no modelo booleano.

Definem Baeza-Yates e Ribeiro-Neto (1999, p.27):

> *No modelo vetorial, o peso* $w_{i,j}$*, associado ao par* (k_i, d_j) *é positivo e não binário, onde* k_i *é um termo de índice e* d_j *um documento. Além disso, os termos de índices na consulta também são ponderados. Seja* $w_{i,q}$ *um peso associado ao par* $[k_i, q]$*, onde* $w_{i,q} \geq 0$*. Então, o vetor da consulta* $\vec{q}$ *é definido como* $\vec{q} = (w_{1,q}, w_{2,q}, \ldots, w_{t,q})$ *onde t é o número total de termos de índices no sistema. O vetor do documento* d_j *é representado por* $\overrightarrow{d_j} = (w_{1,j}, w_{2,j}, \ldots, w_{t,j})$.

Os pesos podem ser calculados de diversas formas. A principal idéia subjacente às diferentes técnicas de cálculo desses pesos está associada aos princípios básicos de análise de cluster[5]. Isto é, dada uma descrição *vaga* de um conjunto A, o objetivo é

[4] SALTON, Gerard. The SMART Retrieval System - Experiments in Automatic Document Processing. Englewood Cliffs, NJ: Prentice Hall, 1971.

[5] DUBES, Richard C.; JAIN, Anil K. Algorithms for Clustering Data. Englewood Cliffs, NJ: Prentice Hall, 1988. p. 55-58.

separar uma coleção C de objetos em dois conjuntos distintos: um relacionado ao conjunto A e um outro não relacionado. Esta descrição vaga quer dizer que não há informações suficientes para decidir, com precisão, quais objetos pertencem ou não ao conjunto A.

O problema de RI pode ser visto como o de *clustering*, ou seja, estamos interessados em determinar quais documentos (objetos) da coleção C estão no conjunto A, cuja descrição é dada vagamente por uma consulta. Para resolver este problema, é preciso determinar quais características dão a melhor descrição de objetos do conjunto A (tal conjunto de características quantificam o que é definido como *similaridade intra-cluster*) e quais permitem distingui-las, da melhor maneira (quantificando o que é definido como *dissimilaridade inter-cluster*). No modelo vetorial são utilizados os fatores: *tf factor*, que calcula a quantidade de vezes que o termo k_i aparece no documento d_j, (fornecendo uma medida de quão bem k_i descreve d_j, isto é, a similaridade intra-cluster), e o *idf factor* (*inverse document frequency*) que mede o inverso da freqüência do termo k_i dentro da coleção dos documentos (fornecendo uma medida da dissimilaridade inter-cluster). O principal motivo do uso desse fator idf se deve ao fato de que uma palavra que apareça em muitos documentos não pode ser usada para distinguir os objetos da coleção.

Definem Baeza-Yates e Ribeiro-Neto (1999, p.29-30):

> *Seja N o número total de documentos no sistema e n_i o número de documentos que o índice k_i aparece. Seja $freq_{i,j}$ a freqüência bruta do termo k_i no documentos d_j (isto é, o número de vezes que o termo k_i é mencionado no texto do documento d_j). Então, a freqüência normalizada $f_{i,j}$ do termo k_i no documento d_j é dada por:*
>
> $$f_{i,j} = \frac{freq_{i,j}}{\max_l f_{l,j}}$$
>
> *onde o máximo é computado sobre todos os termos que são mencionados no texto do documento d_j .Se o termo k_i não aparece no documento d_j, então $f_{i,j} = 0$. Além disso, seja idf_i, a freqüência inversa do documento para k_i, dada por:*
>
> $$idf_i = \log \frac{N}{n_i}$$
>
> *Assim, a fórmula, conhecida como tf-idf, dada aos pesos é:*

$$w_{i,j} = f_{i,j} \times \log \frac{N}{n_i}$$

ou uma variação dessa fórmula.

Por fim, para calcular a similaridade entre $\overrightarrow{d_j}$ e $\vec{q}$, utiliza-se a correlação entre os dois vetores, bastando calcular, por exemplo, o cosseno entre $\overrightarrow{d_j}$ e $\vec{q}$:

$$sim(d_j \cdot q) = \frac{\overrightarrow{d_j} \cdot \vec{q}}{|\overrightarrow{d_j}| \times |\vec{q}|} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{j=1}^{t} w_{i,q}^2}}$$

onde $|\overrightarrow{d_j}|$ e $|\vec{q}|$ são as normas do vetor do documento e do vetor da consulta, respectivamente. Baeza-Yates e Ribeiro-Neto ressaltam que $|\vec{q}|$ não afetará a ordenação dos documentos, pois a consulta q é o mesmo para todos os documentos. Além disso, como $w_{i,j} \geq 0$ e $w_{i,q} \geq 0$, $sim(d_j \cdot q)$ varia de 0 até 1. Como o modelo vetorial objetiva quantificar a relevância de um documento, e não simplesmente, definir se o documento é relevante ou não, é possível, analisando o grau de similaridade calculado acima, fazer um ranqueamento dos documentos relevantes.

As principais vantagens de utilizar o modelo vetorial são:

1. O uso de termos de índices ponderados melhora o desempenho da recuperação de informação.
2. Os documentos recuperados tendem a se aproximar mais ao que a consulta descreve, uma vez que a correspondência pode ser parcial entre ambos.
3. Após o cálculo da correlação entre o documento e a consulta, é possível ordenar os resultados obtidos conforme o grau de similaridade.
4. É um modelo simples e rápido.

Uma das desvantagens seria, teoricamente, o fato de supor que os termos de índices são mutuamente independentes. Contudo, na prática, a consideração indiscriminada das dependências entre os termos pode prejudicar o desempenho geral do modelo, devido à questão da localidade.

3. Tecnologias e Aplicações

3.1. Hibernate

O Hibernate[7] é um arcabouço de gerenciamento/persistência de banco de dados para aplicações Java. É responsável por criar mapeamentos objeto/relacional, além de disponibilizar facilidades para recuperar dados. Um dos principais objetivos é reduzir o tempo gasto em questões envolvendo o banco de dados.

3.1.1. Mapeamento

A representação das entidades é feito por meio de *JavaBeans*, classes que não contém nenhuma lógica de negócios, servindo apenas para armazenar dados. O mapeamento do banco de dados pode ser feito de duas maneiras: utilizando um arquivo de configuração em XML ou por anotações. Adotamos o último por ser semanticamente mais fácil de entender e alterar.

3.1.2. Entidades

Uma entidade representa um objeto ou conceito do mundo real, como um empregado ou um projeto, que é descrito em um banco de dados (ELMASRI e NAVATHE, 1999, p. 39).

A descrição dessas entidades é feita utilizando atributos, isto é, campos que representam características da entidade. Os atributos podem ser classificados em:

Simples	Caracteriza uma propriedade que é indivisível
Composto	Reúne um conjunto de atributos simples
Monovalorados	Possui apenas um valor para a propriedade referente
Multivalorados	Registra um ou mais valores para a propriedade referente
Chave	Define unicamente um registro na tabela

Tabela 1 – Tipos de atributos de uma entidade

[7] HIBERNATE.ORG – Hibernate. Disponível em: < http://www.hibernate.org/>. Acesso em: 18/Nov/2008

3.1.3. Relacionamentos

Um relacionamento é uma associação entre entidades. Geralmente, é possível definir papéis para cada entidade que participam do relacionamento. É o caso do relacionamento, por exemplo, entre a entidade Livro e a entidade Autor. Um livro tem um ou mais autores, assim como um autor pode escrever um ou mais livros.

3.2. Hibernate Annotations

O Hibernate Annotations[8] é um módulo do Hibernate que contém metadados responsáveis pela configuração do mapeamento objeto-relacional.

3.2.1. Entidades

Anotação		Descrição
@Entity		Declara que a classe é uma entidade
@Table		Define a tabela e esquema que a classe representará.
	name	Especifica o nome da tabela física no banco de dados
@Id		Declara a propriedade que representa o id da tabela
@Column		Indica que a propriedade do *bean* representa campos da tabela
	name	Especifica o nome do campo da tabela
	unique	Define que a coluna possui restrição de unicidade
	nullable	Indica que a coluna pode conter valores *null*
	length	Declara tamanho do campo

Tabela 2 - Anotações básicas utilizadas no projeto Colméia

[8] HIBERNATE Annotations. Disponível em: <http://www.hibernate.org/hib_docs/annotations/reference/en/html/>. Acesso em: 18/Nov/2008

3.2.2. Relacionamentos

Anotação	Descrição
@OneToOne	Representa o tipo de relacionamento um para um
@ OneToMany	Representa o tipo de relacionamento um para muitos
@ ManyToOne	Representa o tipo de relacionamento muitos para um
@ ManyToMany	Representa o tipo de relacionamento muitos para muitos

Tabela 3 - Anotações básicas utilizadas no projeto Colméia

3.3. Hibernate Search

O Hibernate Search[9] é um arcabouço de busca de texto completo que utiliza as anotações do Hibernate Annotations aliada à API do Lucene (seção 3.4). Seu objetivo é integrar os mecanismos de indexação e recuperação de informação no gerenciamento do banco de dados.

3.3.1. Anotações

Anotação	Descrição
@Indexed	Declara que a classe será indexada
@ DocumentId	Especifica o atributo id da entidade
@ Field	Declara que o atributo da entidade será indexado
index	Especifica a forma como o atributo será indexado. Pode-se configurar um analisador para processar os valores do atributo (Index.TOKENIZED).
store	Especifica se o valor do atributo será armazenado no índice do Lucene.

Tabela 4 - Anotações básicas para indexação

[9] HIBERNATE.ORG – Hibernate Search. Disponível em: <http://www.hibernate.org/410.html>. Acesso em: 18/Nov/2008

3.3.2. Análise

Há basicamente duas funcionalidades importantes, como já foi explicado anteriormente:

1) Processo de indexação
2) Processo de busca

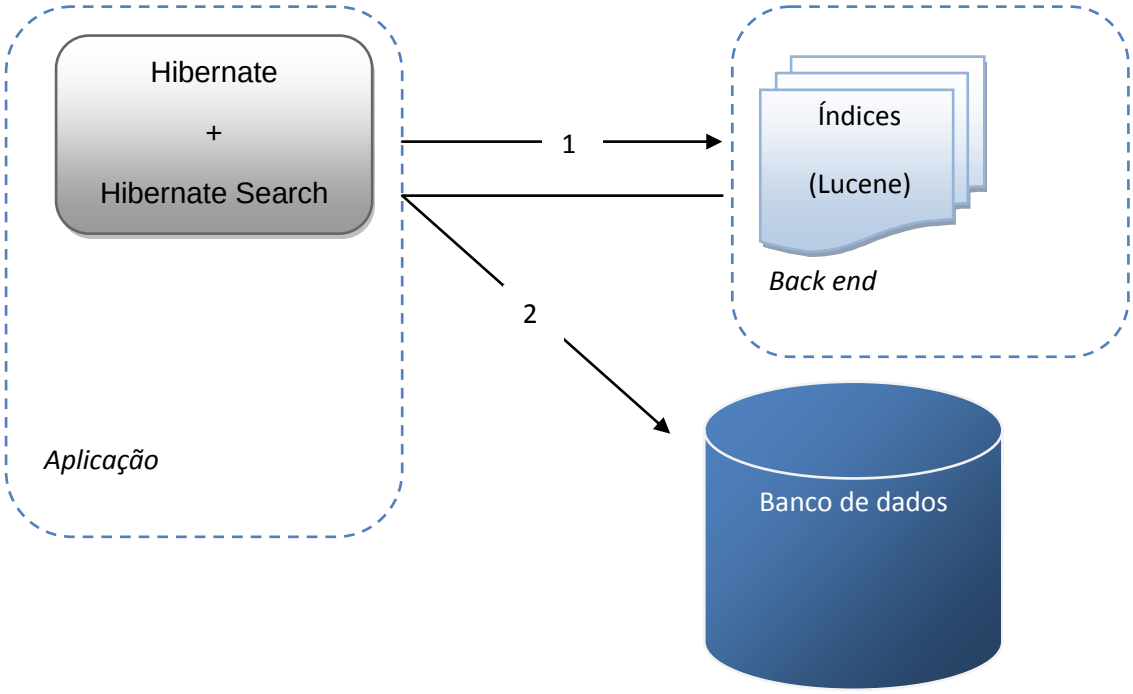

Figura 6 – Principais processos de RI

Processo de Indexação

O processo de indexação gera arquivos de textos para registrar os arquivos invertidos. Posteriormente, o Lucene acessará esses arquivos, que poderão armazenar alguns dados relativos ao registro, em vez de fazer consultas diretas no banco de dados. As etapas de indexação são:

1) O Hibernate fornece uma sessão (instância de Session) para aplicação. Nela são armazenados comandos e o Hibernate se encarrega de gerenciar o banco de dados. Essa sessão é encapsulada no FullTextSession, disponibilizado pelo HibernateSearch. Trata-se de uma extensão de Session que pode executar as funções de busca e indexação.

2) Com a instância do FullTextSession, o Hibernate fornece um canal por onde a aplicação comunica com o banco de dados por meio de uma instância de Transaction.
3) Usando a sessão, a aplicação solicita ao Hibernate os dados do banco para criar os índices. Utilizando as anotações (Hibernate Annotations), é possível instanciar objetos (POJO) que representem os registros das entidades.
4) Com os dados em POJO, a aplicação recorre ao Lucene para gerar os índices. Como o objetivo do Hibernate Search é prover um arcabouço responsável pela integração do banco de dados, a indexação é feita pelo FullTextSession.

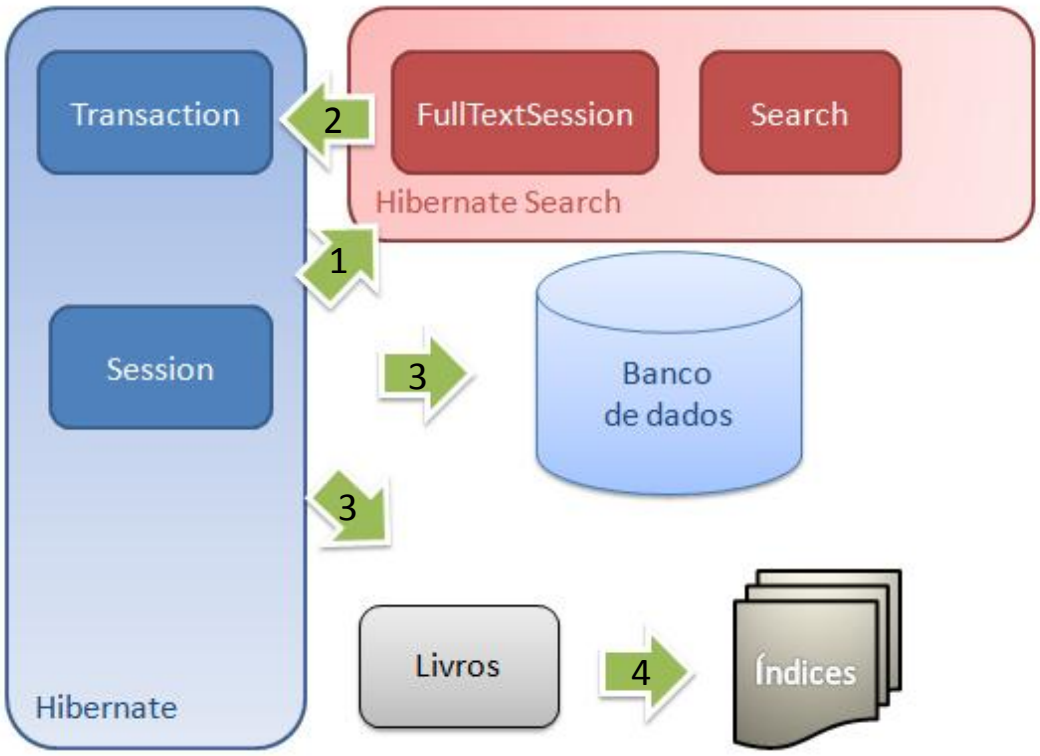

Figura 7 - Processo de indexação de livros

Processo de busca

O processo de busca segue as seguintes etapas:

1) São obtidos: **Termo**, expressão que se deseja procurar no banco de dados; **Parâmetros**, algumas especificações, como forma de ordenação ou paginação que o resultado da consulta deve obedecer; **Field**, campos nos quais a busca será realizada.
2) No Lucene, define-se um analisador, que é responsável por executar operações de texto, tais como remoção de ***stopwords***, inclusão de sinônimos e ***stemming***. O

Lucene sugere implementações próprias para cada idioma, mas não aprofundaremos muito sobre esse tópico por não ser o ponto central deste trabalho. (GOSPODNETIC e HATCHER, 2005, p.102)

3) O próximo passo é criar um ***QueryParser***, que com o auxílio do ***Analyser,*** executa o parsing das informações do termo da busca.
4) Com o QueryParser pronto, obtêm-se uma instância de ***Query*** do Lucene.
5) A Query é repassada para o ***FullTextQuery*** do HibernateSearch e este se encarrega de obter os livros a partir dos índices.

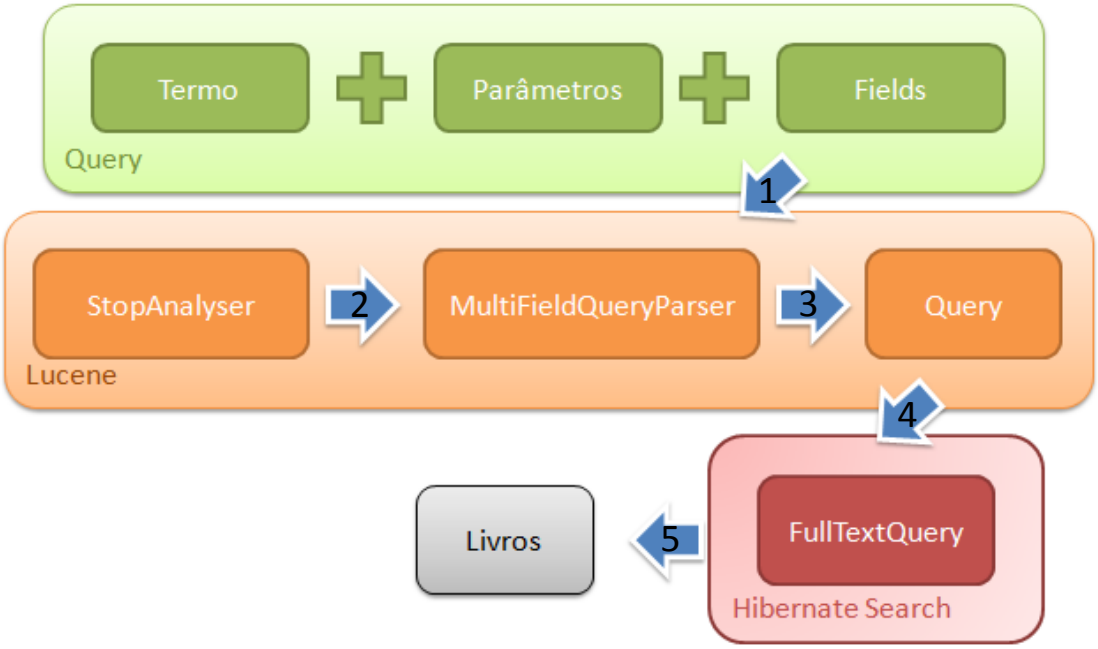

Figura 8 - Processo de busca de livros

3.4. Lucene

O Lucene é um projeto de código aberto da Apache implementado em Java. Trata-se de uma biblioteca de Recuperação de Informação, que provê os meios de indexação e busca por meio de uma API simples e robusta.

3.4.1. Score

O *score* do Lucene é a forma de avaliar a similaridade de um resultado da busca com a sua respectiva consulta. É uma combinação do **Modelo Vetorial** e do **Modelo Booleano**. O *score* de uma consulta *q* para um documento *d* correlaciona à distância do

cosseno ou ao produto escalar entre vetores de documentos e consultas no Modelo Vetorial, utilizando a estratégia de atribuição de pesos de termos *tf-idf*. Sua fórmula é dada por:

$$s(q,d) = coord(q,d).queryNorm(q,d) \sum_{t\ in\ d} (tf(t\ in\ d).idf(t)^2.t.boost().norm(t,d))$$

Termo	Descrição
tf(t in d) *term frequency*	O número de vezes que o termo *t* ocorre no documento *d*
idf(t in d) *inverse document frequency*	O número de documentos que contêm o termo *t*
coord(q, d)	Baseado na quantidade de termos da consulta *q* que são encontrados no documento *d*
queryNorm(q, d)	Usada para permitir a comparação entre os *scores* das consultas
t.boost()	Atribui peso para o termo *t* na consulta *q* em tempo de busca
norm(t, d)	Encapsula alguns fatores de *boost* em tempo de indexação

Tabela 5 - Descrição dos termos da fórmula do *score*

Para ilustrarmos o cálculo do *score*, consideremos o exemplo da consulta "Information Retrieval" e o registro do livro recuperado cujo título é "Information analysis and retrieval". Para cada termo da consulta, localizado no documento recuperado, calcula-se um valor dado por:

$\boldsymbol{weight} = term\ weight\ \times field\ weight$, onde:

$$\boldsymbol{term\ weight} = query\ norm\ \times idf\ \ e\ \ \boldsymbol{field\ weight} = tf\ \times idf\ \times field\ norm$$

Term weight representa o valor referente ao termo da consulta e *field weight* representa o valor referente ao campo do documento no qual tal termo foi localizado. Nesse exemplo, os valores de *coord* e *t.boost()* são iguais a 1.

Por fim, o *score* é dado pelo somatório dos valores *weight* de cada termo. No apêndice B, podemos ver os esquemas com os valores obtidos para o exemplo dado.

3.5. Estudo de caso

3.5.1. Projeto Colméia

O projeto Colméia[10] é um sistema de gerenciamento de tarefas de bibliotecas universitárias. Ele está sendo desenvolvido no Instituto de Matemática e Estatística da USP (IME/USP) e coordenado pelo Prof. Eduardo Colli, o projeto teve início em 2002 por alunos da graduação e pós-graduação durante as disciplinas de Laboratório de Programação eXtrema (ministrada pelo Prof. Fabio Kon) e Laboratório de Banco de Dados (graduação, ministrada pelo Prof. João Eduardo Ferreira), além de outros colaboradores em paralelo às aulas.

O banco de dados do projeto Colméia foi escolhido como estudo de caso, pois apresenta os pré-requisitos necessários para aplicarmos nossos estudos:

- Um banco de dados com grande volume de dados (cerca de 40 mil livros) e estruturalmente complexo;
- Utilização dos arcabouços Hibernate e Hibernate Annotations para realizar a persistência de dados.

Logo, implementamos um módulo no projeto, com uma interface gráfica que permite aos usuários realizarem buscas de livros do acervo da biblioteca, e que utiliza o Hibernate Search para integrar as funcionalidades de RI do Lucene no banco de dados.

[10] PROJETO Colméia. Disponível em: <http://colmeia.incubadora.fapesp.br/portal>. Acesso em: 18/Nov/2008

3.5.2. Resultados

Com o intuito de avaliar o desempenho da busca indexada, realizamos uma série de buscas, num total de 110 mil consultas, de registros de livros do banco de dados do projeto Colméia.

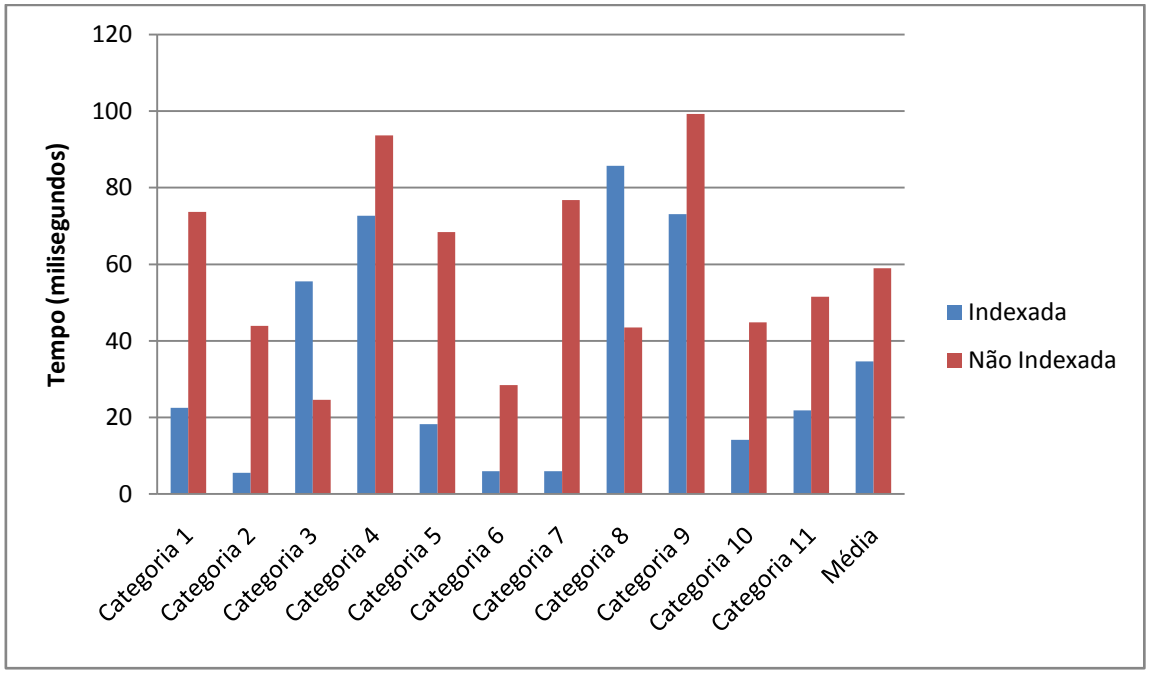

Figura 9 – Resultados dos testes de busca sem utilizar paginação

As consultas foram separadas em 11 categorias, de acordo com os parâmetros utilizados:

- Categoria 1: busca por título
- Categoria 2: busca por autor
- Categoria 3: busca por editora
- Categoria 4: busca por assunto
- Categoria 5: busca por título e autor
- Categoria 6: busca por título e editora
- Categoria 7: busca por título e assunto
- Categoria 8: busca por autor e editora
- Categoria 9: busca por autor e assunto
- Categoria 10: busca por editora e assunto
- Categoria 11: busca por autor, editora e assunto

Como podemos ver pelo gráfico da Figura 8, a busca indexada não teve um desempenho muito expressivo em algumas categorias, principalmente nas categorias 3 e 8, nas quais obteve um desempenho pior que a busca não indexada, o que era inesperado. Porém, ao analisarmos as consultas utilizadas nos testes, vimos que algumas delas retornavam um grande número de resultados. Logo, a provável causa dessa perda de desempenho é o fato do Hibernate Search ter que processar todos os elementos destes resultados tratados pelo Lucene. Entretanto, numa aplicação real, os resultados não serão processados todos de uma vez, portanto a busca indexada continuará sendo rápida. De fato, ao configurarmos os parâmetros de paginação de resultados e otimizarmos o índice, obtivemos os seguintes resultados para a mesma série de buscas anterior, conforme ilustra a Figura 9.

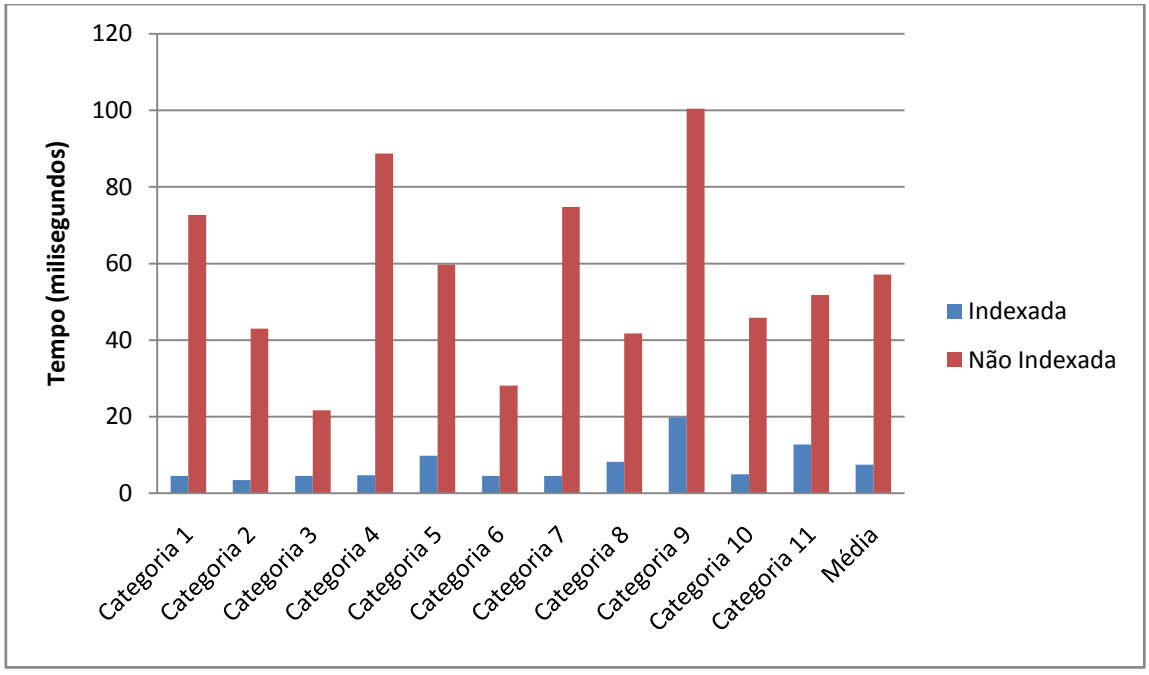

Figura 10 – Resultados dos testes de busca utilizando paginação

É importante observarmos que todos os testes foram realizados com o servidor de banco de dados sendo executado localmente à aplicação. Em um ambiente de execução real, isto é, com o servidor de banco de dados executando remotamente, a diferença de desempenho entre os dois tipos de busca tende a aumentar, devido ao tempo adicional da comunicação via rede.

4. Conclusão

A área de RI vem sendo estudada desde a década de 60 e, por um período, foi visto como uma área de interesse apenas de bibliotecários e pesquisadores da área da ciência da informação. Porém, os avanços tecnológicos na biologia, na comunicação e na computação trouxeram novos desafios que re-despertaram o interesse nos estudos de RI.

O Lucene é um dos exemplos de aplicação das pesquisas em RI. Com este trabalho, conseguimos verificar que se trata de uma implementação fiel aos conceitos que estudamos durante o decorrer do ano. Além disso, o Hibernate Search permitiu que estes conceitos pudessem ser aplicados num ambiente de banco de dados de forma simples e transparente.

Com a implementação de um módulo de busca indexada no projeto Colméia, nós conseguimos não só acelerar o processo de busca de livros, como também enriquecer a forma como recuperar as informações do banco de dados, superando as limitações de consultas possíveis através da linguagem SQL.

Referências Bibliográficas

BAEZA-YATES, Ricardo; RIBEIRO-NETO, Berthier. Modern Information Retrieval. 1st Edition. Harlow: Addison Wesley, 1999.

BAVER, Christian; KING, Gavin – Java Persistence with Hibernate. Revised Edition of Hibernate in Action. New York: Manning, 2007.

ELMASRI, Ramez A.; NAVATHE, Shankrant B. Fundamentals of Database Systems. Third Edition. Boston: Addison Wesley, 1999.

GOSPODNETIC, Otis; HATCHER, Erik. Lucene in Action. Greenwich: Manning, 2005.

KNUTH, Donald E. *The art of computer programming*: Sorting and searching, Reading, Mass.: Addison-Wesley, 1973, v.3.

SHATKAY, Hagit; EDWARDS Stephen; BOGUSKI, Mark. Information Retrieval Meets Gene Analysis. *IEEE Intelligent Systems*, Piscataway, NJ, v. 17, nº 2, p. 45-53, Mar./Apr. 2002.

Apêndice A

Anotações básicas para entidades

```
@Entity
@Table(name = "livro", schema = "public")
public class Livro implements java.io.Serializable {
     private Long idLivro;
     private String tituloOriginal;
     ...
     @Id
     @Column(name = "id_livro", nullable = false)
     public Long getIdLivro() {
        return this.idLivro;
     }
     @Column(name = "titulo_original", length = 300)
     public String getTituloOriginal() {
        return this.tituloOriginal;
     }
```

Quadro 1 - Trecho de código adaptado que ilustra anotações básicas para entidades do Hibernate

Anotações básicas para relacionamentos

```
   @OneToOne
       public ItemAcervo getItemAcervo() {
        return itemAcervo;
    }
    @OneToMany(fetch=FetchType.LAZY, mappedBy =
"livro")
    public Set<Subtitulo> getSubtitulos() {
        return subtitulos;
    }
    @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "id_tp_evento")
    public TpEvento getTpEvento() {
        return this.tpEvento;
    }

    @ManyToMany(fetch = FetchType.LAZY, mappedBy =
"obras")
    public Set<Lingua> getLinguas() {
        return this.linguas;
```

Quadro 2 - Trecho de código adaptado para ilustrar anotações básicas para relacionamentos do Hibernate

Anotações básicas para indexação

```
@Entity
@Indexed
@Table(name = "livro", schema = "public")
public class Livro implements java.io.Serializable {
    private Long idLivro;
    private String tituloOriginal;
    ...
    @Id
    @DocumentId
    @Column(name = "id_livro", nullable = false)
    public Long getIdLivro() {
        return this.idLivro;
    }
    @Field(index=Index.TOKENIZED, store=Store.YES)
    @Column(name = "titulo_original", length = 300)
    public String getTituloOriginal() {
        return this.tituloOriginal;
    }
```

Quadro 3 - Trecho de código adaptado para ilustrar anotações básicas do Hibernate Search

Apêndice B

Descrição do *score* de um registro

Query: Information Retrieval
Resultado: Information analysis and retrieval
4.734231 = (MATCH) sum of:
1.9067632 = (MATCH) weight(tituloOriginal:information in 2099), product of:
0.6346345 = queryWeight(tituloOriginal:information), product of:
6.009012 = idf(docFreq=36, numDocs=5541)
0.105613776 = queryNorm
3.004506 = (MATCH) fieldWeight(tituloOriginal:information in 2099), product of:
1.0 = tf(termFreq(tituloOriginal:information)=1)
6.009012 = idf(docFreq=36, numDocs=5541)
0.5 = fieldNorm(field=tituloOriginal, doc=2099)
2.8274677 = (MATCH) weight(tituloOriginal:retrieval in 2099), product of:
0.7728124 = queryWeight(tituloOriginal:retrieval), product of:
7.317345 = idf(docFreq=9, numDocs=5541)
0.105613776 = queryNorm
3.6586726 = (MATCH) fieldWeight(tituloOriginal:retrieval in 2099), product of:
1.0 = tf(termFreq(tituloOriginal:retrieval)=1)
7.317345 = idf(docFreq=9, numDocs=5541)
0.5 = fieldNorm(field=tituloOriginal, doc=2099)

Quadro 4 – Um exemplo de *score* obtido por meio da consulta "Information Retrieval"

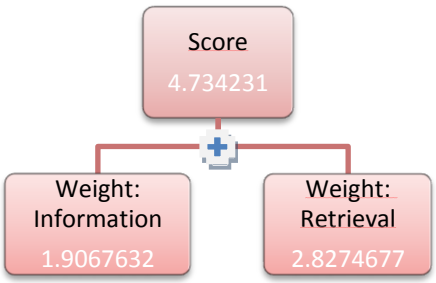

Diagrama 1 - *Score* do registro é dado pelo somatório dos *subscores* de cada termo da consulta

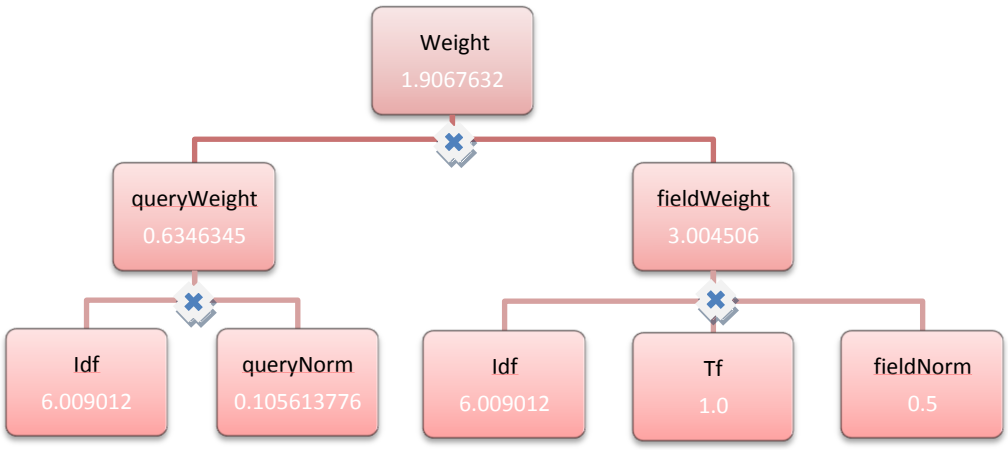

Diagrama 2 - Detalhamento de um *subscore* de um termo da consulta